

CS 473: Algorithms^①

Sariel Har-Peled

November 28, 2018

^①This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Preface

This manuscript is a collection of class notes for the (no longer required graduate) course “473G/573 Graduate Algorithms” taught in the University of Illinois, Urbana-Champaign, in 1. Spring 2006, 2. Fall 07, 3. Fall 09, 4. Fall 10, 5. Fall 13, and 6. Fall 14.

Class notes for algorithms class are as common as mushrooms after a rain. I have no plan of publishing these class notes in any form except on the web. In particular, Jeff Erickson has class notes for 374/473 which are better written and cover some of the topics in this manuscript (but naturally, I prefer my exposition over his).

My reasons in writing the class notes are to (i) avoid the use of a (prohibitly expensive) book in this class, (ii) cover some topics in a way that deviates from the standard exposition, and (iii) have a clear description of the material covered. In particular, as far as I know, no book covers all the topics discussed here. Also, this manuscript is available (on the web) in more convenient lecture notes form, where every lecture has its own chapter.

Most of the topics covered are core topics that I believe every graduate student in computer science should know about. This includes NP-Completeness, dynamic programming, approximation algorithms, randomized algorithms and linear programming. Other topics on the other hand are more optional and are nice to know about. This includes topics like network flow, minimum-cost network flow, and union-find. Nevertheless, I strongly believe that knowing all these topics is useful for carrying out any worthwhile research in any subfield of computer science.

Teaching such a class always involve choosing what not to cover. Some other topics that might be worthy of presentation include advanced data-structures, computational geometry, etc – the list goes on. Since this course is for general consumption, more theoretical topics were left out (e.g., expanders, derandomization, etc).

In particular, these class notes cover way more than can be covered in one semester. For my own sanity, I try to cover some new material every semester I teach this class. Furthermore, some of the notes contains more detail than I cover in class.

In any case, these class notes should be taken for what they are. A short (and sometime dense) tour of some key topics in algorithms. The interested reader should seek other sources to pursue them further.

If you find any typos, mistakes, errors, or lies, please email me.

Acknowledgments

(No preface is complete without them.) I would like to thank the students in the class for their input, which helped in discovering numerous typos and errors in the manuscript. Furthermore, the content was greatly effected by numerous insightful discussions with Chandra Chekuri, Jeff Erickson, and Edgar Ramos.

Copyright

This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

— Sariel Har-Peled
December 2014, Urbana, IL

Contents

Preface	1
Contents	2
I NP Completeness	13
1 NP Completeness I	13
1.1 Introduction	13
1.2 Complexity classes	15
1.2.1 Reductions	16
1.3 More NP-COMplete problems	17
1.3.1 3SAT	17
1.4 Bibliographical Notes	19
2 NP Completeness II	19
2.1 Max-Clique	19
2.2 Independent Set	21
2.3 Vertex Cover	21
2.4 Graph Coloring	22
3 NP Completeness III	24
3.1 Hamiltonian Cycle	24
3.2 Traveling Salesman Problem	26
3.3 Subset Sum	26
3.4 3 dimensional Matching (3DM)	27
3.5 Partition	28
3.6 Some other problems	28
II Dynamic programming	29
4 Dynamic programming	29
4.1 Basic Idea - Partition Number	29
4.1.1 A Short sermon on memoization	31
4.2 Example – Fibonacci numbers	31
4.2.1 Why, where, and when?	31
4.2.2 Computing Fibonacci numbers	32
4.3 Edit Distance	33
4.3.1 Shortest path in a DAG and dynamic programming	35
5 Dynamic programming II - The Recursion Strikes Back	36

5.1	Optimal search trees	36
5.2	Optimal Triangulations	37
5.3	Matrix Multiplication	38
5.4	Longest Ascending Subsequence	39
5.5	Pattern Matching	39
5.6	Slightly faster TSP algorithm via dynamic programming	40
III Approximation algorithms		42
6	Approximation algorithms	42
6.1	Greedy algorithms and approximation algorithms	42
6.1.1	Alternative algorithm – two for the price of one	44
6.2	Fixed parameter tractability, approximation, and fast exponential time algorithms (to say nothing of the dog)	44
6.2.1	A silly brute force algorithm for vertex cover	44
6.2.2	A fixed parameter tractable algorithm	44
6.2.2.1	Remarks	46
6.3	Approximating maximum matching	46
6.4	Graph diameter	47
6.5	Traveling Salesman Person	47
6.5.1	TSP with the triangle inequality	47
6.5.1.1	A 2-approximation	47
6.5.1.2	A 3/2-approximation to $TSP_{\Delta \neq \text{Min}}$	48
6.6	Biographical Notes	50
7	Approximation algorithms II	50
7.1	Max Exact 3SAT	50
7.2	Approximation Algorithms for Set Cover	52
7.2.1	Guarding an Art Gallery	52
7.2.2	Set Cover	52
7.2.3	Lower bound	53
7.2.4	Just for fun – weighted set cover	54
7.2.4.1	Analysis	54
7.3	Biographical Notes	55
8	Approximation algorithms III	56
8.1	Clustering	56
8.1.1	The approximation algorithm for k -center clustering	57
8.2	Subset Sum	58
8.2.1	On the complexity of ϵ -approximation algorithms	59
8.2.2	Approximating subset-sum	59
8.2.2.1	Bounding the running time of <code>ApproxSubsetSum</code>	61
8.2.2.2	The result	62
8.3	Approximate Bin Packing	62
8.4	Bibliographical notes	63
IV Randomized algorithms		64
9	Randomized Algorithms	64

9.1	Some Probability	64
9.2	Sorting Nuts and Bolts	65
9.2.1	Running time analysis	66
9.2.1.1	Alternative incorrect solution	67
9.2.2	What are randomized algorithms?	67
9.3	Analyzing QuickSort	68
9.4	QuickSelect – median selection in linear time	68
10	Randomized Algorithms II	70
10.1	QuickSort and Treaps with High Probability	70
10.1.1	Proving that an element participates in small number of rounds	70
10.1.2	An alternative proof of the high probability of QuickSort	72
10.2	Chernoff inequality	72
10.2.1	Preliminaries	72
10.2.2	Chernoff inequality	73
10.2.2.1	The Chernoff Bound — General Case	74
10.3	Treaps	75
10.3.1	Construction	75
10.3.2	Operations	75
10.3.2.1	Insertion	76
10.3.2.2	Deletion	76
10.3.2.3	Split	76
10.3.2.4	Meld	76
10.3.3	Summery	77
10.4	Bibliographical Notes	77
11	Hashing	77
11.1	Introduction	78
11.2	Universal Hashing	79
11.2.1	How to build a 2-universal family	80
11.2.1.1	On working modulo prime	80
11.2.1.2	Constructing a family of 2-universal hash functions	81
11.2.1.3	Analysis	81
11.2.1.4	Explanation via pictures	82
11.3	Perfect hashing	82
11.3.1	Some easy calculations	82
11.3.2	Construction of perfect hashing	83
11.3.2.1	Analysis	83
11.4	Bloom filters	84
11.5	Bibliographical notes	84
12	Min Cut	86
12.1	Branching processes – Galton-Watson Process	86
12.1.1	The problem	86
12.1.2	On coloring trees	86
12.2	Min Cut	88
12.2.1	Problem Definition	88
12.2.2	Some Definitions	88
12.3	The Algorithm	88
12.3.1	Analysis	90

12.3.1.1	The probability of success	90
12.3.1.2	Running time analysis.	91
12.4	A faster algorithm	91
12.5	Bibliographical Notes	93
V	Network flow	94
13	Network Flow	94
13.1	Network Flow	94
13.2	Some properties of flows and residual networks	95
13.3	The Ford-Fulkerson method	98
13.4	On maximum flows	98
14	Network Flow II - The Vengeance	99
14.1	Accountability	99
14.2	The Ford-Fulkerson Method	100
14.3	The Edmonds-Karp algorithm	100
14.4	Applications and extensions for Network Flow	102
14.4.1	Maximum Bipartite Matching	102
14.4.2	Extension: Multiple Sources and Sinks	103
15	Network Flow III - Applications	104
15.1	Edge disjoint paths	104
15.1.1	Edge-disjoint paths in a directed graphs	104
15.1.2	Edge-disjoint paths in undirected graphs	105
15.2	Circulations with demands	105
15.2.1	Circulations with demands	105
15.2.1.1	The algorithm for computing a circulation	106
15.3	Circulations with demands and lower bounds	106
15.4	Applications	107
15.4.1	Survey design	107
16	Network Flow IV - Applications II	108
16.1	Airline Scheduling	108
16.1.1	Modeling the problem	109
16.1.2	Solution	109
16.2	Image Segmentation	110
16.3	Project Selection	112
16.3.1	The reduction	112
16.4	Baseball elimination	113
16.4.1	Problem definition	114
16.4.2	Solution	114
16.4.3	A compact proof of a team being eliminated	115
17	Network Flow V - Min-cost flow	116
17.1	Minimum Average Cost Cycle	116
17.2	Potentials	118
17.3	Minimum cost flow	118
17.4	A Strongly Polynomial Time Algorithm for Min-Cost Flow	121
17.5	Analysis of the Algorithm	121

17.5.1	Reduced cost induced by a circulation	122
17.5.2	Bounding the number of iterations	122
17.6	Bibliographical Notes	124
18	Network Flow VI - Min-Cost Flow Applications	124
18.1	Efficient Flow	124
18.2	Efficient Flow with Lower Bounds	125
18.3	Shortest Edge-Disjoint Paths	126
18.4	Covering by Cycles	126
18.5	Minimum weight bipartite matching	126
18.6	The transportation problem	127
VI	Linear Programming	128
19	Linear Programming in Low Dimensions	128
19.1	Some geometry first	128
19.2	Linear programming	129
19.2.1	A solution and how to verify it	130
19.3	Low-dimensional linear programming	131
19.3.1	An algorithm for a restricted case	131
19.3.1.1	Running time analysis	132
19.3.2	The algorithm for the general case	133
20	Linear Programming	134
20.1	Introduction and Motivation	134
20.1.1	History	134
20.1.2	Network flow via linear programming	135
20.2	The Simplex Algorithm	136
20.2.1	Linear program where all the variables are positive	136
20.2.2	Standard form	136
20.2.3	Slack Form	136
20.2.4	The Simplex algorithm by example	137
20.2.4.1	Starting somewhere	140
21	Linear Programming II	140
21.1	The Simplex Algorithm in Detail	140
21.2	The SimplexInner Algorithm	141
21.2.1	Degeneracies	142
21.2.2	Correctness of linear programming	143
21.2.3	On the ellipsoid method and interior point methods	143
21.3	Duality and Linear Programming	143
21.3.1	Duality by Example	143
21.3.2	The Dual Problem	144
21.3.3	The Weak Duality Theorem	144
21.4	The strong duality theorem	145
21.5	Some duality examples	146
21.5.1	Shortest path	146
21.5.2	Set Cover and Packing	147
21.5.3	Network flow	147
21.6	Solving LPs without ever getting into a loop - symbolic perturbations	150

21.6.1	The problem and the basic idea	150
21.6.2	Pivoting as a Gauss elimination step	151
21.6.2.1	Back to the perturbation scheme	151
21.6.2.2	The overall algorithm	152
22	Approximation Algorithms using Linear Programming	152
22.1	Weighted vertex cover	152
22.2	Revisiting Set Cover	154
22.3	Minimizing congestion	155
VII	Miscellaneous topics	157
23	Fast Fourier Transform	157
23.1	Introduction	157
23.2	Computing a polynomial quickly on n values	158
23.2.1	Generating Collapsible Sets	159
23.3	Recovering the polynomial	160
23.4	The Convolution Theorem	162
23.4.1	Complex numbers – a quick reminder	163
24	Sorting Networks	163
24.1	Model of Computation	164
24.2	Sorting with a circuit – a naive solution	164
24.2.1	Definitions	165
24.2.2	Sorting network based on insertion sort	165
24.3	The Zero-One Principle	165
24.4	A bitonic sorting network	166
24.4.1	Merging sequence	168
24.5	Sorting Network	168
24.6	Faster sorting networks	169
25	Union Find	170
25.1	Union-Find	170
25.1.1	Requirements from the data-structure	170
25.1.2	Amortized analysis	170
25.1.3	The data-structure	170
25.2	Analyzing the Union-Find Data-Structure	172
26	Approximate Max Cut	175
26.1	Problem Statement	176
26.1.1	Analysis	176
26.2	Semi-definite programming	178
26.3	Bibliographical Notes	178
27	The Perceptron Algorithm	179
27.1	The perceptron algorithm	179
27.2	Learning A Circle	182
27.3	A Little Bit On VC Dimension	183
27.3.1	Examples	184

VIII	Compression, entropy, and randomness	185
28	Huffman Coding	185
28.1	Huffman coding	185
28.1.1	The algorithm to build Hoffman's code	187
28.1.2	Analysis	187
28.1.3	What do we get	189
28.1.4	A formula for the average size of a code word	189
29	Entropy, Randomness, and Information	189
29.1	Entropy	190
29.1.1	Extracting randomness	192
30	Even more on Entropy, Randomness, and Information	193
30.1	Extracting randomness	193
30.1.1	Enumerating binary strings with j ones	193
30.1.2	Extracting randomness	194
30.2	Bibliographical Notes	195
31	Shannon's theorem	195
31.1	Coding: Shannon's Theorem	195
31.1.0.1	Intuition behind Shanon's theorem	196
31.1.0.2	What is wrong with the above?	197
31.2	Proof of Shannon's theorem	197
31.2.1	How to encode and decode efficiently	197
31.2.1.1	The scheme	197
31.2.1.2	The proof	197
31.2.2	Lower bound on the message size	200
31.3	Bibliographical Notes	200
IX	Miscellaneous topics II	201
32	Matchings	201
32.1	Definitions and basic properties	201
32.1.1	Definitions	201
32.1.2	Matchings and alternating paths	202
32.2	Unweighted matching in bipartite graph	203
32.2.1	The slow algorithm; <code>algSlowMatch</code>	203
32.2.2	The Hopcroft-Karp algorithm	204
32.2.2.1	Some more structural observations	204
32.2.2.2	Improved algorithm	205
32.2.2.3	Extracting many augmenting paths: <code>algExtManyPaths</code>	205
32.2.2.4	The result	208
32.3	Bibliographical notes	208
33	Matchings II	208
33.1	Maximum weight matchings in a bipartite graph	209
33.1.1	On the structure of the problem	209
33.1.2	Maximum Weight Matchings in a bipartite Graph	210
33.1.2.1	Building the residual graph	210

33.1.2.2	The algorithm	210
33.1.3	Faster Algorithm	210
33.2	The Bellman-Ford algorithm - a quick reminder	211
33.3	Maximum size matching in a non-bipartite graph	211
33.3.1	Finding an augmenting path	211
33.3.2	The algorithm	214
33.3.2.1	Running time analysis	214
33.4	Maximum weight matching in a non-bipartite graph	215
34	Lower Bounds	215
34.1	Sorting	215
34.1.1	Decision trees	215
34.1.2	An easier direct argument	217
34.2	Uniqueness	218
34.3	Other lower bounds	219
34.3.1	Algebraic tree model	219
34.3.2	3Sum-Hard	219
35	Backwards analysis	219
35.1	How many times can the minimum change?	220
35.2	Yet another analysis of QuickSort	220
35.3	Closest pair: Backward analysis in action	221
35.3.1	Definitions	221
35.3.2	Back to the problem	221
35.3.3	Slow algorithm	222
35.3.4	Linear time algorithm	223
35.4	Computing a good ordering of the vertices of a graph	224
35.4.1	The algorithm	224
35.4.2	Analysis	224
35.5	Computing nets	225
35.5.1	Basic definitions	225
35.5.1.1	Metric spaces	225
35.5.1.2	Nets	225
35.5.2	Computing nets quickly for a point set in \mathbb{R}^d	225
35.5.3	Computing an r -net in a sparse graph	226
35.5.3.1	The algorithm	226
35.5.3.2	Analysis	227
35.6	Bibliographical notes	228
36	Linear time algorithms	228
36.1	The lowest point above a set of lines	228
36.2	Bibliographical notes	230
37	Streaming	231
37.1	How to sample a stream	231
37.2	Sampling and median selection	232
37.2.1	A median selection with few comparisons	233
37.3	Big data and the streaming model	233
37.4	Heavy hitters	234
37.5	Chernoff inequality	234

X	Exercises	235
	38 Exercises - Prerequisites	235
	38.1 Graph Problems	235
	38.2 Recurrences	237
	38.3 Counting	239
	38.4 O notation and friends	239
	38.5 Probability	240
	38.6 Basic data-structures and algorithms	242
	38.7 General proof thingies	244
	38.8 Miscellaneous	245
	39 Exercises - NP Completeness	246
	39.1 Equivalence of optimization and decision problems	246
	39.2 Showing problems are NP-COMPLETE	247
	39.3 Solving special subcases of NP-COMPLETE problems in polynomial time	248
	40 Exercises - Network Flow	256
	40.1 Network Flow	256
	40.2 Min Cost Flow	265
	41 Exercises - Miscellaneous	267
	41.1 Data structures	267
	41.2 Divide and Conqueror	267
	41.3 Fast Fourier Transform	268
	41.4 Union-Find	269
	41.5 Lower bounds	271
	41.6 Number theory	271
	41.7 Sorting networks	272
	41.8 Max Cut	272
	42 Exercises - Approximation Algorithms	273
	42.1 Greedy algorithms as approximation algorithms	273
	42.2 Approximation for hard problems	274
	43 Randomized Algorithms	276
	43.1 Randomized algorithms	276
	44 Exercises - Linear Programming	279
	44.1 Miscellaneous	279
	44.2 Tedious	279
	45 Exercises - Computational Geometry	283
	45.1 Misc	283
	46 Exercises - Entropy	285
XI	Homeworks/midterm/final	287
	47 Fall 2001	287
	47.1 Homeworks	287

47.1.1	Homework 0	287
47.1.2	Homework 1	293
47.1.3	Homework 2	299
47.1.4	Homework 3	305
47.1.5	Homework 4	314
47.1.6	Homework 5	320
47.1.7	Homework 6	325
47.2	Midterm	331
47.3	Final	332
48	Spring 2002	332
49	Fall 2002	332
50	Spring 2003	332
50.1	Homeworks	332
50.1.1	Homework 0	332
50.1.2	Homework 1	340
50.1.3	Homework 2	349
50.1.4	Homework 3	356
50.1.5	Homework 4	363
50.1.6	Homework 5	367
50.1.7	Homework 6	371
50.2	Midterm 1	374
50.3	Midterm 2	379
50.4	Final	381
51	Fall 2003	383
52	Spring 2005	383
53	Spring 2006	383
54	Fall 2007	383
55	Fall 2009	383
56	Spring 2011	383
57	Fall 2011	383
58	Fall 2012	383
59	Fall 2013	383
60	Spring 2013: CS 473: Fundamental algorithms	383
60.1	Homeworks	383
60.1.1	Homework 0	383
60.1.2	Homework 1	385
60.1.3	Homework 2	387
60.1.4	Homework 3	388
60.1.5	Homework 4	389

60.1.6	Homework 5	390
60.1.7	Homework 6	392
60.1.8	Homework 7	394
60.1.9	Homework 8	396
60.1.10	Homework 9	397
60.1.11	Homework 10	399
60.2	Midterm 1	402
60.3	Midterm 2	404
60.4	Final	408
61	Fall 2014: CS 573 – Graduate algorithms	411
61.1	Homeworks	411
61.1.1	Homework 0	411
61.1.2	Homework 1	412
61.1.3	Homework 2	413
61.1.4	Homework 3	415
61.1.5	Homework 4	417
61.1.6	Homework 5	420
61.2	Midterm	424
61.3	Final	426
62	Fall 2015: CS 473 – Theory II	428
62.1	Homeworks	428
62.1.1	Homework 0	428
62.1.2	Homework 1	429
62.1.3	Homework 2	433
62.1.4	Homework 3	434
62.1.5	Homework 4	436
62.1.6	Homework 5	437
62.1.7	Homework 6	438
62.1.8	Homework 7	440
62.1.9	Homework 8	441
62.1.10	Homework 9	441
62.1.11	Homework 10	442
62.1.12	Homework 11	445
62.2	Midterm	448
62.3	Final	450
	Bibliography	452
	Index	455

Part I

NP Completeness

Chapter 1

NP Completeness I

"Then you must begin a reading program immediately so that you man understand the crises of our age," Ignatius said solemnly. "Begin with the late Romans, including Boethius, of course. Then you should dip rather extensively into early Medieval. You may skip the Renaissance and the Enlightenment. That is mostly dangerous propaganda. Now, that I think about of it, you had better skip the Romantics and the Victorians, too. For the contemporary period, you should study some selected comic books."

"You're fantastic."

"I recommend Batman especially, for he tends to transcend the abysmal society in which he's found himself. His morality is rather rigid, also. I rather respect Batman."

– A confederacy of Dunces, John Kennedy Toole.

1.1. Introduction

The question governing this course, would be the development of efficient algorithms. Hopefully, what is an algorithm is a well understood concept. But what is an *efficient* algorithm? A natural answer (but not the only one!) is an algorithm that runs quickly.

What do we mean by quickly? Well, we would like our algorithm to:

- (A) Scale with input size. That is, it should be able to handle large and hopefully huge inputs.
- (B) Low level implementation details should not matter, since they correspond to small improvements in performance. Since faster CPUs keep appearing it follows that such improvements would (usually) be taken care of by hardware.
- (C) What we will really care about are asymptotic running time. Explicitly, polynomial time.

In our discussion, we will consider the input size to be n , and we would like to bound the overall running time by a function of n which is asymptotically as small as possible. An algorithm with better asymptotic running time would be considered to be *better*.

Example 1.1.1. It is illuminating to consider a concrete example. So assume we have an algorithm for a problem that needs to perform $c2^n$ operations to handle an input of size n , where c is a small constant (say 10). Let assume that we have a CPU that can do 10^9 operations a second. (A somewhat conservative assumption, as

Input size	n^2 ops	n^3 ops	n^4 ops	2^n ops	$n!$ ops
5	0 secs	0 secs	0 secs	0 secs	0 secs
20	0 secs	0 secs	0 secs	0 secs	16 mins
30	0 secs	0 secs	0 secs	0 secs	$3 \cdot 10^9$ years
50	0 secs	0 secs	0 secs	0 secs	never
60	0 secs	0 secs	0 secs	7 mins	never
70	0 secs	0 secs	0 secs	5 days	never
80	0 secs	0 secs	0 secs	15.3 years	never
90	0 secs	0 secs	0 secs	15,701 years	never
100	0 secs	0 secs	0 secs	10^7 years	never
8000	0 secs	0 secs	1 secs	never	never
16000	0 secs	0 secs	26 secs	never	never
32000	0 secs	0 secs	6 mins	never	never
64000	0 secs	0 secs	111 mins	never	never
200,000	0 secs	3 secs	7 days	never	never
2,000,000	0 secs	53 mins	202.943 years	never	never
10^8	4 secs	12.6839 years	10^9 years	never	never
10^9	6 mins	12683.9 years	10^{13} years	never	never

Figure 1.1: Running time as function of input size. Algorithms with exponential running times can handle only relatively small inputs. We assume here that the computer can do $2.5 \cdot 10^{15}$ operations per second, and the functions are the exact number of operations performed. Remember – never is a long time to wait for a computation to be completed.

currently [Jan 2006]^①, the blue-gene supercomputer can do about $3 \cdot 10^{14}$ floating-point operations a second. Since this super computer has about 131,072 CPUs, it is not something you would have on your desktop any time soon.) Since $2^{10} \approx 10^3$, you have that our (cheap) computer can solve in (roughly) 10 seconds a problem of size $n = 27$.

But what if we increase the problem size to $n = 54$? This would take our computer about 3 million years to solve. (It is better to just wait for faster computers to show up, and then try to solve the problem. Although there are good reasons to believe that the exponential growth in computer performance we saw in the last 40 years is about to end. Thus, unless a substantial breakthrough in computing happens, it might be that solving problems of size, say, $n = 100$ for this problem would forever be outside our reach.)

The situation dramatically change if we consider an algorithm with running time $10n^2$. Then, in one second our computer can handle input of size $n = 10^4$. Problem of size $n = 10^8$ can be solved in $10n^2/10^9 = 10^{17-9} = 10^8$ which is about 3 years of computing (but blue-gene might be able to solve it in less than 20 minutes!).

Thus, algorithms that have asymptotically a polynomial running time (i.e., the algorithms running time is bounded by $O(n^c)$ where c is a constant) are able to solve large instances of the input and can solve the problem even if the problem size increases dramatically.

Can we solve all problems in polynomial time? The answer to this question is unfortunately no. There are several synthetic examples of this, but it is believed that a large class of important problems can not be solved in polynomial time.

Circuit Satisfiability

Instance: A circuit C with m inputs

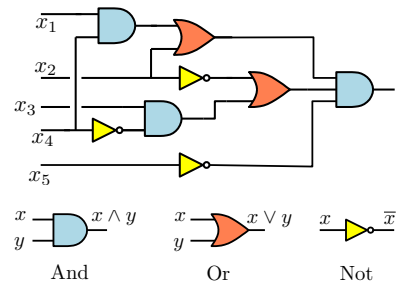
Question: Is there an input for C such that C returns true for it.

^①But the recently announced Super Computer that would be completed in 2012 in Urbana, is naturally way faster. It supposedly would do 10^{15} operations a second (i.e., petaflop). Blue-gene probably can not sustain its theoretical speed stated above, which is only slightly slower.

As a concrete example, consider the circuit depicted on the right.

Currently, all solutions known to **Circuit Satisfiability** require checking all possibilities, requiring (roughly) 2^m time. Which is exponential time and too slow to be useful in solving large instances of the problem.

This leads us to the most important open question in theoretical computer science:



Question 1.1.2. Can one solve **Circuit Satisfiability** in polynomial time?

The common belief is that **Circuit Satisfiability** can **NOT** be solved in polynomial time. **Circuit Satisfiability** has two interesting properties.

- (A) Given a supposed positive solution, with a detailed assignment (i.e., proof): $x_1 \leftarrow 0, x_2 \leftarrow 1, \dots, x_m \leftarrow 1$ one can verify in polynomial time if this assignment really satisfies C . This is done by computing what every gate in the circuit what its output is for this input. Thus, computing the output of C for its input. This requires evaluating the gates of C in the right order, and there are some technicalities involved, which we are ignoring. (But you should verify that you know how to write a program that does that efficiently.)

Intuitively, this is the difference in hardness between coming up with a proof (hard), and checking that a proof is correct (easy).

- (B) It is a **decision problem**. For a specific input an algorithm that solves this problem has to output either **TRUE** or **FALSE**.

1.2. Complexity classes

Definition 1.2.1 (P: Polynomial time). Let **P** denote is the class of all decision problems that can be solved in polynomial time in the size of the input.

Definition 1.2.2 (NP: Nondeterministic Polynomial time). Let **NP** be the class of all decision problems that can be verified in polynomial time. Namely, for an input of size n , if the solution to the given instance is true, one (i.e., an oracle) can provide you with a proof (of polynomial length!) that the answer is indeed **TRUE** for this instance. Furthermore, you can verify this proof in polynomial time in the length of the proof.

Clearly, if a decision problem can be solved in polynomial time, then it can be verified in polynomial time. Thus, $P \subseteq NP$.

Remark. The notation **NP** stands for Non-deterministic Polynomial. The name come from a formal definition of this class using Turing machines where the machines first guesses (i.e., the non-deterministic stage) the proof that the instance is **TRUE**, and then the algorithm verifies the proof.

Definition 1.2.3 (co-NP). The class **co-NP** is the opposite of **NP** – if the answer is **FALSE**, then there exists a short proof for this negative answer, and this proof can be verified in polynomial time.

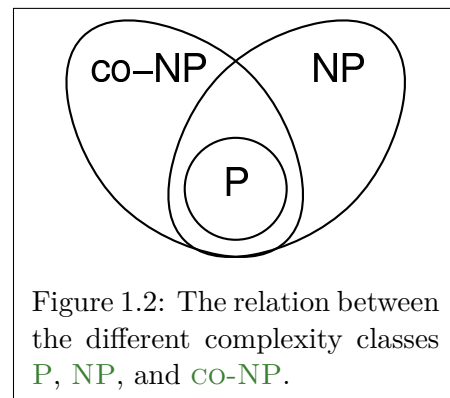


Figure 1.2: The relation between the different complexity classes **P**, **NP**, and **co-NP**.

See **Figure 1.2** for the currently *believed* relationship between these classes (of course, as mentioned above, $P \subseteq NP$ and $P \subseteq co-NP$ is easy to verify). Note, that it is quite possible that $P = NP = co-NP$, although this would be extremely surprising.

Definition 1.2.4. A problem Π is **NP-HARD**, if being able to solve Π in polynomial time implies that $P = NP$.

Question 1.2.5. Are there any problems which are **NP-HARD**?

Intuitively, being **NP-HARD** implies that a problem is ridiculously hard. Conceptually, it would imply that proving and verifying are equally hard - which nobody that did CS 473 believes is true.

In particular, a problem which is **NP-HARD** is at least as hard as ALL the problems in **NP**, as such it is safe to assume, based on overwhelming evidence that it can not be solved in polynomial time.

Theorem 1.2.6 (Cook’s Theorem). *Circuit Satisfiability is NP-HARD.*

Definition 1.2.7. A problem Π is **NP-COMPLETE** (**NPC** in short) if it is both **NP-HARD** and in **NP**.

Clearly, **Circuit Satisfiability** is **NP-COMPLETE**, since we can verify a positive solution in polynomial time in the size of the circuit,

By now, thousands of problems have been shown to be **NP-COMPLETE**. It is extremely unlikely that any of them can be solved in polynomial time.

Definition 1.2.8. In the **formula satisfiability** problem, (a.k.a. **SAT**) we are given a formula, for example:

$$(a \vee b \vee c \vee \bar{d}) \iff ((b \wedge \bar{c}) \vee \overline{(a \implies d)} \vee (c \neq a \wedge b))$$

and the question is whether we can find an assignment to the variables a, b, c, \dots such that the formula evaluates to **TRUE**.

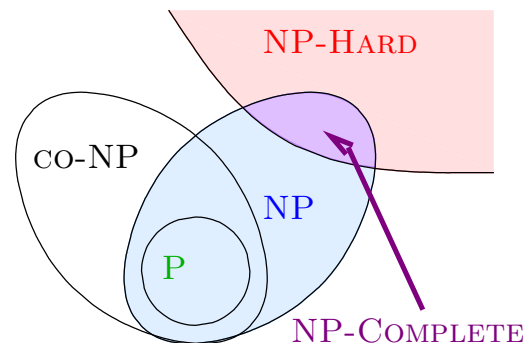


Figure 1.3: The relation between the complexity classes.

It seems that **SAT** and **Circuit Satisfiability** are “similar” and as such both should be **NP-HARD**.

Remark 1.2.9. Cook’s theorem implies something somewhat stronger than implied by the above statement. Specifically, for any problem in **NP**, there is a polynomial time reduction to **Circuit Satisfiability**. Thus, the reader can think about **NPC** problems has being equivalent under polynomial time reductions.

1.2.1. Reductions

Let A and B be two decision problems.

Given an input I for problem A , a *reduction* is a transformation of the input I into a new input I' , such that

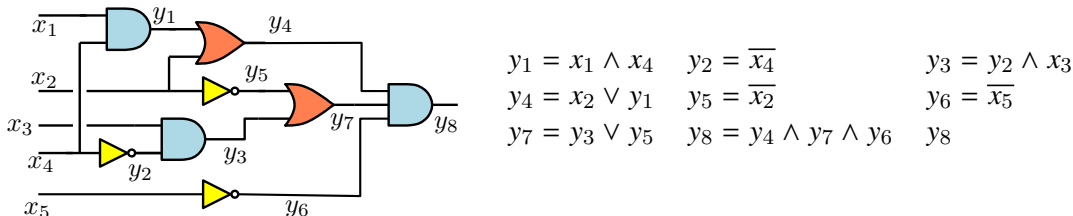
$$A(I) \text{ is TRUE} \iff B(I') \text{ is TRUE.}$$

Thus, one can solve A by first transforming and input I into an input I' of B , and solving $B(I')$.

This idea of using reductions is omnipresent, and used almost in any program you write.

Let $T : I \rightarrow I'$ be the input transformation that maps A into B . How fast is T ? Well, for our nefarious purposes we need **polynomial reductions**; that is, reductions that take polynomial time.

For example, given an instance of **Circuit Satisfiability**, we would like to generate an equivalent formula. We will explicitly write down what the circuit computes in a formula form. To see how to do this, consider the following example.



We introduced a variable for each wire in the circuit, and we wrote down explicitly what each gate computes. Namely, we wrote a formula for each gate, which holds only if the gate computes correctly the output for its given input.

The circuit is satisfiable **if and only if** there is an assignment such that all the above formulas hold. Alternatively, the circuit is satisfiable if and only if the following (single) formula is satisfiable

$$\begin{aligned} (y_1 = x_1 \wedge x_4) \wedge (y_2 = \bar{x}_4) \wedge (y_3 = y_2 \wedge x_3) \\ \wedge (y_4 = x_2 \vee y_1) \wedge (y_5 = \bar{x}_2) \\ \wedge (y_6 = \bar{x}_5) \wedge (y_7 = y_3 \vee y_5) \\ \wedge (y_8 = y_4 \wedge y_7 \wedge y_6) \wedge y_8. \end{aligned}$$

It is easy to verify that this transformation can be done in polynomial time.

The resulting reduction is depicted in Figure 1.4.

Namely, given a solver for SAT that runs in $T_{SAT}(n)$, we can solve the CSAT problem in time

$$T_{CSAT}(n) \leq O(n) + T_{SAT}(O(n)),$$

where n is the size of the input circuit. Namely, if we have polynomial time algorithm that solves SAT then we can solve CSAT in polynomial time.

Another way of looking at it, is that we believe that solving CSAT requires exponential time; namely, $T_{CSAT}(n) \geq 2^n$. Which implies by the above reduction that

$$2^n \leq T_{CSAT}(n) \leq O(n) + T_{SAT}(O(n)).$$

Namely, $T_{SAT}(n) \geq 2^{n/c} - O(n)$, where c is some positive constant. Namely, if we believe that we need exponential time to solve CSAT then we need exponential time to solve SAT.

This implies that if $SAT \in P$ then $CSAT \in P$.

We just proved that SAT is as hard as CSAT. Clearly, $SAT \in NP$ which implies the following theorem.

Theorem 1.2.10. *SAT (formula satisfiability) is NP-COMplete.*

1.3. More NP-Complete problems

1.3.1. 3SAT

A boolean formula is in conjunctive normal form (CNF) if it is a conjunction (AND) of several *clauses*, where a clause is the disjunction (or) of several *literals*, and a literal is either a variable or a negation of a variable. For example, the following is a CNF formula:

$$\overbrace{(a \vee b \vee \bar{c})}^{clause} \wedge (a \vee \bar{e}) \wedge (c \vee e).$$

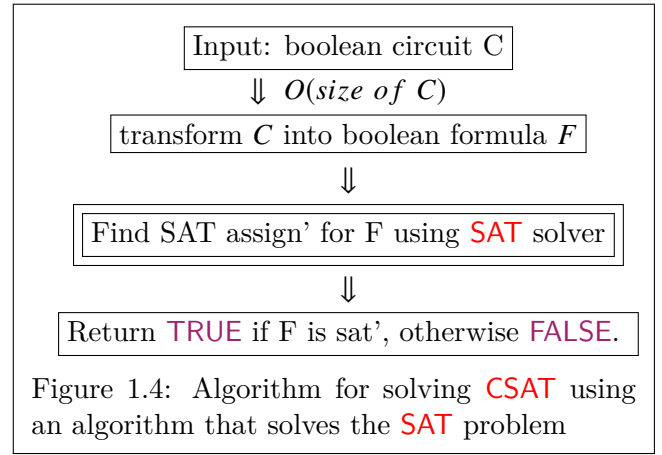
Definition 1.3.1. 3CNF formula is a CNF formula with *exactly* three literals in each clause.

The problem 3SAT is formula satisfiability when the formula is restricted to be a 3CNF formula.

Theorem 1.3.2. *3SAT is NP-COMplete.*

Proof: First, it is easy to verify that 3SAT is in NP.

Next, we will show that 3SAT is NP-COMplete by a reduction from CSAT (i.e., **Circuit Satisfiability**). As such, our input is a circuit C of size n . We will transform it into a 3CNF in several steps:



- (A) Make sure every AND/OR gate has only two inputs. If (say) an AND gate have more inputs, we replace it by cascaded tree of AND gates, each one of degree two.
- (B) Write down the circuit as a formula by traversing the circuit, as was done for **SAT**. Let F be the resulting formula.

A clause corresponding to a gate in F will be of the following forms: (i) $a = b \wedge c$ if it corresponds to an AND gate, (ii) $a = b \vee c$ if it corresponds to an OR gate, and (iii) $a = \bar{b}$ if it corresponds to a NOT gate. Notice, that except for the single clause corresponding to the output of the circuit, all clauses are of this form. The clause that corresponds to the output is a single variable.

- (C) Change every gate clause into several CNF clauses.
- (i) For example, an AND gate clause of the form $a = b \wedge c$ will be translated into

$$(a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c). \quad (1.1)$$

Note that Eq. (1.1) is true if and only if $a = b \wedge c$ is true. Namely, we can replace the clause $a = b \wedge c$ in F by Eq. (1.1).

- (ii) Similarly, an OR gate clause the form $a = b \vee c$ in F will be transformed into

$$(\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}).$$

- (iii) Finally, a clause $a = \bar{b}$, corresponding to a NOT gate, will be transformed into

$$(a \vee b) \wedge (\bar{a} \vee \bar{b}).$$

- (D) Make sure every clause is exactly three literals. Thus, a single variable clause a would be replaced by

$$(a \vee x \vee y) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee \bar{y}),$$

by introducing two new dummy variables x and y . And a two variable clause $a \vee b$ would be replaced by

$$(a \vee b \vee y) \wedge (a \vee b \vee \bar{y}),$$

by introducing the dummy variable y .

This completes the reduction, and results in a new 3CNF formula G which is satisfiable if and only if the original circuit C is satisfiable. The reduction is depicted in Figure 1.5. Namely, we generated an equivalent 3CNF to the original circuit. We conclude that if $T_{3SAT}(n)$ is the time required to solve 3SAT then

$$T_{CSAT}(n) \leq O(n) + T_{3SAT}(O(n)),$$

which implies that if we have a polynomial time algorithm for 3SAT, we would solve CSAT is polynomial time. Namely, 3SAT is NP-COMplete. ■

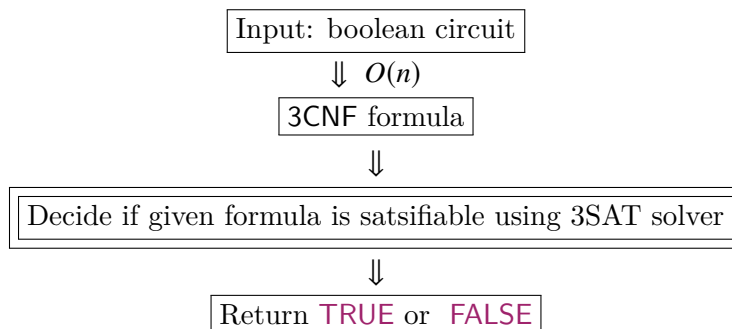


Figure 1.5: Reduction from CSAT to 3SAT

1.4. Bibliographical Notes

Cook's theorem was proved by Stephen Cook (http://en.wikipedia.org/wiki/Stephen_Cook). It was proved independently by Leonid Levin (http://en.wikipedia.org/wiki/Leonid_Levin) more or less in the same time. Thus, this theorem should be referred to as the Cook-Levin theorem.

The standard text on this topic is [GJ90]. Another useful book is [ACG⁺99], which is a more recent and more updated, and contain more advanced stuff.

Chapter 2

NP Completeness II

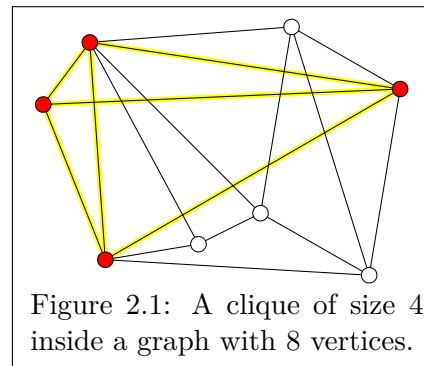
2.1. Max-Clique

We remind the reader, that a *clique* is a complete graph, where every pair of vertices are connected by an edge. The **MaxClique** problem asks what is the largest clique appearing as a subgraph of G . See Figure 2.1.

MaxClique

Instance: A graph G

Question: What is the largest number of nodes in G forming a complete subgraph?



Note that **MaxClique** is an *optimization* problem, since the output of the algorithm is a number and not just true/false.

The first natural question, is how to solve **MaxClique**. A naive algorithm would work by enumerating all subsets $S \subseteq V(G)$, checking for each such subset S if it induces a clique in G (i.e., all pairs of vertices in S are connected by an edge of G). If so, we know that G_S is a clique, where G_S denotes the *induced subgraph* on S defined by G ; that is, the graph formed by removing all the vertices are not in S from G (in particular, only edges that have both endpoints in S appear in G_S). Finally, our algorithm would return the largest S encountered, such that G_S is a clique. The running time of this algorithm is $O(2^n n^2)$ as can be easily verified.

Suggestion 2.1.1. When solving any algorithmic problem, always try first to find a simple (or even naive) solution. You can try optimizing it later, but even a naive solution might give you useful insight into a problem structure and behavior. TIP

We will prove that **MaxClique** is **NP-HARD**. Before dwelling into that, the simple algorithm we devised for **MaxClique** shed some light on why intuitively it should be **NP-HARD**: It does not seem like there is any way of avoiding the brute force enumeration of all possible subsets of the vertices of G . Thus, a problem is **NP-HARD** or **NP-COMPLETE**, *intuitively*, if the only way we know how to solve the problem is to use naive brute force enumeration of all relevant possibilities.

How to prove that a problem X is NP-Hard? Proving that a given problem X is NP-HARD is usually done in two steps. First, we pick a known NP-COMPLETE problem A . Next, we show how to solve any instance of A in polynomial time, assuming that we are given a polynomial time algorithm that solves X .

Proving that a problem X is NP-COMPLETE requires the additional burden of showing that is in NP. Note, that only decision problems can be NP-COMPLETE, but optimization problems can be NP-HARD; namely, the set of NP-HARD problems is much bigger than the set of NP-COMPLETE problems.

Theorem 2.1.2. *MaxClique is NP-HARD.*

Proof: We show a reduction from 3SAT. So, consider an input to 3SAT, which is a formula F defined over n variables (and with m clauses).

We build a graph from the formula F by scanning it, as follows:

- (i) For every literal in the formula we generate a vertex, and label the vertex with the literal it corresponds to.

Note, that every clause corresponds to the three such vertices.

- (ii) We connect two vertices in the graph, if they are:
 - (i) in different clauses, and
 - (ii) they are *not* a negation of each other.

Let G denote the resulting graph. See Figure 2.2 for a concrete example. Note, that this reduction can be easily be done in quadratic time in the size of the given formula.

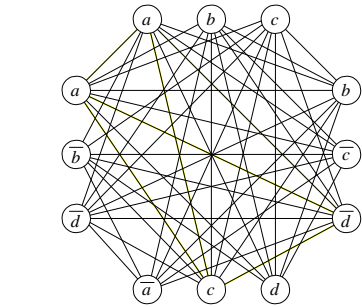


Figure 2.2: The generated graph for the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$.

We claim that F is satisfiable iff there exists a clique of size m in G .

\implies Let x_1, \dots, x_n be the variables appearing in F , and let $v_1, \dots, v_n \in \{0, 1\}$ be the satisfying assignment for F . Namely, the formula F holds if we set $x_i = v_i$, for $i = 1, \dots, n$.

For every clause C in F there must be at least one literal that evaluates to TRUE. Pick a vertex that corresponds to such TRUE value from each clause. Let W be the resulting set of vertices. Clearly, W forms a clique in G . The set W is of size m , since there are m clauses and each one contribute one vertex to the clique.

\impliedby Let U be the set of m vertices which form a clique in G .

We need to translate the clique G_U into a satisfying assignment of F .

- (i) set $x_i \leftarrow$ TRUE if there is a vertex in U labeled with x_i .
- (ii) set $x_i \leftarrow$ FALSE if there is a vertex in U labeled with \bar{x}_i .

This is a valid assignment as can be easily verified. Indeed, assume for the sake of contradiction, that there is a variable x_i such that there are two vertices u, v in U labeled with x_i and \bar{x}_i ; namely, we are trying to assign to contradictory values of x_i . But then, u and v , by construction will not be connected in G , and as such G_S is not a clique. A contradiction.

Furthermore, this is a satisfying assignment as there is at least one vertex of U in each clause. Implying, that there is a literal evaluating to TRUE in each clause. Namely, F evaluates to TRUE.

Thus, given a polytime (i.e., polynomial time) algorithm for MaxClique, we can solve 3SAT in polytime. We conclude that MaxClique is NP-HARD. ■

MaxClique is an optimization problem, but it can be easily restated as a decision problem.

Clique

Instance: A graph G , integer k

Question: Is there a clique in G of size k ?

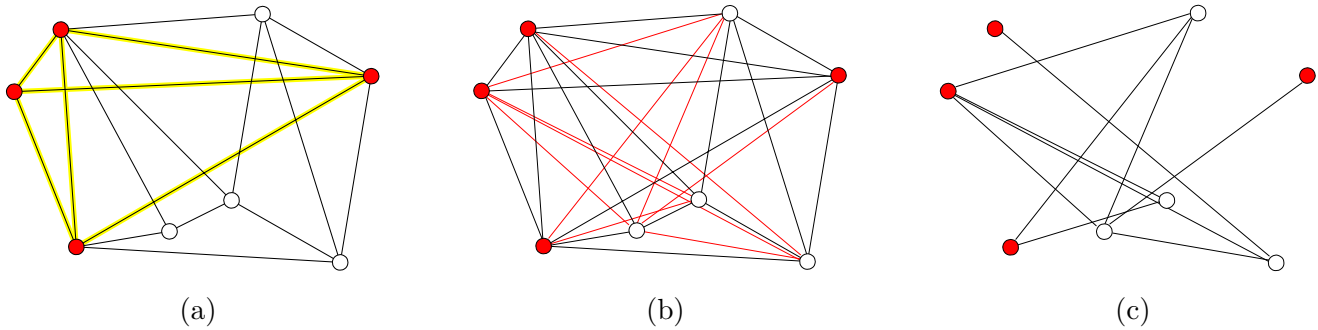


Figure 2.3: (a) A clique in a graph G , (b) the complement graph is formed by all the edges not appearing in G , and (c) the complement graph and the independent set corresponding to the clique in G .

Theorem 2.1.3. *Clique is NP-COMplete.*

Proof: It is NP-HARD by the reduction of Theorem 2.1.2. Thus, we only need to show that it is in NP. This is quite easy. Indeed, given a graph G having n vertices, a parameter k , and a set W of k vertices, verifying that every pair of vertices in W form an edge in G takes $O(u + k^2)$, where u is the size of the input (i.e., number of edges + number of vertices). Namely, verifying a positive answer to an instance of **Clique** can be done in polynomial time.

Thus, **Clique** is NP-COMplete. ■

2.2. Independent Set

Definition 2.2.1. A set S of nodes in a graph $G = (V, E)$ is an *independent set*, if no pair of vertices in S are connected by an edge.

Independent Set

Instance: A graph G , integer k

Question: Is there an independent set in G of size k ?

Theorem 2.2.2. *Independent Set is NP-COMplete.*

Proof: This readily follows by a reduction from **Clique**. Given G and k , compute the complement graph \overline{G} where we connected two vertices u, v in \overline{G} iff they are independent (i.e., not connected) in G . See Figure 2.3. Clearly, a clique in G corresponds to an independent set in \overline{G} , and vice versa. Thus, **Independent Set** is NP-HARD, and since it is in NP, it is NPC. ■

2.3. Vertex Cover

Definition 2.3.1. For a graph G , a set of vertices $S \subseteq V(G)$ is a *vertex cover* if it touches every edge of G . Namely, for every edge $uv \in E(G)$ at least one of the endpoints is in S .

Vertex Cover

Instance: A graph G , integer k

Question: Is there a vertex cover in G of size k ?

Lemma 2.3.2. A set S is a vertex cover in G iff $V \setminus S$ is an independent set in G .

Proof: If S is a vertex cover, then consider two vertices $u, v \in V \setminus S$. If $uv \in E(G)$ then the edge uv is not covered by S . A contradiction. Thus $V \setminus S$ is an independent set in G .

Similarly, if $V \setminus S$ is an independent set in G , then for any edge $uv \in E(G)$ it must be that either u or v are not in $V \setminus S$. Namely, S covers all the edges of G . ■

Theorem 2.3.3. *Vertex Cover* is NP-COMplete.

Proof: *Vertex Cover* is in NP as can be easily verified. To show that it NP-HARD we will do a reduction from *Independent Set*. So, we are given an instance of *Independent Set* which is a graph G and parameter k , and we want to know whether there is an independent set in G of size k . By Lemma 2.3.2, G has an independent set of size k iff it has a vertex cover of size $n - k$. Thus, feeding G and $n - k$ into (the supposedly given) black box that can solve vertex cover in polynomial time, we can decide if G has an independent set of size k in polynomial time. Thus *Vertex Cover* is NP-COMplete. ■

2.4. Graph Coloring

Definition 2.4.1. A *coloring*, by c colors, of a graph $G = (V, E)$ is a mapping $C : V(G) \rightarrow \{1, 2, \dots, c\}$ such that every vertex is assigned a color (i.e., an integer), such that no two vertices that share an edge are assigned the same color.

Usually, we would like to color a graph with a minimum number of colors. Deciding if a graph can be colored with two colors is equivalent to deciding if a graph is bipartite and can be done in linear time using DFS or BFS^①.

Coloring is useful for resource allocation (used in compilers for example) and scheduling type problems. Surprisingly, moving from two colors to three colors make the problem much harder.

3Colorable

Instance: A graph G .

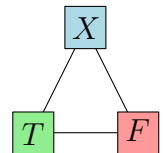
Question: Is there a coloring of G using three colors?

Theorem 2.4.2. *3Colorable* is NP-COMplete.

Proof: Clearly, *3Colorable* is in NP.

We prove that it is NP-COMplete by a reduction from *3SAT*. Let \mathcal{F} be the given *3SAT* instance. The basic idea of the proof is to use gadgets to transform the formula into a graph. Intuitively, a *gadget* is a small component that corresponds to some feature of the input.

The first gadget will be the *color generating gadget*, which is formed by three special vertices connected to each other, where the vertices are denoted by X , F and T , respectively. We will consider the color used to color T to correspond to the TRUE value, and the color of the F to correspond to the FALSE value.



^①If you do not know the algorithm for this, please read about it to fill this monstrous gap in your knowledge.

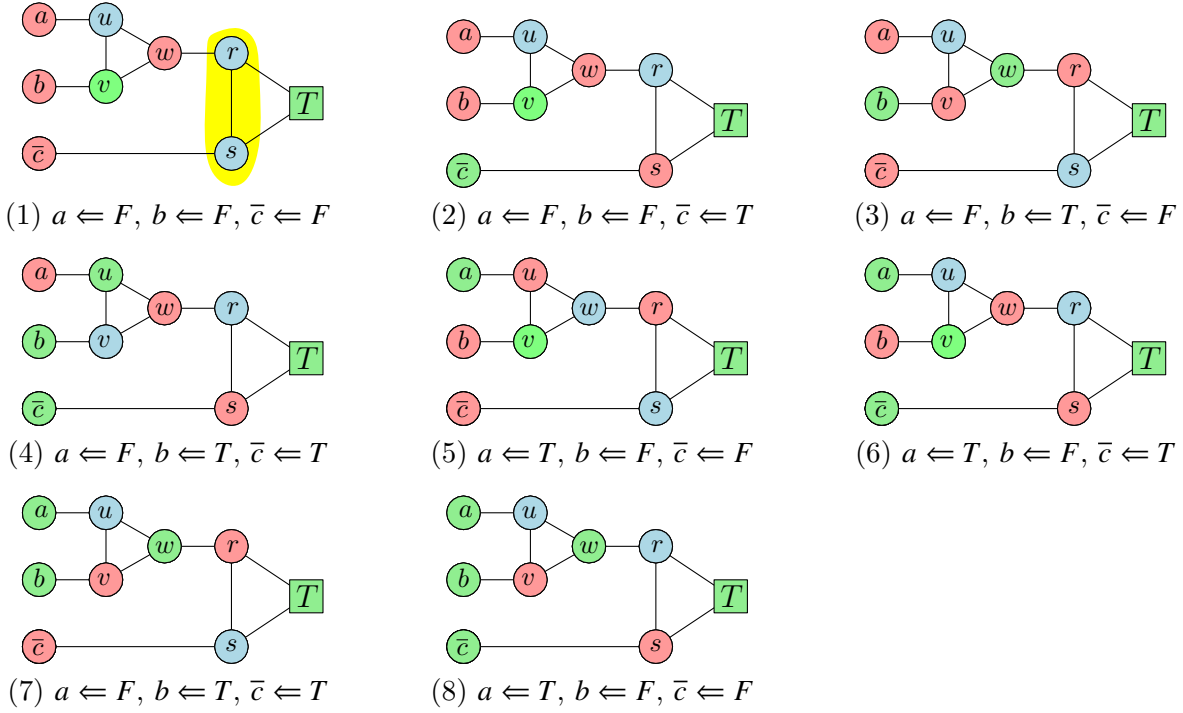
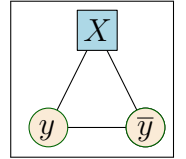
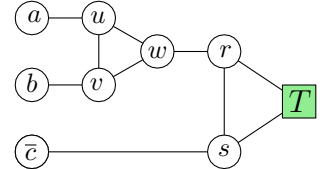


Figure 2.4: The clause $a \vee b \vee \bar{c}$ and all the three possible colorings to its literals. If all three literals are colored by the color of the special node F , then there is no valid coloring of this component, see case (1).

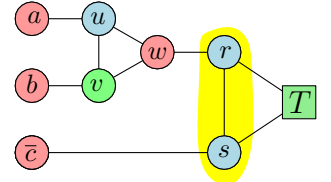
For every variable y appearing in \mathcal{F} , we will generate a *variable gadget*, which is (again) a triangle including two new vertices, denoted by y and \bar{y} , and the third vertex is the auxiliary vertex X from the color generating gadget. Note, that in a valid 3-coloring of the resulting graph either y would be colored by T (i.e., it would be assigned the same color as the color as the vertex T) and \bar{y} would be colored by F , or the other way around. Thus, a valid coloring could be interpreted as assigning **TRUE** or **FALSE** value to each variable y , by just inspecting the color used for coloring the vertex y .



Finally, for every clause we introduce a *clause gadget*. See the figure on the right – for how the gadget looks like for the clause $a \vee b \vee \bar{c}$. Note, that the vertices marked by a, b and \bar{c} are the corresponding vertices from the corresponding variable gadget. We introduce five new variables for every such gadget. The claim is that this gadget can be colored by three colors if and only if the clause is satisfied. This can be done by brute force checking all 8 possibilities, and we demonstrate it only for two cases. The reader should verify that it works also for the other cases.



Indeed, if all three vertices (i.e., three variables in a clause) on the left side of a variable clause are assigned the F color (in a valid coloring of the resulting graph), then the vertices u and v must be either be assigned X and T or T and X , respectively, in any valid 3-coloring of this gadget (see figure on the left). As such, the vertex w must be assigned the color F . But then, the vertex r must be assigned the X color. But then, the vertex s has three neighbors with all three different colors, and there is no valid coloring for s .



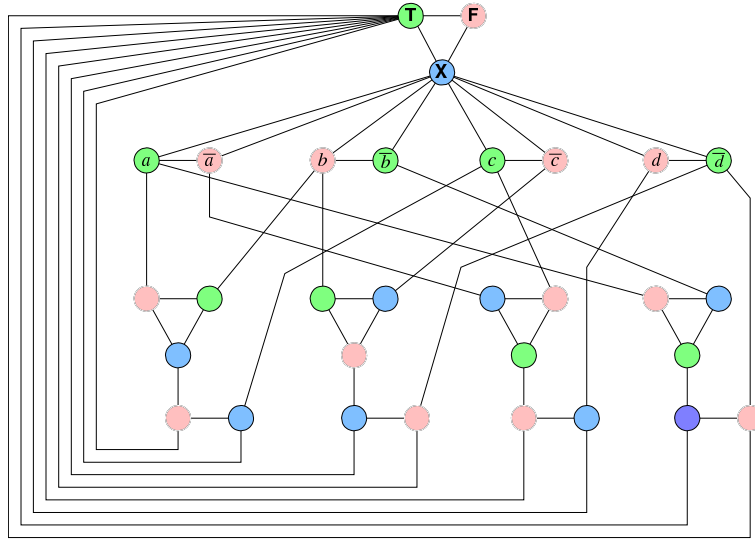
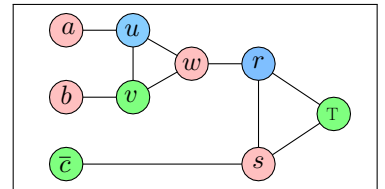


Figure 2.5: The formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ reduces to the depicted graph.

As another example, consider the case when one of the variables on the left is assigned the T color. Then the clause gadget can be colored in a valid way, as demonstrated on the figure on the right.

This concludes the reduction. Clearly, the generated graph can be computed in polynomial time. By the above argumentation, if there is a valid 3-coloring of the resulting graph G , then there is a satisfying assignment for \mathcal{F} . Similarly, if there is a satisfying assignment for \mathcal{F} then the G be colored in a valid way using three colors. For how the resulting graph looks like, see Figure 2.5.



This implies that **3Colorable** is **NP-COMLETE**. ■

Here is an interesting related problem. You are given a graph G as input, and you know that it is 3-colorable. In polynomial time, what is the minimum number of colors you can use to color this graph legally? Currently, the best polynomial time algorithm for coloring such graphs, uses $O(n^{3/14})$ colors.

Chapter 3

NP Completeness III

3.1. Hamiltonian Cycle

Definition 3.1.1. A **Hamiltonian cycle** is a cycle in the graph that visits every vertex exactly once.

Definition 3.1.2. An **Eulerian cycle** is a cycle in a graph that uses every edge exactly once.

Finding Eulerian cycle can be done in linear time. Surprisingly, finding a Hamiltonian cycle is much harder.

Hamiltonian Cycle

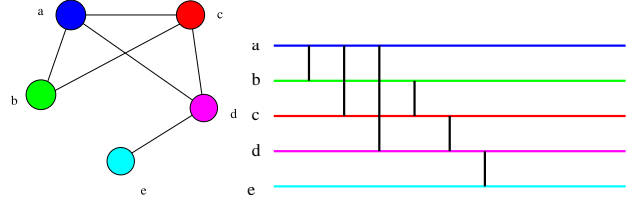
Instance: A graph G .

Question: Is there a Hamiltonian cycle in G ?

Theorem 3.1.3. *Hamiltonian Cycle is NP-COMplete.*

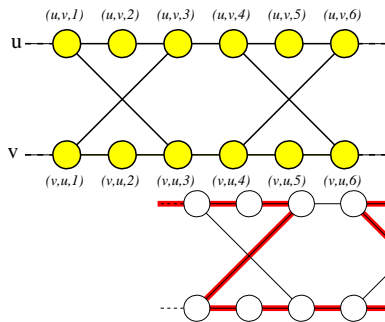
Proof: **Hamiltonian Cycle** is clearly in NP.

We will show a reduction from **Vertex Cover**. Given a graph G and integer k we redraw G in the following way: We turn every vertex into a horizontal line segment, all of the same length. Next, we turn an edge in the original graph G into a *gate*, which is a vertical segment connecting the two relevant vertices.



Note, that there is a **Vertex Cover** in G of size k if and only if there are k horizontal lines that stab all the gates in the resulting graph H (a line stabs a gate if one of the endpoints of the gate lies on the line).

Thus, computing a vertex cover in G is equivalent to computing k disjoint paths through the graph G that visits all the gates. However, there is a technical problem: a path might change venues or even go back. See figure on the right.



To overcome this problem, we will replace each gate with a component that guarantees, that if you visit all its vertices, you have to go forward and can NOT go back (or change “lanes”). The new component is depicted on the left.

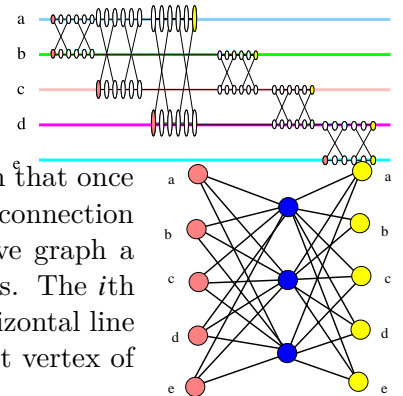
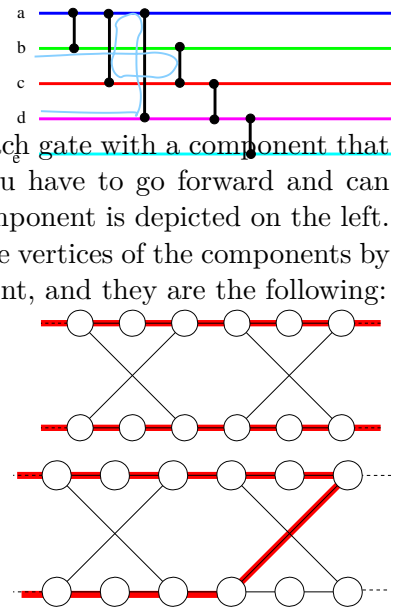
There only three possible ways to visit *all* the vertices of the components by paths that do not start/end inside the component, and they are the following:

The proof that this is the only three possibilities is by brute force. Depicted on the right is one impossible path, that tries to backtrack by entering on the top and leaving on the bottom. Observe, that there are vertices left unvisited. Which means that not all the vertices in the graph are going to be visited, because we add the constraint, that the paths start/end outside the gate-component (this condition would be enforced naturally by our final construction).

The resulting graph H_1 for the example graph we started with is depicted on the right. There exists a **Vertex Cover** in the original graph **iff** there exists k paths that start on the left side and end on the right side, in this weird graph. And these k paths visits all the vertices.

The final stroke is to add connection from the left side to the right side, such that once you arrive to the right side, you can go back to the left side. However, we want connection that allow you to travel exactly k times. This is done by adding to the above graph a “routing box” component H_2 depicted on the right, with k new middle vertices. The i th vertex on the left of the routing component is the left most vertex of the i th horizontal line in the graph, and the i th vertex on the right of the component is the right most vertex of the i th horizontal line in the graph.

It is now easy (but tedious) to verify that the resulting graph $H_1 \cup H_2$ has a Hamiltonian path **iff** H_1 has k paths going from left to right, which happens, **iff** the original graph has a **Vertex Cover** of size k . It is easy to verify that this reduction can be done in polynomial time. ■



3.2. Traveling Salesman Problem

A traveling salesman tour, is a Hamiltonian cycle in a graph, which its price is the price of all the edges it uses.

TSP

Instance: $G = (V, E)$ a complete graph - n vertices, $c(e)$: Integer cost function over the edges of G , and k an integer.

Question: Is there a traveling-salesman tour with cost at most k ?

Theorem 3.2.1. *TSP is NP-COMplete.*

Proof: Reduction from Hamiltonian cycle. Consider a graph $G = (V, E)$, and let H be the complete graph defined over V . Let

$$c(e) = \begin{cases} 1 & e \in E(G) \\ 2 & e \notin E(G). \end{cases}$$

Clearly, the cheapest TSP in H with cost function equal to n iff G is Hamiltonian. Indeed, if G is not Hamiltonian, then the TSP must use one edge that does not belong to G , and then, its price would be at least $n + 1$. ■

3.3. Subset Sum

We would like to prove that the following problem, **Subset Sum** is NPC.

Subset Sum

Instance: S - set of positive integers, t : - an integer number (Target)

Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

How does one prove that a problem is NP-COMplete? First, one has to choose an appropriate NPC to reduce from. In this case, we will use 3SAT. Namely, we are given a 3CNF formula with n variables and m clauses. The second stage, is to “play” with the problem and understand what kind of constraints can be encoded in an instance of a given problem and understand the general structure of the problem.

The first observation is that we can use very long numbers as input to **Subset Sum**. The numbers can be of polynomial length in the size of the input 3SAT formula F .

The second observation is that in fact, instead of thinking about **Subset Sum** as adding numbers, we can think about it as a problem where we are given vectors with k components each, and the sum of the vectors (coordinate by coordinate, must match. For example, the input might be the vectors $(1, 2), (3, 4), (5, 6)$ and the target vector might be $(6, 8)$. Clearly, $(1, 2) + (5, 6)$ give the required target vector. Lets refer to this new problem as **Vec Subset Sum**.

Vec Subset Sum

Instance: S - set of n vectors of dimension k , each vector has non-negative numbers for its coordinates, and a target vector \vec{t} .

Question: Is there a subset $X \subseteq S$ such that $\sum_{\vec{x} \in X} \vec{x} = \vec{t}$?

Given an instance of **Vec Subset Sum**, we can convert it into an instance of **Subset Sum** as follows: We compute the largest number in the given instance, multiply it by $n^2 \cdot k \cdot 100$, and compute how many digits are required to write this number down. Let U be this number of digits. Now, we take every vector in the given instance and we write it down using U digits, padding it with zeroes as necessary. Clearly, each vector is now converted into a huge integer number. The property is now that a sub of numbers in a specific column of the given instance can not spill into digits allocated for a different column since there are enough zeroes separating the digits corresponding to two different columns.

Next, let us observe that we can force the solution (if it exists) for **Vec Subset Sum** to include exactly one vector out of two vectors. To this end, we will introduce a new coordinate (i.e., a new column in the table on the right) for all the vectors. The two vectors a_1 and a_2 will have 1 in this coordinate, and all other vectors will have zero in this coordinate. Finally, we set this coordinate in the target vector to be 1. Clearly, a solution is a subset of vectors that in this coordinate add up to 1. Namely, we have to choose either a_1 or a_2 into our solution.

Target	??	??	01	???
a_1	??	??	01	??
a_2	??	??	01	??

In particular, for each variable x appearing in F , we will introduce two rows, denoted by x and \bar{x} and introduce the above mechanism to force choosing either x or \bar{x} to the optimal solution. If x (resp. \bar{x}) is chosen into the solution, we will interpret it as the solution to F assigns **TRUE** (resp. **FALSE**) to x .

Next, consider a clause $C \equiv a \vee b \vee \bar{c}$ appearing in F . This clause requires that we choose at least one row from the rows corresponding to a , b to \bar{c} . This can be enforced by introducing a new coordinate for the clauses C , and setting 1 for each row that if it is picked then the clause is satisfied. The question now is what do we set the target to be? Since a valid solution might have any number between 1 to 3 as a sum of this coordinate. To overcome this, we introduce three new dummy rows, that store in this coordinate, the numbers 7, 8 and 9, and we set this coordinate in the target to be 10. Clearly, if we pick to dummy rows into the optimal solution then sum in this coordinate would exceed 10. Similarly, if we do not pick one of these three dummy rows to the optimal solution, the maximum sum in this coordinate would be $1+1+1 = 3$, which is smaller than 10. Thus, the only possibility is to pick one dummy row, and some subset of the rows such that the sum is 10. Notice, this “gadget” can accommodate any (non-empty) subset of the three rows chosen for a , b and \bar{c} .

numbers	...	$C \equiv a \vee b \vee \bar{c}$...
a	...	01	...
\bar{a}	...	00	...
b	...	01	...
\bar{b}	...	00	...
c	...	00	...
\bar{c}	...	01	...
C fix-up 1	000	07	000
C fix-up 2	000	08	000
C fix-up 3	000	09	000
TARGET		10	

We repeat this process for each clause of F . We end up with a set U of $2n+3m$ vectors with $n+m$ coordinate, and the question if there is a subset of these vectors that add up to the target vector. There is such a subset if and only if the original formula F is satisfiable, as can be easily verified. Furthermore, this reduction can be done in polynomial time.

Finally, we convert these vectors into an instance of **Subset Sum**. Clearly, this instance of **Subset Sum** has a solution if and only if the original instance of **3SAT** had a solution. Since **Subset Sum** is in **NP** as an be easily verified, we conclude that that **Subset Sum** is **NP-COMplete**.

Theorem 3.3.1. *Subset Sum is NP-COMplete.*

For a concrete example of the reduction, see **Figure 3.1**.

3.4. 3 dimensional Matching (3DM)

3DM

Instance: X, Y, Z sets of n elements, and T a set of triples, such that $(a, b, c) \in T \subseteq X \times Y \times Z$.

Question: Is there a subset $S \subseteq T$ of n disjoint triples, s.t. every element of $X \cup Y \cup Z$ is covered exactly once.?

Theorem 3.4.1. *3DM is NP-COMplete.*

The proof is long and tedious and is omitted.

BTW, **2DM** is polynomial (later in the course?).

numbers	$a \vee \bar{a}$	$b \vee \bar{b}$	$c \vee \bar{c}$	$d \vee \bar{d}$	$D \equiv \bar{b} \vee c \vee \bar{d}$	$C \equiv a \vee b \vee \bar{c}$
a	1	0	0	0	00	01
\bar{a}	1	0	0	0	00	00
b	0	1	0	0	00	01
\bar{b}	0	1	0	0	01	00
c	0	0	1	0	01	00
\bar{c}	0	0	1	0	00	01
d	0	0	0	1	00	00
\bar{d}	0	0	0	1	01	01
C fix-up 1	0	0	0	0	00	07
C fix-up 2	0	0	0	0	00	08
C fix-up 3	0	0	0	0	00	09
D fix-up 1	0	0	0	0	07	00
D fix-up 2	0	0	0	0	08	00
D fix-up 3	0	0	0	0	09	00
TARGET	1	1	1	1	10	10

numbers
010000000001
010000000000
000100000001
000100000100
000001000100
000001000001
000000010000
000000010101
000000000007
000000000008
000000000009
000000000700
000000000800
000000000900
010101011010

Figure 3.1: The **Vec Subset Sum** instance generated for the 3SAT formula $F = (\bar{b} \vee c \vee \bar{d}) \wedge (a \vee b \vee \bar{c})$ is shown on the left. On the right side is the resulting instance of **Subset Sum**.

3.5. Partition

Partition

Instance: A set S of n numbers.
Question: Is there a subset $T \subseteq S$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$?

Theorem 3.5.1. *Partition is NP-COMLETE.*

Proof: **Partition** is in NP, as we can easily verify that such a partition is valid.

Reduction from **Subset Sum**. Let the given instance be n numbers a_1, \dots, a_n and a target number t . Let $S = \sum_{i=1}^n a_i$, and set $a_{n+1} = 3S - t$ and $a_{n+2} = 3S - (S - t) = 2S + t$. It is easy to verify that there is a solution to the given instance of subset sum, iff there is a solution to the following instance of partition:

$$a_1, \dots, a_n, a_{n+1}, a_{n+2}.$$

Clearly, Partition is in NP and thus it is NP-COMLETE. ■

3.6. Some other problems

It is not hard to show that the following problems are NP-COMLETE:

SET COVER

Instance: (S, \mathcal{F}, k) :
 S : A set of n elements
 \mathcal{F} : A family of subsets of S , s.t. $\bigcup_{X \in \mathcal{F}} X = S$.
 k : A positive integer.
Question: Are there k sets $S_1, \dots, S_k \in \mathcal{F}$ that cover S . Formally, $\bigcup_i S_i = S$?

Part II

Dynamic programming

Chapter 4

Dynamic programming

The events of 8 September prompted Foch to draft the later legendary signal: “My centre is giving way, my right is in retreat, situation excellent. I attack.” It was probably never sent.

– – The first world war, John Keegan..

4.1. Basic Idea - Partition Number

Definition 4.1.1. For a positive integer n , the *partition number* of n , denoted by $p(n)$, is the number of different ways to represent n as a decreasing sum of positive integers.

The different number of partitions of 6 are shown on the right.

It is natural to ask how to compute $p(n)$. The “trick” is to think about a recursive solution and observe that once we decide what is the leading number d , we can solve the problem recursively on the remaining budget $n-d$ under the constraint that no number exceeds d .

$6 = 6$		
$6=5+1$		
$6=4+2$	$6=4+1+1$	
$6 = 3 + 3$	$6 = 3 + 2 + 1$	$6+3+1+1+1$
$6=2+2+2$	$6=2+2+1+1$	$6=2+1+1+1+1$
$6=1+1+1+1+1+1$		

Suggestion 4.1.2. Recursive algorithms are one of the main tools in developing algorithms (and writing programs). If you do not feel comfortable with recursive algorithms you should spend time playing with recursive algorithms till you feel comfortable using them. Without the ability to think recursively, this class would be a long and painful torture to you. Speak with me if you need guidance on this topic.

TIP

The resulting algorithm is depicted on the right. We are interested in analyzing its running time. To this end, draw the recursion tree of **PARTITIONS** and observe that the amount of work spend at each node, is proportional to the number of children it has. Thus, the overall time spend by the algorithm is proportional to the size of the recurrence tree, which is proportional (since every node is either a leaf or has at least two children) to the number of leaves in the tree, which is $\Theta(p(n))$.

This is not very exciting, since it is easy verify that $3^{\sqrt{n}/4} \leq p(n) \leq n^n$.

Exercise 4.1.3. Prove the above bounds on $p(n)$ (or better bounds).

```

PartitionsI(num,d) //d-max digit
  if (num ≤ 1) or (d = 1)
    return 1
  if d > num
    d ← num
  res ← 0
  for i ← d down to 1
    res = res + PartitionsI(num - i,i)
  return res

Partitions(n)
  return PartitionsI(n,n)

```

Suggestion 4.1.4. Exercises in the class notes are a natural easy questions for inclusions in exams. You probably want to spend time doing them.

TIP

Hardy and Ramanujan (in 1918) showed that $p(n) \approx \frac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}}$ (which I am sure was your first guess).

It is natural to ask, if there is a faster algorithm. Or more specifically, why is the algorithm **Partitions** so slowwwwwwwwwwwwwwwwwwwwwww? The answer is that during the computation of **Partitions(n)** the function **PartitionsI(num,max_digit)** is called a lot of times with the *same* parameters.

An easy way to overcome this problem is cache the results of **PartitionsI** using a hash table.^① Whenever **PartitionsI** is being called, it checks in a cache table if it already computed the value of the function for this parameters, and if so it returns the result. Otherwise, it computes the value of the function and before returning the value, it stores it in the cache. This simple (but powerful) idea is known as **memoization**.

What is the running time of **PartitionS_C**? Analyzing recursive algorithm that have been transformed by memoization are usually analyzed as follows: (i) bound the number of values stored in the hash table, and (ii) bound the amount of work involved in storing one value into the hash table (ignoring recursive calls).

Here is the argument in this case:

- (A) If a call to **PartitionsI_C** takes (by itself) more than constant time, then this call performs a store in the cache.
- (B) Number of store operations in the cache is $O(n^2)$, since this is the number of different entries stored in the cache. Indeed, for **PartitionsI_C(num,max_digit)**, the parameters num and max_digit are both integers in the range $1, \dots, n$.
- (C) We charge the work in the loop to the resulting store. The work in the loop is at most $O(n)$ time (since $max_digit \leq n$).
- (D) As such, the overall running time of **PartitionS_C(n)** is $O(n^2) \times O(n) = O(n^3)$.

Note, that this analysis is naive but it would be sufficient for our purposes (verify that the bound of $O(n^3)$ on the running time is tight in this case).

```

PartitionsI_C(num,max_digit)
  if (num ≤ 1) or (max_digit = 1)
    return 1
  if max_digit > num
    d ← num
  if ⟨num,max_digit⟩ in cache
    return cache(⟨num,max_digit⟩)
  res ← 0
  for i ← max_digit down to 1 do
    res += PartitionsI_C(num - i,i)
  cache(⟨num,max_digit⟩) ← res
  return res

PartitionS_C(n)
  return PartitionsI_C(n,n)

```

^①Throughout the course, we will assume that a hash table operation can be done in constant time. This is a reasonable assumption using randomization and perfect hashing.

4.1.1. A Short sermon on memoization

This idea of memoization is generic and nevertheless very useful. To recap, it works by taking a recursive function and caching the results as the computations goes on. Before trying to compute a value, check if it was already computed and if it is already stored in the cache. If so, return result from the cache. If it is not in the cache, compute it and store it in the cache (for the time being, you can think about the cache as being a hash table).

- **When does it work:** There is a lot of inefficiency in the computation of the recursive function because the same call is being performed repeatedly.
- **When it does NOT work:**
 - (A) The number of different recursive function calls (i.e., the different values of the parameters in the recursive call) is “large”.
 - (B) When the function has side effects.

Tidbit 4.1.5. Some functional programming languages allow one to take a recursive function $f(\cdot)$ that you already implemented and give you a memorized version $f'(\cdot)$ of this function without the programmer doing any extra work. For a nice description of how to implement it in Scheme see [ASS96].

tidbit

It is natural to ask if we can do better than just using caching? As usual in life – more pain, more gain. Indeed, in a lot of cases we can analyze the recursive calls, and store them directly in an (sometime multi-dimensional) array. This gets rid of the recursion (which used to be an important thing long time ago when memory, used by the stack, was a truly limited resource, but it is less important nowadays) which usually yields a slight improvement in performance in the real world.

This technique is known as *dynamic programming*^②. We can sometime save space and improve running time in dynamic programming over memoization.

Dynamic programming made easy:

- (A) Solve the problem using recursion - easy (?).
- (B) Modify the recursive program so that it caches the results.
- (C) Dynamic programming: Modify the cache into an array.

4.2. Example – Fibonacci numbers

Let us revisit the classical problem of computing Fibonacci numbers.

4.2.1. Why, where, and when?

To remind the reader, in the Fibonacci sequence, the first two numbers $F_0 = 0$ and $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$, for $i > 1$. This sequence was discovered independently in several places and times. From Wikipedia:

“The Fibonacci sequence appears in Indian mathematics, in connection with Sanskrit prosody. In the Sanskrit oral tradition, there was much emphasis on how long (L) syllables mix with the short (S), and counting the different patterns of L and S within a given fixed length results in the Fibonacci numbers; the number of patterns that are m short syllables long is the Fibonacci number F_{m+1} .”

(To see that, imagine that a long syllable is equivalent in length to two short syllables.) Surprisingly, the credit for this formalization goes back more than 2000 years (!)

^②As usual in life, it is not dynamic, it is not programming, and its hardly a technique. To overcome this, most texts find creative ways to present this topic in the most opaque way possible.


```

FibDP( $n$ )
  if  $n \leq 1$ 
    return 1
  if  $F[n]$  initialized
    return  $F[n]$ 
   $F[n] \leftarrow \text{FibDP}(n-1) + \text{FibDP}(n-2)$ 
  return  $F[n]$ 

```

Figure 4.1

Fibonacci was a decent mathematician (1170—1250 AD), and his most significant and lasting contribution was spreading the Hindu-Arabic numerical system (i.e., zero) in Europe. He was the son of a rich merchant that spend much time growing up in Algiers, where he learned the decimal notation system. He traveled throughout the Mediterranean world to study mathematics. When he came back to Italy he published a sequence of books (the first one “Liber Abaci” contained the description of the decimal notations system). In this book, he also posed the following problem:

Consider a rabbit population, assuming that: A newly born pair of rabbits, one male, one female, are put in a field; rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits; rabbits never die and a mating pair always produces one new pair (one male, one female) every month from the second month on. The puzzle that Fibonacci posed was: how many pairs will there be in one year?

(The above is largely based on Wikipedia.)

4.2.2. Computing Fibonacci numbers

The recursive function for computing Fibonacci numbers is depicted on the right. As before, the running time of **FibR**(n) is proportional to $O(F_n)$, where F_n is the n th Fibonacci number. It is known that

$$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right] = \Theta(\phi^n),$$

where $\phi = \frac{1 + \sqrt{5}}{2}$.

We can now use memoization, and with a bit of care, it is easy enough to come up with the dynamic programming version of this procedure, see **FibDP** in Figure 4.1. Clearly, the running time of **FibDP**(n) is linear (i.e., $O(n)$).

A careful inspection of **FibDP** exposes the fact that it fills the array $F[\dots]$ from left to right. In particular, it only requires the last two numbers in the array.

As such, we can get rid of the array all together, and reduce space needed to $O(1)$: This is a phenomena that is quite common in dynamic programming: By carefully inspecting the way the array/table is being filled, sometime one can save space by being careful about the implementation.

The running time of **FibI** is identical to the running time of **FibDP**. Can we do better?

Surprisingly, the answer is yes, to this end observe that

$$\begin{pmatrix} y \\ x + y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

```

FibR( $n$ )
  if  $n = 0$ 
    return 1
  if  $n = 1$ 
    return 1
  return FibR( $n-1$ ) + FibR( $n-2$ )

```

```

FibI( $n$ )
   $prev \leftarrow 0, curr \leftarrow 1$ 
  for  $i = 1$  to  $n$  do
     $next \leftarrow curr + prev$ 
     $prev \leftarrow curr$ 
     $curr \leftarrow next$ 
  return  $curr$ 

```

	h	a	r	-	p		e	l	e	d
s	h	a	r		p	<space>	e	y	e	d
1	0	0	0	1	0	1	0	1	0	0

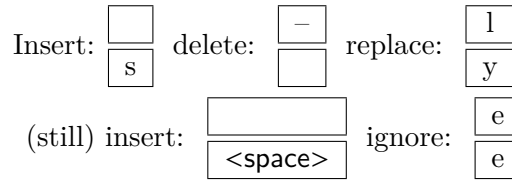


Figure 4.2: Interpreting edit-distance as a alignment task. Aligning identical characters to each other is free of cost. The price in the above example is 4. There are other ways to get the same edit-distance in this case.

As such,

$$\begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \begin{pmatrix} F_{n-3} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}.$$

Thus, computing the n th Fibonacci number can be done by computing

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3}.$$

How to this quickly? Well, we know that $a*b*c = (a*b)*c = a*(b*c)$ ^③, as such one can compute a^n by repeated squaring, see pseudo-code on the right. The running time of **FastExp** is $O(\log n)$ as can be easily verified. Thus, we can compute in F_n in $O(\log n)$ time.

But, something is very strange. Observe that F_n has $\approx \log_{10} 1.68\dots^n = \Theta(n)$ digits. How can we compute a number that is that large in logarithmic time? Well, we assumed that the time to handle a number is $O(1)$ independent of its size. This is not true in practice if the numbers are large. Naturally, one has to be very careful with such assumptions.

```

FastExp(a, n)
  if n = 0 then
    return 1
  if n = 1 then
    return a
  if n is even then
    return (FastExp(a, n/2))^2
  else
    return a * (FastExp(a, (n-1)/2))^2

```

4.3. Edit Distance

We are given two strings A and B , and we want to know how close the two strings are too each other. Namely, how many edit operations one has to make to turn the string A into B ?

We allow the following operations: (i) insert a character, (ii) delete a character, and (iii) replace a character by a different character. Price of each operation is one unit.

For example, consider the strings A =“har-peled” and B =“sharp eyed”. Their **edit distance** is 4, as can be easily seen.

^③Associativity of multiplication...

But how do we compute the edit-distance (min # of edit operations needed)?

The idea is to list the edit operations from left to right. Then edit distance turns into an alignment problem. See [Figure 4.2](#).

In particular, the idea of the recursive algorithm is to inspect the last character and decide which of the categories it falls into: insert, delete or ignore. See pseudo-code on the right.

The running time of `ed(...)`? Clearly exponential, and roughly 2^{n+m} , where $n + m$ is the size of the input.

So how many **different** recursive calls `ed` performs? Only: $O(m * n)$ different calls, since the only parameters that matter are n and m .

So the natural thing is to introduce memoization. The resulting algorithm `edM` is depicted on the right. The running time of `edM(n,m)` when executed on two strings of length n and m respectively is $O(nm)$, since there are $O(nm)$ store operations in the cache, and each store requires $O(1)$ time (by charging one for each recursive call). Looking on the entry $T[i, j]$ in the table, we realize that it depends only on $T[i - 1, j]$, $T[i, j - 1]$ and $T[i - 1, j - 1]$. Thus, instead of recursive algorithm, we can fill the table T row by row, from left to right.

```

ed(A[1..m], B[1..n])
  if m = 0 return n
  if n = 0 return m
  pinsert = ed(A[1..m], B[1..(n - 1)]) + 1
  pdelete = ed(A[1..(m - 1)], B[1..n]) + 1
  pr/i = ed(A[1..(m - 1)], B[1..(n - 1)])
          + [A[m] ≠ B[n]]
  return min(pinsert, pdelete, preplace/ignore)

```

```

edM(A[1..m], B[1..n])
  if m = 0 return n
  if n = 0 return m
  if T[m, n] is initialized then return T[m, n]
  pinsert = edM(A[1..m], B[1..(n - 1)]) + 1
  pdelete = edM(A[1..(m - 1)], B[1..n]) + 1
  pr/i = edM(A[1..(m - 1)], B[1..(n - 1)]) + [A[m] ≠ B[n]]
  T[m, n] ← min(pinsert, pdelete, preplace/ignore)
  return T[m, n]

```

```

edDP(A[1..m], B[1..n])
  for i = 1 to m do T[i, 0] ← i
  for j = 1 to n do T[0, j] ← j
  for i ← 1 to m do
    for j ← 1 to n do
      pinsert = T[i, j - 1] + 1
      pdelete = T[i - 1, j] + 1
      pr/ignore = T[i - 1, j - 1] + [A[i] ≠ B[j]]
      T[i, j] ← min(pinsert, pdelete, pr/ignore)
  return T[m, n]

```

The dynamic programming version that uses a two dimensional array is pretty simple now to derive and is depicted on the left. Clearly, it requires $O(nm)$ time, and $O(nm)$ space. See the pseudo-code of the resulting algorithm `edDP` on the left.

It is enlightening to think about the algorithm as computing for each $T[i, j]$ the cell it got the value from. What you get is a tree encoded in the table. See [Figure 4.3](#). It is now easy to extract from the table the

sequence of edit operations that realizes the minimum edit distance between A and B . Indeed, we start a walk on this graph from the node corresponding to $T[n, m]$. Every time we walk left, it corresponds to a deletion, every time we go up, it corresponds to an insertion, and going sideways corresponds to either replace/ignore.

Note, that when computing the i th row of $T[i, j]$, we only need to know the value of the cell to the left of the current cell, and two cells in the row above the current cell. It is thus easy to verify that the algorithm needs only to remember the current and previous row to compute the edit distance. We conclude:

Theorem 4.3.1. *Given two strings A and B of length n and m , respectively, one can compute their edit distance in $O(nm)$. This uses $O(nm)$ space if we want to extract the sequence of edit operations, and $O(n + m)$ space if we only want to output the price of the edit distance.*

Exercise 4.3.2. *Show how to compute the sequence of edit-distance operations realizing the edit distance using only $O(n + m)$ space and $O(nm)$ running time. (Hint: Use a recursive algorithm, and argue that the recursive call is always on a matrix which is of size, roughly, half of the input matrix.)*

		A	L	G	O	R	I	T	H	M
	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8	← 9
A	↑ 1	↖	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7
L	↑ 2	↑	↖	0	← 1	← 2	← 3	← 4	← 5	← 6
T	↑ 3	↑	↑	↖	1	← 2	← 3	← 4	↙	4
R	↑ 4	↑	↑	↑	↖	↖	2	← 3	← 4	← 5
U	↑ 5	↑	↑	↖	↖	↖	↖	3	← 4	← 5
I	↑ 6	↑	↑	↖	↖	↖	↖	↖	3	← 4
S	↑ 7	↑	↑	↑	↑	↑	↑	↖	4	← 5
T	↑ 8	↑	↑	↑	↑	↑	↑	↖	↖	4
I	↑ 9	↑	↑	↑	↑	↑	↑	↖	↖	↖
C	↑ 10	↑	↑	↑	↑	↑	↑	↑	↖	↖

Figure 4.3: Extracting the edit operations from the table.

4.3.1. Shortest path in a DAG and dynamic programming

Given a dynamic programming problem and its associated recursive program, one can consider all the different possible recursive calls, as *configurations*. We can create graph, every configuration is a node, and an edge is introduced between two configurations if one configuration is computed from another configuration, and we put the additional price that might be involved in moving between the two configurations on the edge connecting them. As such, for the edit distance, we have directed edges from the vertex (i, j) to $(i, j - 1)$ and $(i - 1, j)$ both with weight 1 on them. Also, we have an edge between (i, j) to $(i - 1, j - 1)$ which is of weight 0 if $A[i] = B[j]$ and 1 otherwise. Clearly, in the resulting graph, we are asking for the shortest path between (n, m) and $(0, 0)$.

And here are where things gets interesting. The resulting graph G is a DAG (*directed acyclic graph*^④). DAG can be interpreted as a partial ordering of the vertices, and by topological sort on the graph (which takes linear time), one can get a full ordering of the vertices which agrees with the DAG. Using this ordering, one can compute the shortest path in a DAG in linear time (in the size of the DAG). For edit-distance the DAG size is $O(nm)$, and as such this algorithm takes $O(nm)$ time.

This interpretation of dynamic programming as a shortest path problem in a DAG is a useful way of thinking about it, and works for many dynamic programming problems.

More surprisingly, one can also compute the longest path in a DAG in linear time. Even for negative weighted edges. This is also sometime a problem that solving it is equivalent to dynamic programming.

^④No cycles in the graph – its a miracle!

Chapter 5

Dynamic programming II - The Recursion Strikes Back

“No, mademoiselle, I don’t capture elephants. I content myself with living among them. I like them. I like looking at them, listening to them, watching them on the horizon. To tell you the truth, I’d give anything to become an elephant myself. That’ll convince you that I’ve nothing against the Germans in particular: they’re just men to me, and that’s enough.”

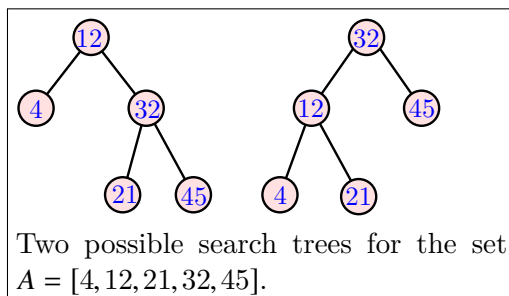
– – The roots of heaven, Romain Gary.

5.1. Optimal search trees

Given a binary search tree \mathcal{T} , the time to search for an element x , that is stored in \mathcal{T} , is $O(1 + \text{depth}(\mathcal{T}, x))$, where $\text{depth}(\mathcal{T}, x)$ denotes the depth of x in \mathcal{T} (i.e., this is the length of the path connecting x with the root of \mathcal{T}).

Problem 5.1.1. Given a set of n (sorted) keys $A[1 \dots n]$, build the best binary search tree for the elements of A .

Note, that we store the values in the internal node of the binary trees. The figure on the right shows two possible search trees for the same set of numbers. Clearly, if we are accessing the number 12 all the time, the tree on the left would be better to use than the tree on the right.



Usually, we just build a balanced binary tree, and this is good enough. But assume that we have additional information about what is the frequency in which we access the element $A[i]$, for $i = 1, \dots, n$. Namely, we know that $A[i]$ is going to be accessed $f[i]$ times, for $i = 1, \dots, n$.

In this case, we know that the total search time for a tree \mathcal{T} is $S(\mathcal{T}) = \sum_{i=1}^n (\text{depth}(\mathcal{T}, i) + 1)f[i]$, where $\text{depth}(\mathcal{T}, i)$ is the depth of the node in \mathcal{T} storing the value $A[i]$. Assume that $A[r]$ is the value stored in the root of the tree \mathcal{T} . Clearly, all the values smaller than $A[r]$ are in the subtree $\text{left}_{\mathcal{T}}$, and all values larger than r are in $\text{right}_{\mathcal{T}}$. Thus, the total search time, in this case, is

$$S(\mathcal{T}) = \sum_{i=1}^{r-1} (\text{depth}(\text{left}_{\mathcal{T}}, i) + 1)f[i] + \overbrace{\sum_{i=1}^n f[i]}^{\text{price of access to root}} + \sum_{i=r+1}^n (\text{depth}(\text{right}_{\mathcal{T}}, i) + 1)f[i].$$

Observe, that if \mathcal{T} is the optimal search tree for the access frequencies $f[1], \dots, f[n]$, then the subtree $\text{left}_{\mathcal{T}}$ must be *optimal* for the elements accessing it (i.e., $A[1 \dots r - 1]$ where r is the root).

Thus, the price of \mathcal{T} is

$$S(\mathcal{T}) = S(\text{left}_{\mathcal{T}}) + S(\text{right}_{\mathcal{T}}) + \sum_{i=1}^n f[i],$$

where $S(Q)$ is the price of searching in Q for the frequency of elements stored in Q .

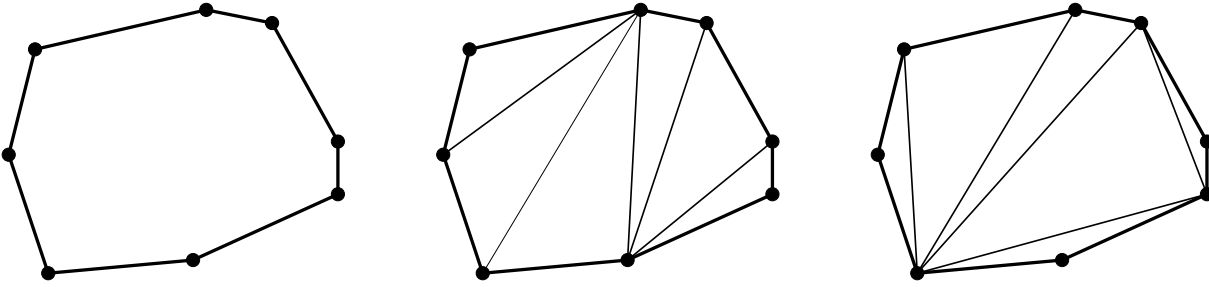


Figure 5.1: A polygon and two possible triangulations of the polygon.

This recursive formula naturally gives rise to a recursive algorithm, which is depicted on the right. The naive implementation requires $O(n^2)$ time (ignoring the recursive call). But in fact, by a more careful implementation, together with the tree \mathcal{T} , we can also return the price of searching on this tree with the given frequencies. Thus, this modified algorithm. Thus, the running time for this function takes $O(n)$ time (ignoring recursive calls). The running time of the resulting algorithm is

```

CompBestTreeI ( $A[i \dots j]$ ,  $f[i \dots j]$  )
  for  $r = i \dots j$  do
     $T_{left} \leftarrow \mathbf{CompBestTreeI}(A[i \dots r - 1], f[i \dots r - 1])$ 
     $T_{right} \leftarrow \mathbf{CompBestTreeI}(A[r + 1 \dots j], f[r + 1 \dots j])$ 
     $T_r \leftarrow \mathbf{Tree}(T_{left}, A[r], T_{right})$ 
     $P_r \leftarrow S(T_r)$ 

  return cheapest tree out of  $T_i, \dots, T_j$ .

```

```

CompBestTree ( $A[1 \dots n]$ ,  $f[1 \dots n]$  )
  return CompBestTreeI(  $A[1 \dots n]$ ,  $f[1 \dots n]$  )

```

$$\alpha(n) = O(n) + \sum_{i=0}^{n-1} (\alpha(i) + \alpha(n - i - 1)),$$

and the solution of this recurrence is $O(n3^n)$.

We can, of course, improve the running time using memoization. There are only $O(n^2)$ different recursive calls, and as such, the running time of **CompBestTreeMemoize** is $O(n^2) \cdot O(n) = O(n^3)$.

Theorem 5.1.2. *One can compute the optimal binary search tree in $O(n^3)$ time using $O(n^2)$ space.*

A further improvement arises from the fact that the root location is “monotone”. Formally, if $R[i, j]$ denotes the location of the element stored in the root for the elements $A[i \dots j]$ then it holds that $R[i, j - 1] \leq R[i, j] \leq R[i, j + 1]$. This limits the search space, and we can be more efficient in the search. This leads to $O(n^2)$ algorithm. Details are in Jeff Erickson class notes.

5.2. Optimal Triangulations

Given a convex polygon P in the plane, we would like to find the triangulation of P of minimum total length. Namely, the total length of the diagonals of the triangulation of P , plus the (length of the) perimeter of P are minimized. See **Figure 5.1**.

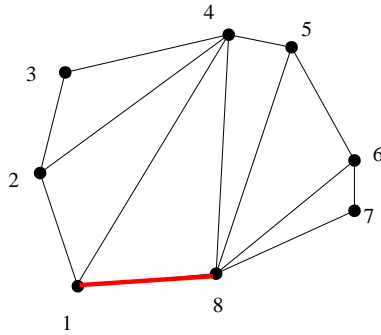
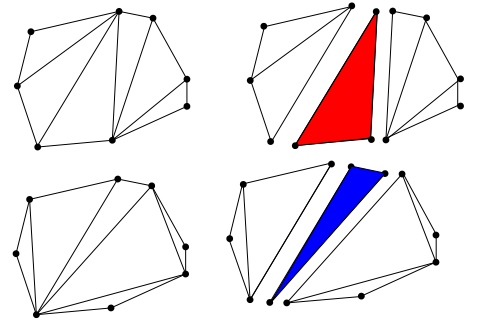
Definition 5.2.1. A set $S \subseteq \mathbb{R}^d$ is **convex** if for any to $x, y \in S$, the segment xy is contained in S .

A **convex polygon** is a closed cycle of segments, with no vertex pointing inward. Formally, it is a simple closed polygonal curve which encloses a convex set.

A **diagonal** is a line segment connecting two vertices of a polygon which are not adjacent. A **triangulation** is a partition of a convex polygon into (interior) disjoint triangles using diagonals.

Observation 5.2.2. *Any triangulation of a convex polygon with n vertices is made out of exactly $n - 2$ triangles.*

Our purpose is to find the triangulation of P that has the minimum total length. Namely, the total length of diagonals used in the triangulation is minimized. We would like to compute the optimal triangulation using divide and conquer. As the figure on the right demonstrate, there is always a triangle in the triangulation, that breaks the polygon into two polygons. Thus, we can try and guess such a triangle in the optimal triangulation, and recurse on the two polygons such created. The only difficulty, is to do this in such a way that the recursive subproblems can be described in succinct way.



To this end, we assume that the polygon is specified as list of vertices $1 \dots n$ in a clockwise ordering. Namely, the input is a list of the vertices of the polygon, for every vertex, the two coordinates are specified. The key observation, is that in any triangulation of P , there exist a triangle that uses the edge between vertex 1 and n (red edge in figure on the left).

In particular, removing the triangle using the edge $1 - n$ leaves us with two polygons which their vertices are *consecutive* along the original polygon.

Let $M[i, j]$ denote the price of triangulating a polygon starting at vertex i and ending at vertex j , where every diagonal used contributes its length twice to this quantity, and the perimeter edges contribute their length exactly once. We have the following “natural” recurrence:

$$M[i, j] = \begin{cases} 0 & j \leq i \\ 0 & j = i + 1 \\ \min_{i < k < j} (\Delta(i, j, k) + M[i, k] + M[k, j]) & \text{Otherwise} \end{cases}$$

Where $Dist(i, j) = \sqrt{(x[i] - x[j])^2 + (y[i] - y[j])^2}$ and $\Delta(i, j, k) = Dist(i, j) + Dist(j, k) + Dist(i, k)$, where the i th point has coordinates $(x[i], y[i])$, for $i = 1, \dots, n$. Note, that the quantity we are interested in is $M[1, n]$, since it the triangulation of P with minimum total weight.

Using dynamic programming (or just memoization), we get an algorithm that computes optimal triangulation in $O(n^3)$ time using $O(n^2)$ space.

5.3. Matrix Multiplication

We are given two matrix: (i) A is a matrix with dimensions $p \times q$ (i.e., p rows and q columns) and (ii) B is a matrix of size $q \times r$. The product matrix AB , with dimensions $p \times r$, can be computed in $O(pqr)$ time using the standard algorithm.

A	1000×2
B	2×1000
C	1000×2

Things becomes considerably more interesting when we have to multiply a chain for matrices. Consider for example the three matrices A, B and C with dimensions as listed on the left. Computing the matrix $ABC = A(BC)$ requires $2 \cdot 1000 \cdot 2 + 1000 \cdot 2 \cdot 2 = 8,000$ operations. On the other hand, computing the same matrix using $(AB)C$ requires $1000 \cdot 2 \cdot 1000 + 1000 \cdot 1000 \cdot 2 = 4,000,000$. Note, that matrix multiplication is associative, and as such $(AB)C = A(BC)$.

Thus, given a chain of matrices that we need to multiply, the exact ordering in which we do the multiplication matters as far to multiply the order is important as far as efficiency.

Problem 5.3.1. The input is n matrices M_1, \dots, M_n such that M_i is of size $D[i - 1] \times D[i]$ (i.e., M_i has $D[i - 1]$ rows and $D[i]$ columns), where $D[0 \dots n]$ is array specifying the sizes. Find the ordering of multiplications to compute $M_1 \cdot M_2 \cdot \dots \cdot M_{n-1} \cdot M_n$ most efficiently.

Again, let us define a recurrence for this problem, where $M[i, j]$ is the amount of work involved in computing the product of the matrices $M_i \cdots M_j$. We have

$$M[i, j] = \begin{cases} 0 & j = i \\ D[i - 1] \cdot D[i] \cdot D[i + 1] & j = i + 1 \\ \min_{i \leq k < j} (M[i, k] + M[k + 1, j] + D[i - 1] \cdot D[k] \cdot D[j]) & j > i + 1. \end{cases}$$

Again, using memoization (or dynamic programming), one can compute $M[1, n]$, in $O(n^3)$ time, using $O(n^2)$ space.

5.4. Longest Ascending Subsequence

Given an array of numbers $A[1 \dots n]$ we are interested in finding the *longest ascending subsequence*. For example, if $A = [6, 3, 2, 5, 1, 12]$ the longest ascending subsequence is 2, 5, 12. To this end, let $M[i]$ denote longest increasing subsequence having $A[i]$ as the last element in the subsequence. The recurrence on the maximum possible length, is

$$M[n] = \begin{cases} 1 & n = 1 \\ 1 + \max_{1 \leq k < n, A[k] < A[n]} M[k] & \text{otherwise.} \end{cases}$$

The length of the longest increasing subsequence is $\max_{i=1}^n M[i]$. Again, using dynamic programming, we get an algorithm with running time $O(n^2)$ for this problem. It is also not hard to modify the algorithm so that it outputs this sequence (you should figure out the details of this modification). A better $O(n \log n)$ solution is possible using some data-structure magic.

5.5. Pattern Matching

Tidbit 5.5.1. *Magna Carta* or *Magna Charta* - the great charter that King John of England was forced by the English barons to grant at Runnymede, June 15, 1215, traditionally interpreted as guaranteeing certain civil and political liberties.

tidbit

Assume you have a string $S = \text{"Magna Carta"}$ and a pattern $P = \text{"?ag * at * a *"}$ where “?” can match a single character, and “*” can match any substring. You would like to decide if the pattern matches the string.

We are interested in solving this problem using dynamic programming. This is not too hard since this is similar to the edit-distance problem that was already covered.

```

IsMatch(S[1...n], P[1...m])
  if m = 0 and n = 0 then return TRUE.
  if m = 0 then return FALSE.
  if n = 0 then
    if P[1...m] is all stars then return TRUE
    else return FALSE
  if (P[m] = '?') then
    return IsMatch(S[1...n - 1], P[1...m - 1])
  if (P[m] ≠ '*') then
    if P[m] ≠ S[n] then return FALSE
    else return IsMatch(S[1...n - 1], P[1...m - 1])
  for i = 0 to n do
    if IsMatch(S[1...i], P[1...m - 1]) then
      return TRUE
  return FALSE

```

The resulting code is depicted on the left, and as you can see this is pretty tedious. Now, use memoization together with this recursive code, and you get an algorithm with running time $O(mn^2)$ and space $O(nm)$, where the input string of length n , and m is the length of the pattern.

Being slightly more clever, one can get a faster algorithm with running time $O(nm)$. BTW, one can do even better. A $O(m + n)$ time is possible but it requires Knuth-Morris-Pratt algorithm, which is a fast string matching algorithm.

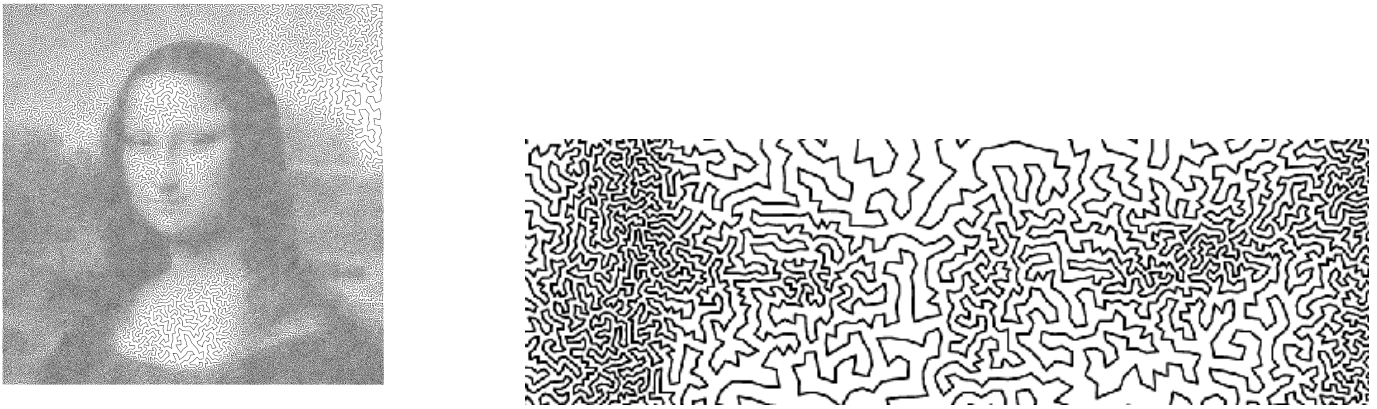


Figure 5.2: A drawing of the Mona Lisa by solving a TSP instance. The figure on the right is the TSP in the eyes region.

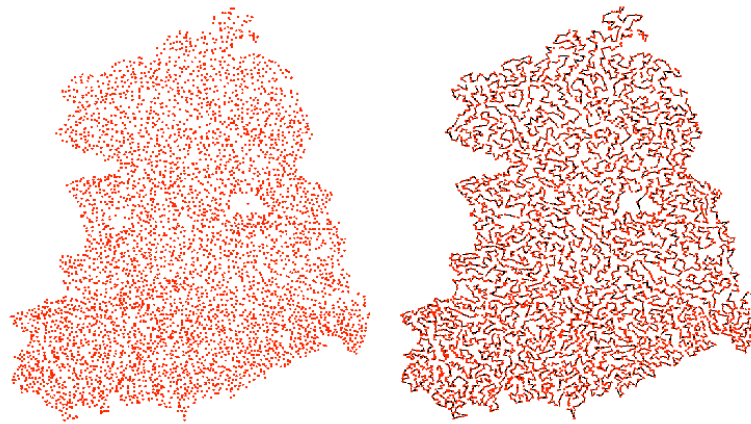


Figure 5.3: A certain country and its optimal TSP tour.

5.6. Slightly faster TSP algorithm via dynamic programming

TSP: Traveling Salesperson Problem

Instance: A graph $G = (V, E)$ with non-negative edge costs/lengths. Cost $c(e)$ for each edge $e \in E$.

Question: Find a tour of minimum cost that visits each node.

No polynomial time algorithm known for TSP— the problem is **NP-HARD**.

Even an exponential Time algorithm requires some work. Indeed, there are $n!$ potential TSP tours. Clearly, $n! \leq n^n = \exp(n \ln n)$ and $n! \geq (n/2)^{n/2} = \exp((n/2) \ln(n/2))$. Using Stirling's formula, we have $n! \approx \sqrt{n}(n/e)^n$, which gives us a somewhat tighter estimate $n! = \Theta(2^{cn \log n})$ for some constant $c > 1$.

So, naively, any running time algorithm would have running time (at least) $\Omega(n!)$. Can we do better? Can we get a $\approx 2^{O(n)}$ running time algorithm in this case?

Towards a Recursive Solution.

- (A) Order the vertices of V in some arbitrary order: v_1, v_2, \dots, v_n .
- (B) $\text{opt}(S)$: optimum TSP tour for the vertices $S \subseteq V$ in the graph restricted to S . We would like to compute $\text{opt}(V)$.

Can we compute $\text{opt}(S)$ recursively?

- (A) Say $v \in S$. What are the two neighbors of v in optimum tour in S ?
- (B) If u, w are neighbors of v in an optimum tour of S then removing v gives an optimum *path* from u to w visiting all nodes in $S - \{v\}$.

Path from u to w is not a recursive subproblem! Need to find a more general problem to allow recursion.

We start with a more general problem: **TSP Path**.

TSP Path

Instance: A graph $G = (V, E)$ with non-negative edge costs/lengths($c(e)$ for edge e) and two nodes s, t .

Question: Find a path from s to t of minimum cost that visits each node exactly once.

We can solve the regular TSP problem using this problem.

We define a recursive problem for the optimum TSP Path problem, as follows:

$\text{opt}(u, v, S)$: optimum TSP Path from u to v in the graph restricted to S s.t. $u, v \in S$.

- (A) What is the next node in the optimum path from u to v ?
- (B) Suppose it is w . Then what is $\text{opt}(u, v, S)$?
- (C) $\text{opt}(u, v, S) = c(u, w) + \text{opt}(w, v, S - \{u\})$
- (D) We do not know w ! So try all possibilities for w .

A Recursive Solution.

- (A) $\text{opt}(u, v, S) = \min_{w \in S, w \neq u, v} (c(u, w) + \text{opt}(w, v, S - \{u\}))$
- (B) What are the subproblems for the original problem $\text{opt}(s, t, V)$? For every subset $S \subseteq V$, we have the subproblem $\text{opt}(u, v, S)$ for $u, v \in S$.

As usual, we need to bound the number subproblems in the recursion:

- (A) number of distinct subsets S of V is at most 2^n
- (B) number of pairs of nodes in a set S is at most n^2
- (C) hence number of subproblems is $O(n^2 2^n)$

Exercise 5.6.1. Show that one can compute TSP using above dynamic program in $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Lemma 5.6.2. Given a graph G with n vertices, one can solve TSP in $O(n^3 2^n)$ time.

The disadvantage of dynamic programming solution is that it uses a lot of memory.

Part III

Approximation algorithms

Chapter 6

Approximation algorithms

6.1. Greedy algorithms and approximation algorithms

A natural tendency in solving algorithmic problems is to locally do what seems to be the right thing. This is usually referred to as *greedy algorithms*. The problem is that usually these kind of algorithms do not really work. For example, consider the following optimization version of **Vertex Cover**:

VertexCoverMin

Instance: A graph G , and integer k .

Question: Return the **smallest** subset $S \subseteq V(G)$, s.t. S touches all the edges of G .

For this problem, the greedy algorithm will always take the vertex with the highest degree (i.e., the one covering the largest number of edges), add it to the cover set, remove it from the graph, and repeat. We will refer to this algorithm as **GreedyVertexCover**.

It is not too hard to see that this algorithm does not output the optimal vertex-cover. Indeed, consider the graph depicted on the right. Clearly, the optimal solution is the black vertices, but the greedy algorithm would pick the four white vertices.

This of course still leaves open the possibility that, while we do not get the optimal vertex cover, what we get is a vertex cover which is “relatively good” (or “good enough”).

Definition 6.1.1. A *minimization problem* is an optimization problem, where we look for a valid solution that minimizes a certain target function.

Example 6.1.2. In the **VertexCoverMin** problem the (minimization) target function is the size of the cover. Formally $\text{Opt}(G) = \min_{S \subseteq V(G), S \text{ cover of } G} |S|$.

The $\text{VertexCover}(G)$ is just the set S realizing this minimum.

Definition 6.1.3. Let $\text{Opt}(G)$ denote the value of the target function for the optimal solution.

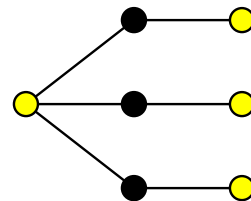


Figure 6.1: Example.

Intuitively, a vertex-cover of size “close” to the optimal solution would be considered to be good.

Definition 6.1.4. Algorithm **Alg** for a minimization problem **Min** achieves an approximation factor $\alpha \geq 1$ if for all inputs G , we have:

$$\frac{\text{Alg}(G)}{\text{Opt}(G)} \leq \alpha.$$

We will refer to **Alg** as an α -*approximation algorithm* for **Min**.

As a concrete example, an algorithm is a 2-approximation for **Vertex-CoverMin**, if it outputs a vertex-cover which is at most twice the size of the optimal solution for vertex cover.

So, how good (or bad) is the **GreedyVertexCover** algorithm described above? Well, the graph in **Figure 6.1** shows that the approximation factor of **GreedyVertexCover** is *at least* $4/3$.

It turns out that **GreedyVertexCover** performance is considerably worse. To this end, consider the following bipartite graph: $G_n = (L \cup R, E)$, where L is a set of n vertices. Next, for $i = 2, \dots, n$, we add a set R_i of $\lfloor n/i \rfloor$ vertices, to R , each one of them of degree i , such that all of them (i.e., all vertices of degree i at L) are connected to distinct vertices in R . The execution of **GreedyVertexCover** on such a graph is shown on the right.

Clearly, in G_n all the vertices in L have degree at most $n - 1$, since they are connected to (at most) one vertex of R_i , for $i = 2, \dots, n$. On the other hand, there is a vertex of degree n at R (i.e., the single vertex of R_n). Thus, **GreedyVertexCover** will first remove this vertex. We claim, that **GreedyVertexCover** will remove all the vertices of R_2, \dots, R_n and put them into the vertex-cover. To see that, observe that if R_2, \dots, R_i are still active, then all the nodes of R_i have degree i , all the vertices of L have degree at most $i - 1$, and all the vertices of R_2, \dots, R_{i-1} have degree strictly smaller than i . As such, the greedy algorithms will use the vertices of R_i . Easy induction now implies that all the vertices of R are going to be picked by **GreedyVertexCover**. This implies the following lemma.

Lemma 6.1.5. *The algorithm **GreedyVertexCover** is $\Omega(\log n)$ approximation to the optimal solution to **Vertex-CoverMin**.*

Proof: Consider the graph G_n above. The optimal solution is to pick all the vertices of L to the vertex cover, which results in a cover of size n . On the other hand, the greedy algorithm picks the set R . We have that

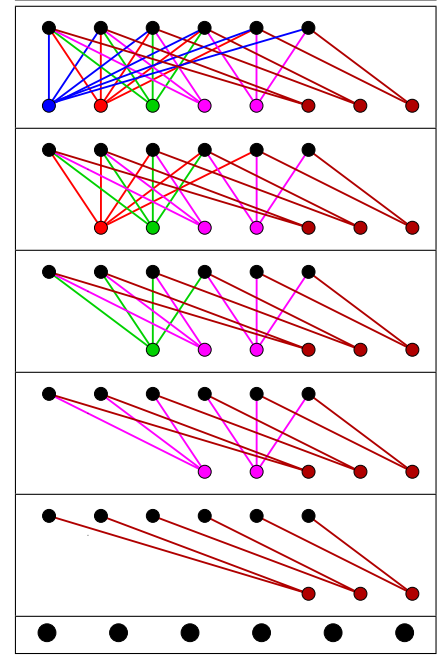
$$|R| = \sum_{i=2}^n |R_i| = \sum_{i=2}^n \left\lfloor \frac{n}{i} \right\rfloor \geq \sum_{i=2}^n \left(\frac{n}{i} - 1 \right) \geq n \sum_{i=1}^n \frac{1}{i} - 2n = n(H_n - 2).$$

Here, $H_n = \sum_{i=1}^n 1/i = \lg n + \Theta(1)$ is the n th harmonic number. As such, the approximation ratio for **GreedyVertexCover** is $\geq \frac{|R|}{|L|} = \frac{n(H_n - 2)}{n} = \Omega(\log n)$. ■

Theorem 6.1.6. *The greedy algorithm for **VertexCover** achieves $\Theta(\log n)$ approximation, where n is the number of vertices in the graph. Its running time is $O(mn^2)$.*

Proof: The lower bound follows from **Lemma 6.1.5**. The upper bound follows from the analysis of the greedy of **Set Cover**, which will be done shortly.

As for the running time, each iteration of the algorithm takes $O(mn)$ time, and there are at most n iterations. ■



6.1.1. Alternative algorithm – two for the price of one

One can still do much better than the greedy algorithm in this case. In particular, let `ApproxVertexCover` be the algorithm that chooses an edge from G , add both endpoints to the vertex cover, and removes the two vertices (and all the edges adjacent to these two vertices) from G . This process is repeated till G has no edges. Clearly, the resulting set of vertices is a vertex-cover, since the algorithm removes an edge only if it is being covered by the generated cover.

Theorem 6.1.7. *`ApproxVertexCover` is a 2-approximation algorithm for `VertexCoverMin` that runs in $O(n^2)$ time.*

Proof: Every edge picked by the algorithm contains at least one vertex of the optimal solution. As such, the cover generated is at most twice larger than the optimal. ■

6.2. Fixed parameter tractability, approximation, and fast exponential time algorithms (to say nothing of the dog)

6.2.1. A silly brute force algorithm for vertex cover

So given a graph $G = (V, E)$ with n vertices, we can approximate `VertexCoverMin` up to a factor of two in polynomial time. Let K be this approximation – we know that any vertex cover in G must be of size at least $K/2$, and we have a cover of size K . Imagine the case that K is truly small – can we compute the optimal vertex-cover in this case quickly? Well, of course, we could just try all possible subsets of vertices size at most K , and check for each one whether it is a cover or not. Checking if a specific set of vertices is a cover takes $O(m) = O(n^2)$ time, where $m = |E|$. So, the running time of this algorithm is

$$\sum_{i=1}^K \binom{n}{i} O(n^2) \leq \sum_{i=1}^K O(n^i \cdot n^2) = O(n^{K+2}),$$

where $\binom{n}{i}$ is the number of subsets of the vertices of G of size exactly i . Observe that we do not require to know K – the algorithm can just try all sizes of subsets, till it finds a solution. We thus get the following (not very interesting result).

Lemma 6.2.1. *Given a graph $G = (V, E)$ with n vertices, one can solve `VertexCoverMin` in $O(n^{\alpha+2})$ time, where α is the size the minimum vertex cover.*

6.2.2. A fixed parameter tractable algorithm

As before, our input is a graph $G = (V, E)$, for which we want to compute a vertex-cover of minimum size. We need the following definition:

Definition 6.2.2. Let $G = (V, E)$ be a graph. For a subset $S \subseteq V$, let G_S be the *induced subgraph* over S . Namely, it is a graph with the set of vertices being S . For any pair of vertices $x, y \in V$, we have that the edge $xy \in E(G_S)$ if and only if $xy \in E(G)$, and $x, y \in S$.

Also, in the following, for a vertex v , let $N_G(v)$ denote the set of vertices of G that are adjacent to v .

Consider an edge $e = uv$ in G . We know that either u or v (or both) must be in any vertex cover of G , so consider the brute force algorithm for `VertexCoverMin` that tries all these possibilities. The resulting algorithm `algFPVertexCover` is depicted in [Figure 6.2](#).

Lemma 6.2.3. *The algorithm `algFPVertexCover` (depicted in [Figure 6.2](#)) returns the optimal solution to the given instance of `VertexCoverMin`.*

```

fpVertexCoverInner (X,β)
    // Computes minimum vertex cover for the induced graph GX
    // β: size of VC computed so far.
    if X = ∅ or GX has no edges then return β
    e ← any edge uv of GX.
    β1 = fpVertexCoverInner(X \ {u,v},β + 2)
    // Only take u to the cover, but then we must also take
    // all the vertices that are neighbors of v,
    // to cover their edges with v
    β2 = fpVertexCoverInner(X \ ({u} ∪ NGX(v)),β + |NGX(v)|)
    // Only take v to the cover...
    β3 = fpVertexCoverInner(X \ ({v} ∪ NGX(u)),β + |NGX(u)|)
    return min(β1,β2,β3).

algFPVertexCover (G = (V,E))
    return fpVertexCoverInner (V,0)

```

Figure 6.2: Fixed parameter tractable algorithm for **VertexCoverMin**.

Proof: It is easy to verify, that if the algorithm returns β then it found a vertex cover of size β . Since the depth of the recursion is at most n , it follows that this algorithm always terminates.

Consider the optimal solution $Y \subseteq V$, and run the algorithm, where every stage of the recursion always pick the option that complies with the optimal solution. Clearly, since in every level of the recursion at least one vertex of Y is being found, then after $O(|Y|)$ recursive calls, the remaining graph would have no edges, and it would return $|Y|$ as one of the candidate solution. Furthermore, since the algorithm always returns the minimum solution encountered, it follows that it would return the optimal solution. ■

Lemma 6.2.4. *The depth of the recursion of $\text{algFPVertexCover}(G)$ is at most α , where α is the minimum size vertex cover in G .*

Proof: The idea is to consider all the vertices that can be added to the vertex cover being computed without covering any new edge. In particular, in the case the algorithm takes both u and v to the cover, then one of these vertices must be in the optimal solution, and this can happen at most α times.

The more interesting case, is when the algorithm picks $N_{G_X}(v)$ (i.e., β_2) to the vertex cover. We can add v to the vertex cover in this case without getting any new edges being covered (again, we are doing this only conceptually – the vertex cover computed by the algorithm would not contain v [only its neighbors]). We do the same thing for the case of β_3 .

Now, observe that in any of these cases, the hypothetical set cover being constructed (which has more vertices than what the algorithm computes, but covers exactly the same set of edges in the original graph) contains one vertex of the optimal solution picked into itself in each level of the recursion. Clearly, the algorithm is done once we pick all the vertices of the optimal solution into the hypothetical vertex cover. It follows that the depth the recursion is $\leq \alpha$. ■

Theorem 6.2.5. *Let G be a graph with n vertices, and with the minimal vertex cover being of size α . Then, the algorithm algFPVertexCover (depicted in **Figure 6.2**) returns the optimal vertex cover for G and the running time of this algorithm is $O(3^\alpha n^2)$.*

Proof: By **Lemma 6.2.4**, the recursion tree has depth α . As such, it contains at most $2 \cdot 3^\alpha$ nodes. Each node in the recursion requires $O(n^2)$ work (ignoring the recursive calls), if implemented naively. Thus, the bound on the running time follows. ■

Algorithms where the running time is of the form $O(n^c f(\alpha))$, where α is some parameter that depends on the problem are *fixed parameter tractable* algorithms for the given problem.

6.2.2.1. Remarks

Currently, the fastest algorithm known for this problem has running time $O(1.2738^\alpha + \alpha n)$ [CKX10]. This algorithm uses similar ideas, but is considerably more complicated.

It is known that no better approximation than 1.3606 is possible for **VertexCoverMin**, unless $P = NP$. The currently best approximation known is $2 - \Theta\left(\frac{1}{\sqrt{\log n}}\right)$. If the *Unique Games Conjecture* is true, then no better constant approximation is possible in polynomial time.

6.3. Approximating maximum matching

Definition 6.3.1. Consider an undirected graph $G = (V, E)$. The graph might have a weight function $\omega(e)$, specifying a positive value on the edges of G (if no weights are specified, treat every edge as having weight 1).

- A subset $M \subseteq E$ is a *matching* if no pair of edges of M share endpoints.
- A *perfect matching* is a matching that covers all the vertices of G .
- A *min-weight perfect matching*, is the minimum weight matching among all perfect matching, where the *weight* of a matching is

$$\omega(M) = \sum_{e \in M} \omega(e).$$

- The *maximum-weight matching* (or just *maximum matching* is the matching with maximum weight among all matchings.
- A matching M is *maximal* if no edge can be added to it. That is, for every edge $e \in E$, we have that the edges of M contains at least one endpoint of e .

Note the subtle difference between maximal and maximum – the first, is a local maximum, while the other one is the global maximum.

Lemma 6.3.2. *Lemma ?? Given an undirected unweighted graph G with n vertices and m edges, one can compute a matching M in G , such that $|M| \geq |\text{opt}|/2$, where opt is the maximum size (i.e., cardinality) matching in G . The running time is $O(n + m)$.*

Proof: The algorithm is shockingly simple – repeatedly pick an edge of G , remove it and the edges adjacent to it, and repeat till there are no edges left in the graph. Let M be the resulting matching.

To see why this is a two approximation (i.e., $2|M| \geq |\text{opt}|$), observe that every edge of M is adjacent to at most two edges of opt . As such, each edge of M pays for two edges of opt , which implies the claim.

One way to see that is to imagine that we start with the matching opt and let $M = \{m_1, \dots, m_t\}$ – at each iteration, we insert m_i into the current matching, and remove any old edges that intersect it. As such, we moved from the matching of M to the matching of opt . In each step, we deleted at most two edges, and inserted one edges. As such, $|\text{opt}| \leq 2|M|$. ■

Lemma 6.3.3. *Given an undirected weighted graph G with n vertices and m edges, one can compute a matching M in G , such that $\omega(M) \geq \omega(\text{opt})/2$, where opt is the maximum weight matching in G . The running time is $O(n \log n + m)$.*

Proof: We run the algorithm for the unweighted case, with the modification that we always pick the heaviest edge still available. The same argument as in Lemma ?? implies that that this is a two approximation. As for the running time – we need a min-heap for m elements, that performs at most n deletions, and as such, the running time is $O(n \log n + m)$ by using a Fibonacci heap. ■

Remark 6.3.4. Note, that maximum matching (and all the variants mentioned above) are solvable in polynomial time. The main thing is that the above algorithm is both simple and give us a decent starting point which can be used in the exact algorithm.

6.4. Graph diameter

FILL IN.

6.5. Traveling Salesman Person

We remind the reader that the optimization variant of the TSP problem is the following.

TSP-Min

Instance: $G = (V, E)$ a complete graph, and $\omega(e)$ a cost function on edges of G .
Question: The cheapest tour that visits all the vertices of G exactly once.

Theorem 6.5.1. *TSP-Min can not be approximated within **any** factor unless $NP = P$.*

Proof: Consider the reduction from **Hamiltonian Cycle** into **TSP**. Given a graph G , which is the input for the Hamiltonian cycle, we transform it into an instance of **TSP-Min**. Specifically, we set the weight of every edge to 1 if it was present in the instance of the Hamiltonian cycle, and 2 otherwise. In the resulting complete graph, if there is a tour price n then there is a Hamiltonian cycle in the original graph. If on the other hand, there was no cycle in G then the cheapest TSP is of price $n + 1$.

Instead of 2, let us assign the missing edges in H a weight of cn , for c an arbitrary number. Let H denote the resulting graph. Clearly, if G does not contain any Hamiltonian cycle in the original graph, then the price of the **TSP-Min** in H is at least $cn + 1$.

Note, that the prices of tours of H are either (i) equal to n if there is a Hamiltonian cycle in G , or (ii) larger than $cn + 1$ if there is no Hamiltonian cycle in G . As such, if one can do a c -approximation, in polynomial time, to **TSP-Min**, then using it on H would yield a tour of price $\leq cn$ if a tour of price n exists. But a tour of price $\leq cn$ exists if and only if G has a Hamiltonian cycle.

Namely, such an approximation algorithm would solve a **NP-COMplete** problem (i.e., **Hamiltonian Cycle**) in polynomial time. ■

Note, that **Theorem 6.5.1** implies that **TSP-Min** can not be approximated to within any factor. However, once we add some assumptions to the problem, it becomes much more manageable (at least as far as approximation).

What the above reduction did, was to take a problem and reduce it into an instance where this is a huge gap, between the optimal solution, and the second cheapest solution. Next, we argued that if had an approximation algorithm that has ratio better than the ratio between the two endpoints of this empty interval, then the approximation algorithm, would in polynomial time would be able to decide if there is an optimal solution.

6.5.1. TSP with the triangle inequality

6.5.1.1. A 2-approximation

Consider the following special case of **TSP**:

TSP _{$\Delta \neq$} -Min

Instance: $G = (V, E)$ is a complete graph. There is also a cost function $\omega(\cdot)$ defined over the edges of G , that complies with the triangle inequality.
Question: The cheapest tour that visits all the vertices of G exactly once.

We remind the reader that the *triangle inequality* holds for $\omega(\cdot)$ if

$$\forall u, v, w \in V(G), \quad \omega(u, v) \leq \omega(u, w) + \omega(w, v).$$

The triangle inequality implies that if we have a path σ in G , that starts at s and ends at t , then $\omega(st) \leq \omega(\sigma)$. Namely, *shortcutting*, that is going directly from s to t , is always beneficial if the triangle inequality holds (assuming that we do not have any reason to visit the other vertices of σ).

Definition 6.5.2. A cycle in a graph G is *Eulerian* if it visits every edge of G exactly once.

Unlike Hamiltonian cycle, which has to visit every vertex exactly once, an Eulerian cycle might visit a vertex an arbitrary number of times. We need the following classical result:

Lemma 6.5.3. *A graph G has a cycle that visits every edge of G exactly once (i.e., an Eulerian cycle) if and only if G is connected, and all the vertices have even degree. Such a cycle can be computed in $O(n + m)$ time, where n and m are the number of vertices and edges of G , respectively.*

Our purpose is to come up with a 2-approximation algorithm for **TSP _{$\Delta \neq$} -Min**. To this end, let C_{opt} denote the optimal **TSP** tour in G . Observe that C_{opt} is a spanning graph of G , and as such we have that

$$\omega(C_{\text{opt}}) \geq \text{weight}(\text{cheapest spanning graph of } G).$$

But the cheapest spanning graph of G , is the minimum spanning tree (MST) of G , and as such $\omega(C_{\text{opt}}) \geq \omega(\text{MST}(G))$. The MST can be computed in $O(n \log n + m) = O(n^2)$ time, where n is the number of vertices of G , and $m = \binom{n}{2}$ is the number of edges (since G is the complete graph). Let T denote the MST of G , and convert T into a tour by duplicating every edge twice. Let H denote the new graph. We have that H is a connected graph, every vertex of H has even degree, and as such H has an Eulerian tour (i.e., a tour that visits every edge of H exactly once).

As such, let C denote the Eulerian cycle in H . Observe that

$$\omega(C) = \omega(H) = 2\omega(T) = 2\omega(\text{MST}(G)) \leq 2\omega(C_{\text{opt}}).$$

Next, we traverse C starting from any vertex $v \in V(C)$. As we traverse C , we skip vertices that we already visited, and in particular, the new tour we extract from C will visit the vertices of $V(G)$ in the order they first appear in C . Let π denote the new tour of G . Clearly, since we are performing shortcutting, and the triangle inequality holds, we have that $\omega(\pi) \leq \omega(C)$. The resulting algorithm is depicted in **Figure 6.3**.

It is easy to verify, that all the steps of our algorithm can be done in polynomial time. As such, we have the following result.

Theorem 6.5.4. *Given an instance of TSP with the triangle inequality (**TSP _{$\Delta \neq$} -Min**) (namely, a graph G with n vertices and $\binom{n}{2}$ edges, and a cost function $\omega(\cdot)$ on the edges that comply with the triangle inequality), one can compute a tour of G of length $\leq 2\omega(C_{\text{opt}})$, where C_{opt} is the minimum cost TSP tour of G . The running time of the algorithm is $O(n^2)$.*

6.5.1.2. A 3/2-approximation to TSP _{$\Delta \neq$} -Min

The following is a known result, and we will see a somewhat weaker version of it in class.

Theorem 6.5.5. *Given a graph G and weights on the edges, one can compute the min-weight perfect matching of G in polynomial time.*

Lemma 6.5.6. *Let $G = (V, E)$ be a complete graph, S a subset of the vertices of V of even size, and $\omega(\cdot)$ a weight function over the edges. Then, the weight of the min-weight perfect matching in G_S is $\leq \omega(\text{TSP}(G))/2$.*

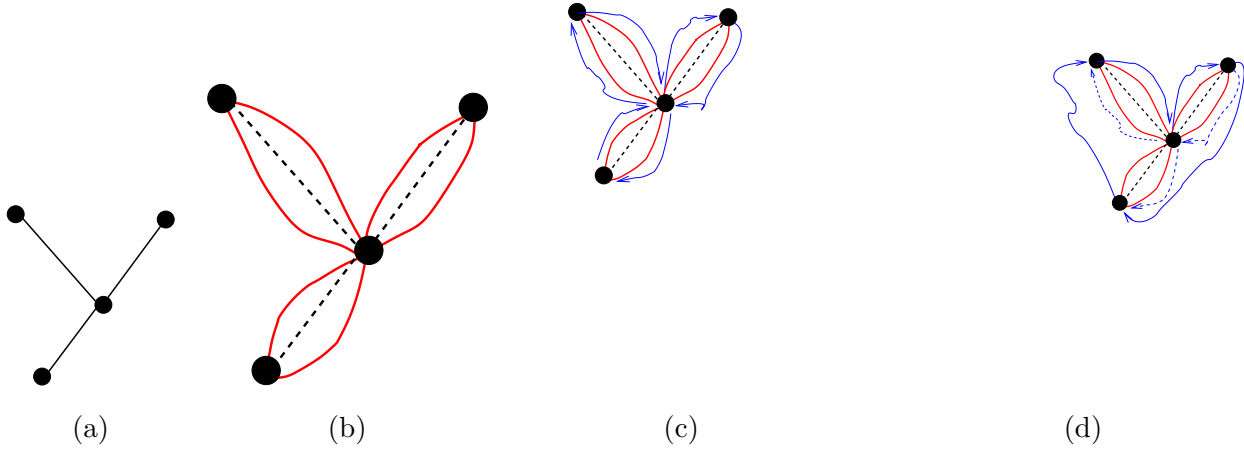


Figure 6.3: The TSP approximation algorithm: (a) the input, (b) the duplicated graph, (c) the extracted Eulerian tour, and (d) the resulting shortcut path.

Proof: Let π be the cycle realizing the TSP in G . Let σ be the cycle resulting from shortcutting π so that it uses only the vertices of S . Clearly, $\omega(\sigma) \leq \omega(\pi)$. Now, let M_e and M_o be the sets of even and odd edges of σ respectively. Clearly, both M_o and M_e are perfect matching in G_S , and

$$\omega(M_o) + \omega(M_e) = \omega(\sigma).$$

We conclude, that $\min(w(M_o), w(M_e)) \leq \omega(\text{TSP}(G))/2$. ■

We now have a creature that has the weight of half of the TSP, and we can compute it in polynomial time. How to use it to approximate the TSP? The idea is that we can make the MST of G into an Eulerian graph by being more careful. To this end, consider the tree on the right. Clearly, it is almost Eulerian, except for these pesky odd degree vertices. Indeed, if all the vertices of the spanning tree had even degree, then the graph would be Eulerian (see [Lemma 6.5.3](#)).

In particular, in the depicted tree, the “problematic” vertices are 1, 4, 2, 7, since they are all the odd degree vertices in the MST T .

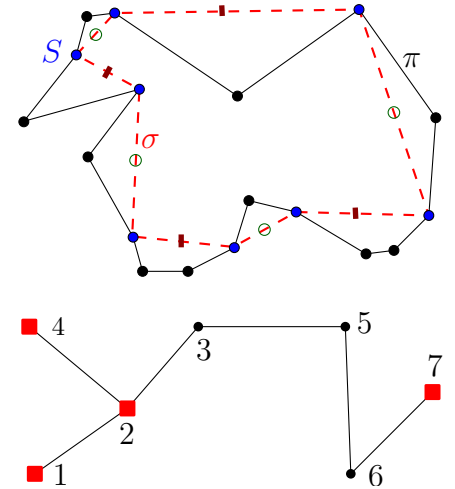
Lemma 6.5.7. *The number of odd degree vertices in any graph G' is even.*

Proof: Observe that $\mu = \sum_{v \in V(G')} d(v) = 2|E(G')|$, where $d(v)$ denotes the degree of v . Let $U = \sum_{v \in V(G'), d(v) \text{ is even}} d(v)$, and observe that U is even as it is the sum of even numbers.

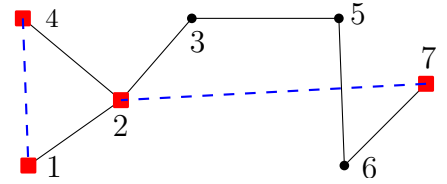
Thus, ignoring vertices of even degree, we have

$$\alpha = \sum_{v \in V, d(v) \text{ is odd}} d(v) = \mu - U = \text{even number},$$

since μ and U are both even. Thus, the number of elements in the above sum of all odd numbers must be even, since the total sum is even. ■



So, we have an even number of problematic vertices in T . The idea now is to compute a minimum-weight perfect matching M on the problematic vertices, and add the edges of the matching to the tree. The resulting graph, for our running example, is depicted on the right. Let $H = (V, E(M) \cup E(T))$ denote this graph, which is the result of adding M to T .



We observe that H is Eulerian, as all the vertices now have even degree, and the graph is connected. We also have

$$\omega(H) = \omega(\text{MST}(G)) + \omega(M) \leq \omega(\text{TSP}(G)) + \omega(\text{TSP}(G))/2 = (3/2)\omega(\text{TSP}(G)),$$

by [Lemma 6.5.6](#). Now, H is Eulerian, and one can compute the Euler cycle for H , shortcut it, and get a tour of the vertices of G of weight $\leq (3/2)\omega(\text{TSP}(G))$.

Theorem 6.5.8. *Given an instance of TSP with the triangle inequality, one can compute in polynomial time, a $(3/2)$ -approximation to the optimal TSP.*

6.6. Biographical Notes

The $3/2$ -approximation for TSP with the triangle inequality is due to Christofides [[Chr76](#)].

Chapter 7

Approximation algorithms II

7.1. Max Exact 3SAT

We remind the reader that an instance of **3SAT** is a boolean formula, for example $F = (x_1 + x_2 + x_3)(x_4 + \overline{x_1} + x_2)$, and the decision problem is to decide if the formula has a satisfiable assignment. Interestingly, we can turn this into an optimization problem.

Max 3SAT

Instance: A collection of clauses: C_1, \dots, C_m .

Question: Find the assignment to x_1, \dots, x_n that satisfies the maximum number of clauses.

Clearly, since **3SAT** is **NP-COMplete** it implies that **Max 3SAT** is **NP-HARD**. In particular, the formula F becomes the following set of two clauses:

$$x_1 + x_2 + x_3 \quad \text{and} \quad x_4 + \overline{x_1} + x_2.$$

Note, that **Max 3SAT** is a *maximization problem*.

Definition 7.1.1. Algorithm **Alg** for a maximization problem achieves an approximation factor α if for all inputs, we have:

$$\frac{\text{Alg}(G)}{\text{Opt}(G)} \geq \alpha.$$

In the following, we present a *randomized algorithm* – it is allowed to consult with a source of random numbers in making decisions. A key property we need about random variables, is the linearity of expectation property, which is easy to derive directly from the definition of expectation.

Definition 7.1.2 (**Linearity of expectations**). Given two random variables X, Y (not necessarily independent, we have that $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$.

Theorem 7.1.3. *One can achieve (in expectation) (7/8)-approximation to **Max 3SAT** in polynomial time. Namely, if the instance has m clauses, then the generated assignment satisfies (7/8) m clauses in expectation.*

Proof: Let x_1, \dots, x_n be the n variables used in the given instance. The algorithm works by randomly assigning values to x_1, \dots, x_n , independently, and equal probability, to 0 or 1, for each one of the variables.

Let Y_i be the indicator variables which is 1 if (and only if) the i th clause is satisfied by the generated random assignment and 0 otherwise, for $i = 1, \dots, m$. Formally, we have

$$Y_i = \begin{cases} 1 & C_i \text{ is satisfied by the generated assignment,} \\ 0 & \text{otherwise.} \end{cases}$$

Now, the number of clauses satisfied by the given assignment is $Y = \sum_{i=1}^m Y_i$. We claim that $\mathbb{E}[Y] = (7/8)m$, where m is the number of clauses in the input. Indeed, we have

$$\mathbb{E}[Y] = \mathbb{E}\left[\sum_{i=1}^m Y_i\right] = \sum_{i=1}^m \mathbb{E}[Y_i]$$

by linearity of expectation. Now, what is the probability that $Y_i = 0$? This is the probability that all three literals appear in the clause C_i are evaluated to **FALSE**. Since the three literals are instance of three distinct variable, these three events are independent, and as such the probability for this happening is

$$\mathbb{P}[Y_i = 0] = \frac{1}{2} * \frac{1}{2} * \frac{1}{2} = \frac{1}{8}.$$

(Another way to see this, is to observe that since C_i has exactly three literals, there is only one possible assignment to the three variables appearing in it, such that the clause evaluates to **FALSE**. Now, there are eight (8) possible assignments to this clause, and thus the probability of picking a **FALSE** assignment is 1/8.) Thus,

$$\mathbb{P}[Y_i = 1] = 1 - \mathbb{P}[Y_i = 0] = \frac{7}{8},$$

and

$$\mathbb{E}[Y_i] = \mathbb{P}[Y_i = 0] * 0 + \mathbb{P}[Y_i = 1] * 1 = \frac{7}{8}.$$

Namely, $\mathbb{E}[\# \text{ of clauses sat}] = \mathbb{E}[Y] = \sum_{i=1}^m \mathbb{E}[Y_i] = (7/8)m$. Since the optimal solution satisfies at most m clauses, the claim follows. ■

Curiously, **Theorem 7.1.3** is stronger than what one usually would be able to get for an approximation algorithm. Here, the approximation quality is independent of how well the optimal solution does (the optimal can satisfy at most m clauses, as such we get a (7/8)-approximation. Curiouser and curiouser^①, the algorithm does not even look on the input when generating the random assignment.

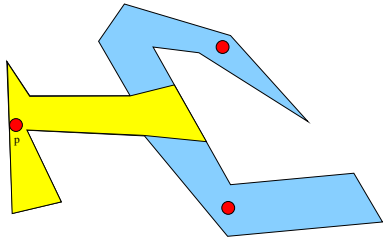
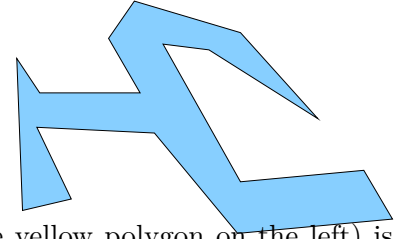
Håstad [Hås01a] proved that one can do no better; that is, for any constant $\varepsilon > 0$, one can not approximate **3SAT** in polynomial time (unless $\text{P} = \text{NP}$) to within a factor of $7/8 + \varepsilon$. It is pretty amazing that a trivial algorithm like the above is essentially optimal.

^①“Curiouser and curiouser!” Cried Alice (she was so much surprised, that for the moment she quite forgot how to speak good English). – Alice in wonderland, Lewis Carol

7.2. Approximation Algorithms for Set Cover

7.2.1. Guarding an Art Gallery

You are given the floor plan of an art gallery, which is a two dimensional simple polygon. You would like to place guards that see the whole polygon. A guard is a point, which can see all points around it, but it can not see through walls. Formally, a point p can *see* a point q , if the segment pq is contained inside the polygon. See figure on the right, for an illustration of how the input looks like.



A *visibility polygon* at p (depicted as the yellow polygon on the left) is the region inside the polygon that p can see. We would like to find the *minimal* number of guards needed to guard the given art-gallery? That is, all the points in the art gallery should be visible from at least one guard we place.

The art-gallery problem is a set-cover problem. We have a ground set (the polygon), and family of sets (the set of all visibility polygons), and the target is to find a minimal number of sets covering the whole polygon.

It is known that finding the minimum number of guards needed is **NP-HARD**. No approximation is currently known. It is also known that a polygon with n corners, can be guarded using $n/3 + 1$ guards. Note, that this problem is harder than the classical set-cover problem because the number of subsets is infinite and the underlining base set is also infinite.

An interesting *open problem* is to find a polynomial time approximation algorithm, such that given P , it computes a set of guards, such that $\#guards \leq \sqrt{nk_{opt}}$, where n is the number of vertices of the input polygon P , and k_{opt} is the number of guards used by the optimal solution.

7.2.2. Set Cover

The optimization version of **Set Cover**, is the following:

Set Cover

Instance: (S, \mathcal{F}) :

S - a set of n elements

\mathcal{F} - a family of subsets of S , s.t. $\bigcup_{X \in \mathcal{F}} X = S$.

Question: The set $\mathcal{X} \subseteq \mathcal{F}$ such that \mathcal{X} contains as few sets as possible, and \mathcal{X} covers S . Formally, $\bigcup_{X \in \mathcal{X}} X = S$.

The set S is sometime called the *ground set*, and a pair (S, \mathcal{F}) is either called a *set system* or a *hypergraph*. Note, that **Set Cover** is a minimization problem which is also **NP-HARD**.

Example 7.2.1. Consider the set $S = \{1, 2, 3, 4, 5\}$ and the following family of subsets

$$\mathcal{F} = \{\{1, 2, 3\}, \{2, 5\}, \{1, 4\}, \{4, 5\}\}.$$

Clearly, the smallest cover of S is $\mathcal{X}_{opt} = \{\{1, 2, 3\}, \{4, 5\}\}$.

The greedy algorithm **GreedySetCover** for this problem is depicted in **Figure 7.1**. Here, the algorithm always picks the set in the family that covers the largest number of elements not covered yet. Clearly, the algorithm is polynomial in the input size. Indeed, we are given a set S of n elements, and m subsets. As such, the input size is at least $\Omega(m + n)$ (and at most of size $O(mn)$), and the algorithm takes time polynomial in m and n . Let $\mathcal{X}_{opt} = \{V_1, \dots, V_k\}$ be the optimal solution.

Let T_i denote the elements not covered in the beginning i th iteration of **GreedySetCover**, where $T_1 = S$. Let U_i be the set added to the cover in the i th iteration, and $\alpha_i = |U_i \cap T_i|$ be the number of new elements being covered in the i th iteration.

```

GreedySetCover( $S, \mathcal{F}$ )
 $\mathcal{X} \leftarrow \emptyset$ ;  $T \leftarrow S$ 
while  $T$  is not empty do
     $U \leftarrow$  set in  $\mathcal{F}$  covering largest
        # of elements in  $T$ 
     $\mathcal{X} \leftarrow \mathcal{X} \cup \{U\}$ 
     $T \leftarrow T \setminus U$ 

return  $\mathcal{X}$ .

```

Figure 7.1

Claim 7.2.2. We have $\alpha_1 \geq \alpha_2 \geq \dots \geq \alpha_k \geq \dots \geq \alpha_m$.

Proof: If $\alpha_i < \alpha_{i+1}$ then U_{i+1} covers more elements than U_i and we can exchange between them, as we found a set that in the i th iteration covers more elements than the set used by **GreedySetCover**. Namely, in the i th iteration we would use U_{i+1} instead of U_i . This contradicts the greediness of **GreedySetCover** of choosing the set covering the largest number of elements not covered yet. A contradiction. ■

Claim 7.2.3. We have $\alpha_i \geq |T_i|/k$. Namely, $|T_{i+1}| \leq (1 - 1/k)|T_i|$.

Proof: Consider the optimal solution. It is made out of k sets and it covers S , and as such it covers $T_i \subseteq S$. This implies that one of the subsets in the optimal solution cover at least $1/k$ fraction of the elements of T_i . Finally, the greedy algorithm picks the set that covers the largest number of elements of T_i . Thus, U_i covers at least $\alpha_i \geq |T_i|/k$ elements.

As for the second claim, we have that $|T_{i+1}| = |T_i| - \alpha_i \leq (1 - 1/k)|T_i|$. ■

Theorem 7.2.4. The algorithm **GreedySetCover** generates a cover of S using at most $O(k \log n)$ sets of \mathcal{F} , where k is the size of the cover in the optimal solution.

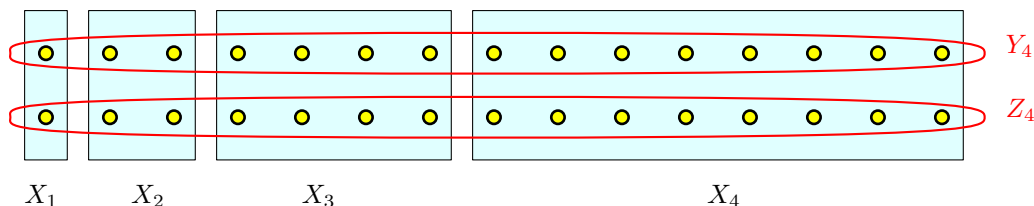
Proof: We have that $|T_i| \leq (1 - 1/k)|T_{i-1}| \leq (1 - 1/k)^i |T_0| = (1 - 1/k)^i n$. In particular, for $M = \lceil 2k \ln n \rceil$ we have

$$|T_M| \leq \left(1 - \frac{1}{k}\right)^M n \leq \exp\left(-\frac{1}{k}M\right)n = \exp\left(-\frac{\lceil 2k \ln n \rceil}{k}\right)n \leq \frac{1}{n} < 1,$$

since $1 - x \leq e^{-x}$, for $x \geq 0$. Namely, $|T_M| = 0$. As such, the algorithm terminates before reaching the M th iteration, and as such it outputs a cover of size $O(k \log n)$, as claimed. ■

7.2.3. Lower bound

The lower bound example is depicted in the following figure.



We provide a more formal description of this lower bound next, and prove that it shows $\Omega(\log n)$ approximation to **GreedySetCover**.

We want to show here that the greedy algorithm analysis is tight. To this end, consider the set system $\Lambda_i = (S_i, \mathcal{F}_i)$, where $S_i = Y_i \cup Z_i$, $Y_i = \{y_1, \dots, y_{2^{i-1}}\}$ and $Z_i = \{z_1, \dots, z_{2^{i-1}}\}$. The family of sets \mathcal{F}_i contains the following sets

$$X_j = \{y_{2^{j-1}}, \dots, y_{2^j-1}, z_{2^{j-1}}, \dots, z_{2^j-1}\},$$

for $j = 1, \dots, i$. Furthermore, \mathcal{F}_i also contains the two special sets Y_i and Z_i . Clearly, minimum set cover for Λ_i is the two sets Y_i and Z_i .

However, sets Y_i and Z_i have size $2^i - 1$. But, the set X_i has size

$$|X_i| = 2(2^i - 1 - 2^{i-1} + 1) = 2^i,$$

and this is the largest set in Λ_i . As such, the greedy algorithm **GreedySetCover** would pick X_i as first set to its cover. However, once you remove X_i from Λ_i (and from its ground set), you remain with the set system Λ_{i-1} . We conclude that **GreedySetCover** would pick the sets X_i, X_{i-1}, \dots, X_1 to the cover, while the optimal cover is by two sets. We conclude:

Lemma 7.2.5. *Let $n = 2^{i+1} - 2$. There exists an instance of **Set Cover** of n elements, for which the optimal cover is by two sets, but **GreedySetCover** would use $i = \lfloor \lg n \rfloor$ sets for the cover. That is, **GreedySetCover** is a $\Theta(\log n)$ approximation to **SetCover**.*

7.2.4. Just for fun – weighted set cover

Weighted Set Cover

Instance: (S, \mathcal{F}, ρ) :

S : a set of n elements

\mathcal{F} : a family of subsets of S , s.t. $\bigcup_{X \in \mathcal{F}} X = S$.

$\rho(\cdot)$: A price function assigning price to each set in \mathcal{F} .

Question: The set $\mathcal{X} \subseteq \mathcal{F}$, such that \mathcal{X} covers S . Formally, $\bigcup_{X \in \mathcal{X}} X = S$, and $\rho(\mathcal{X}) = \sum_{X \in \mathcal{X}} \rho(X)$ is minimized.

The greedy algorithm in this case, **WGreedySetCover**, repeatedly picks the set that pays the least cover each element it cover. Specifically, if a set $X \in \mathcal{F}$ covered t new elements, then the **average price** it pays per element it cover is $\alpha(X) = \rho(X)/t$. **WGreedySetCover** as such, picks the set with the lowest average price. Our purpose here to prove that this greedy algorithm provides $O(\log n)$ approximation.

7.2.4.1. Analysis

Let U_i be the set of elements that are not covered yet in the end of the i th iteration. As such, $U_0 = S$. At the beginning of the i th iteration, the **average optimal cost** is $\alpha_i = \rho(\text{opt})/n_i$, where opt is the optimal solution and $n_i = |U_{i-1}|$ is the number of uncovered elements.

Lemma 7.2.6. *We have that:*

(A) $\alpha_1 \leq \alpha_2 \leq \dots$.

(B) For $i < j$, we have $2\alpha_i \leq \alpha_j$ only if $n_j \leq n_i/2$.

Proof: (A) is hopefully obvious – as the number of elements not covered decreases, the average price to cover the remaining elements using the optimal solution goes up.

(B) $2\alpha_i \leq \alpha_j$ implies that $2\rho(\text{opt})/n_i \leq \rho(\text{opt})/n_j$, which implies in turn that $2n_j \leq n_i$. ■

So, let k be the first iteration such that $n_k \leq n/2$. The basic idea is that total price that **WGreedySetCover** paid during these iterations is at most $2\rho(\text{opt})$. This immediately implies $O(\log n)$ iteration, since this can happen at most $O(\log n)$ times till the ground set is fully covered.

To this end, we need the following technical lemma.

Lemma 7.2.7. *Let U_{i-1} be the set of elements not yet covered in the beginning of the i th iteration, and let $\alpha_i = \rho(\text{opt})/n_i$ be the average optimal cost per element. Then, there exists a set X in the optimal solution, with lower average cost; that is, $\rho(X)/|U_{i-1} \cap X| \leq \alpha_i$.*

Proof: Let X_1, \dots, X_m be the sets used in the optimal solution. Let $s_j = |U_{i-1} \cap X_j|$, for $j = 1, \dots, m$, be the number of new elements covered by each one of these sets. Similarly, let $\rho_j = \rho(X_j)$, for $j = 1, \dots, m$. The average cost of the j th set is ρ_j/s_j (it is $+\infty$ if $s_j = 0$). It is easy to verify that

$$\min_{j=1}^m \frac{\rho_j}{s_j} \leq \frac{\sum_{j=1}^m \rho_j}{\sum_{j=1}^m s_j} = \frac{\rho(\text{opt})}{\sum_{j=1}^m s_j} \leq \frac{\rho(\text{opt})}{|U_{i-1}|} = \alpha_i.$$

The first inequality follows as $a/b \leq c/d$ (all positive numbers), then $\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d}$. In particular, for any such numbers $\min\left(\frac{a}{b}, \frac{c}{d}\right) \leq \frac{a+c}{b+d}$, and applying this repeatedly implies this inequality. The second inequality follows as $\sum_{j=1}^m s_j \geq |U_{i-1}|$. This implies that the optimal solution must contain a set with an average cost smaller than the average optimal cost. ■

Lemma 7.2.8. *Let k be the first iteration such that $n_k \leq n/2$. The total price of the sets picked in iteration 1 to $k-1$, is at most $2\rho(\text{opt})$.*

Proof: By [Lemma 7.2.7](#), at each iteration the algorithm picks a set with average cost that is smaller than the optimal average cost (which goes up in each iteration). However, the optimal average cost iterations, 1 to $k-1$, is at most twice the starting cost, since the number of elements not covered is at least half the total number of elements. It follows, that for each element covered, the greedy algorithm paid at most twice the initial optimal average cost. So, if the number of elements covered by the beginning of the k th iteration is $\beta \geq n/2$, then the total price paid is $2\alpha_1\beta = 2(\rho(\text{opt})/n)\beta \leq 2\rho(\text{opt})$, implying the claim. ■

Theorem 7.2.9. *[WGreedySetCover](#) computes a $O(\log n)$ approximation to the optimal weighted set cover solution.*

Proof: [WGreedySetCover](#) paid at most twice the optimal solution to cover half the elements, by [Lemma 7.2.8](#). Now, you can repeat the argument on the remaining uncovered elements. Clearly, after $O(\log n)$ such halving steps, all the sets would be covered. In each halving step, [WGreedySetCover](#) paid at most twice the optimal cost. ■

7.3. Biographical Notes

The [Max 3SAT](#) remains hard in the “easier” variant of [MAX 2SAT](#) version, where every clause has 2 variables. It is known to be [NP-HARD](#) and approximable within 1.0741 [[FG95](#)], and is not approximable within 1.0476 [[Hås01a](#)]. Notice, that the fact that [MAX 2SAT](#) is hard to approximate is surprising as [2SAT](#) can be solved in polynomial time (!).

Chapter 8

Approximation algorithms III

8.1. Clustering

Consider the problem of *unsupervised learning*. We are given a set of examples, and we would like to partition them into classes of similar examples. For example, given a webpage X about “The reality dysfunction”, one would like to find all webpages on this topic (or closely related topics). Similarly, a webpage about “All quiet on the western front” should be in the same group as webpage as “Storm of steel” (since both are about soldier experiences in World War I).

The hope is that all such webpages of interest would be in the same cluster as X , if the clustering is good.

More formally, the input is a set of examples, usually interpreted as points in high dimensions. For example, given a webpage W , we represent it as a point in high dimensions, by setting the i th coordinate to 1 if the word w_i appears somewhere in the document, where we have a prespecified list of 10,000 words that we care about. Thus, the webpage W can be interpreted as a point of the $\{0,1\}^{10,000}$ hypercube; namely, a point in 10,000 dimensions.

Let X be the resulting set of n points in d dimensions.

To be able to partition points into similar clusters, we need to define a notion of similarity. Such a similarity measure can be any distance function between points. For example, consider the “regular” Euclidean distance between points, where

$$\|p - q\| = \sqrt{\sum_{i=1}^d (p_i - q_i)^2},$$

where $p = (p_1, \dots, p_d)$ and $q = (q_1, \dots, q_d)$.

As another motivating example, consider the *facility location problem*. We are given a set X of n cities and distances between them, and we would like to build k hospitals, so that the maximum distance of a city from its closest hospital is minimized. (So that the maximum time it would take a patient to get to the its closest hospital is bounded.)

Intuitively, what we are interested in is selecting good representatives for the input point-set X . Namely, we would like to find k points in X such that they represent X “well”.

Formally, consider a subset S of k points of X , and a p a point of X . The *distance of p from the set S* is

$$\mathbf{d}(p, S) = \min_{q \in S} \|p - q\|;$$

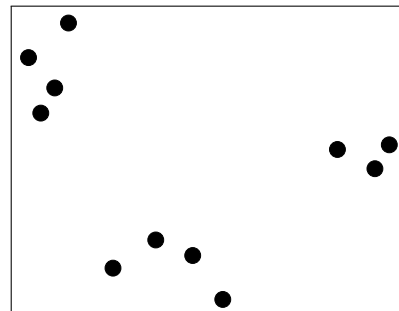
namely, $\mathbf{d}(p, S)$ is the minimum distance of a point of S to p . If we interpret S as a set of centers then $\mathbf{d}(p, S)$ is the distance of p to its closest center.

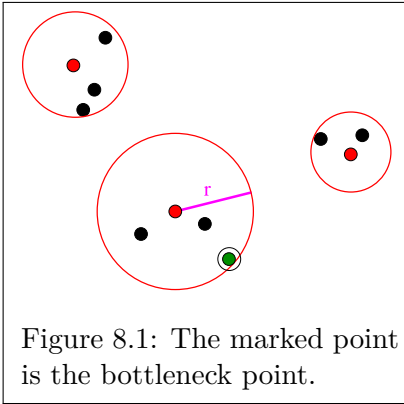
Now, the *price of clustering* X by the set S is

$$v(X, S) = \max_{p \in X} \mathbf{d}(p, S).$$

This is the maximum distance of a point of X from its closest center in S .

It is somewhat illuminating to consider the problem in the plane. We have a set P of n points in the plane, we would like to find k smallest discs centered at input points, such that they cover all the points of P . Consider the example on the right.





In this example, assume that we would like to cover it by 3 disks. One possible solution is being shown in Figure 8.1. The quality of the solution is the radius r of the largest disk. As such, the clustering problem here can be interpreted as the problem of computing an optimal cover of the input point set by k disks/balls of minimum radius. This is known as the k -center problem.

It is known that k -center clustering is NP-HARD, even to approximate within a factor of (roughly) 1.8. Interestingly, there is a simple and elegant 2-approximation algorithm. Namely, one can compute in polynomial time, k centers, such that they induce balls of radius at most twice the optimal radius.

Here is the formal definition of the k -center clustering problem.

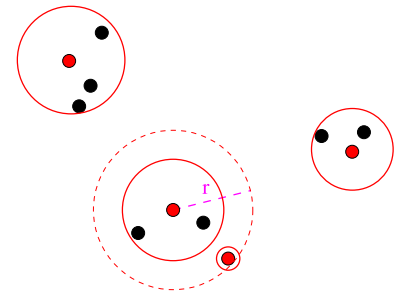
k -center clustering

Instance: A set P of n points, a distance function $\mathbf{d}(p, q)$, for $p, q \in P$, with triangle inequality holding for $\mathbf{d}(\cdot, \cdot)$, and a parameter k .

Question: A subset S that realizes $r_{opt}(P, k) = \min_{S \subseteq P, |S|=k} D_S(P)$, where $D_S(P) = \max_{x \in X} \mathbf{d}(x, S)$ and $\mathbf{d}(x, S) = \min_{s \in S} \mathbf{d}(s, x)$.

8.1.1. The approximation algorithm for k -center clustering

To come up with the idea behind the algorithm, imagine that we already have a solution with $m = 3$ centers. We would like to pick the next $m + 1$ center. Inspecting the examples above, one realizes that the solution is being determined by a bottleneck point; see Figure 8.1. That is, there is a single point which determine the quality of the clustering, which is the point furthest away from the set of centers. As such, the natural step is to find a new center that would better serve this bottleneck point. And, what can be a better service for this point, than make it the next center? (The resulting clustering using the new center for the example is depicted on the right.)



Namely, we always pick the bottleneck point, which is furthest away for the current set of centers, as the next center to be added to the solution.

The resulting approximation algorithm is depicted on the right. Observe, that the quantity r_{i+1} denotes the (minimum) radius of the i balls centered at u_1, \dots, u_i such that they cover P (where all these balls have the same radius). (Namely, there is a point $p \in P$ such that $\mathbf{d}(p, \{u_1, \dots, u_i\}) = r_{i+1}$.)

It would be convenient, for the sake analysis, to imagine that we run `AprxKCenter` one additional iteration, so that the quantity r_{k+1} is well defined.

Observe, that the running time of the algorithm `AprxKCenter` is $O(nk)$ as can be easily verified.

Lemma 8.1.1. *We have that $r_2 \geq \dots \geq r_k \geq r_{k+1}$.*

```

AprxKCenter( $P, k$ )
 $P = \{p_1, \dots, p_n\}$ 
 $S = \{p_1\}, u_1 \leftarrow p_1$ 
while  $|S| < k$  do
   $i \leftarrow |S|$ 
  for  $j = 1 \dots n$  do
     $d_j \leftarrow \min(d_j, \mathbf{d}(p_j, u_i))$ 
   $r_{i+1} \leftarrow \max(d_1, \dots, d_n)$ 
   $u_{i+1} \leftarrow$  point of  $P$  realizing  $r_i$ 
   $S \leftarrow S \cup \{u_{i+1}\}$ 

return  $S$ 

```

Proof: At each iteration the algorithm adds one new center, and as such the distance of a point to the closest center can not increase. In particular, the distance of the furthest point to the centers does not increase. ■

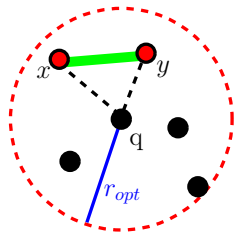
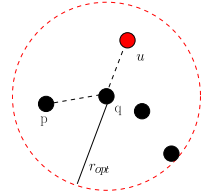
Observation 8.1.2. The radius of the clustering generated by *AprxKCenter* is r_{k+1} .

Lemma 8.1.3. We have that $r_{k+1} \leq 2r_{opt}(P, k)$, where $r_{opt}(P, k)$ is the radius of the optimal solution using k balls.

Proof: Consider the k balls forming the optimal solution: D_1, \dots, D_k and consider the k center points contained in the solution S computed by *AprxKCenter*.

If every disk D_i contain at least one point of S , then we are done, since every point of P is in distance at most $2r_{opt}(P, k)$ from one of the points of S . Indeed, if the ball D_i , centered at q , contains the point $u \in S$, then for any point $p \in P \cap D_i$, we have that

$$d(p, u) \leq d(p, q) + d(q, u) \leq 2r_{opt}.$$



Otherwise, there must be two points x and y of S contained in the same ball D_i of the optimal solution. Let D_i be centered at a point q .

We claim distance between x and y is at least r_{k+1} . Indeed, imagine that x was added at the α th iteration (that is, $u_\alpha = x$), and y was added in a later β th iteration (that is, $u_\beta = y$), where $\alpha < \beta$. Then,

$$r_\beta = d(y, \{u_1, \dots, u_{\beta-1}\}) \leq d(x, y),$$

since $x = u_\alpha$ and $y = u_\beta$. But $r_\beta \geq r_{k+1}$, by **Lemma 8.1.1**. Applying the triangle inequality again, we have that $r_{k+1} \leq r_\beta \leq d(x, y) \leq d(x, q) + d(q, y) \leq 2r_{opt}$, implying the claim. ■

Theorem 8.1.4. One can approximate the k -center clustering up to a factor of two, in time $O(nk)$.

Proof: The approximation algorithm is *AprxKCenter*. The approximation quality guarantee follows from **Lemma 8.1.3**, since the furthest point of P from the k -centers computed is r_{k+1} , which is guaranteed to be at most $2r_{opt}$. ■

8.2. Subset Sum

Subset Sum

Instance: $X = \{x_1, \dots, x_n\}$ – n integer positive numbers, t – target number

Question: Is there a subset of X such the sum of its elements is t ?

Subset Sum is (of course) **NPC**, as we already proved. It can be solved in polynomial time if the numbers of X are small. In particular, if $x_i \leq M$, for $i = 1, \dots, n$, then $t \leq Mn$ (otherwise, there is no solution). Its reasonably easy to solve in this case, as the algorithm on the right shows. The running time of the resulting algorithm is $O(Mn^2)$.

Note, that M might be prohibitly large, and as such, this algorithm is not polynomial in n . In particular, if $M = 2^n$ then this algorithm is prohibitly slow. Since the relevant decision problem is **NPC**, it is unlikely that an efficient algorithm exist for this problem. But still, we would like to be able to solve it quickly and efficiently. So, if we want an efficient solution, we would have to change the problem slightly. As a first step, lets turn it into an optimization problem.

SolveSubsetSum (X, t, M)

$b[0 \dots Mn]$ – boolean array init to **FALSE**.
 // $b[x]$ is **TRUE** if x can be realized by
 a subset of X .

$b[0] \leftarrow$ **TRUE**.

for $i = 1, \dots, n$ **do**

for $j = Mn$ down to x_i **do**

$b[j] \leftarrow B[j - x_i] \vee B[j]$

return $B[t]$

Subset Sum Optimization

Instance: (X, t) : A set X of n positive integers, and a target number t .

Question: The largest number γ_{opt} one can represent as a subset sum of X which is smaller or equal to t .

Intuitively, we would like to find a subset of X such that its sum is smaller than t but very close to t . Next, we turn the problem into an approximation problem.

Subset Sum Approx

Instance: (X, t, ε) : A set X of n positive integers, a target number t , and parameter $\varepsilon > 0$.

Question: A number z that one can represent as a subset sum of X , such that $(1 - \varepsilon)\gamma_{\text{opt}} \leq z \leq \gamma_{\text{opt}} \leq t$.

The challenge is to solve this approximation problem efficiently. To demonstrate that there is hope that can be done, consider the following simple approximation algorithm, that achieves a constant factor approximation.

Lemma 8.2.1. *Let (X, t) be an instance of **Subset Sum**. Let γ_{opt} be optimal solution to given instance. Then one can compute a subset sum that adds up to at least $\gamma_{\text{opt}}/2$ in $O(n \log n)$ time.*

Proof: Add the numbers from largest to smallest, whenever adding a number will make the sum exceed t , we throw it away. We claim that the generated sum s has the property that $\gamma_{\text{opt}}/2 \leq s \leq t$. Clearly, if the total sum of the numbers is smaller than t , then no number is being rejected and $s = \gamma_{\text{opt}}$.

Otherwise, let u be the first number being rejected, and let s' be the partial subset sum, just before u is being rejected. Clearly, $s' > u > 0$, $s' < t$, and $s' + u > t$. This implies $t < s' + u < s' + s' = 2s'$, which implies that $s' \geq t/2$. Namely, the subset sum output is larger than $t/2$. ■

8.2.1. On the complexity of ε -approximation algorithms

Definition 8.2.2 (PTAS). For a maximization problem **PROB**, an algorithm $\mathcal{A}(I, \varepsilon)$ (i.e., \mathcal{A} receives as input an instance of **PROB**, and an approximation parameter $\varepsilon > 0$) is a *polynomial time approximation scheme (PTAS)* if for any instance I we have

$$(1 - \varepsilon)|\text{opt}(I)| \leq |\mathcal{A}(I, \varepsilon)| \leq |\text{opt}(I)|,$$

where $|\text{opt}(I)|$ denote the price of the optimal solution for I , and $|\mathcal{A}(I, \varepsilon)|$ denotes the price of the solution outputted by \mathcal{A} . Furthermore, the running time of the algorithm \mathcal{A} is polynomial in n (the input size), when ε is fixed.

For a minimization problem, the condition is that $|\text{opt}(I)| \leq |\mathcal{A}(I, \varepsilon)| \leq (1 + \varepsilon)|\text{opt}(I)|$.

Example 8.2.3. An approximation algorithm with running time $O(n^{1/\varepsilon})$ is a PTAS, while an algorithm with running time $O(1/\varepsilon^n)$ is not.

Definition 8.2.4 (FPTAS). An approximation algorithm is *fully polynomial time approximation scheme (FPTAS)* if it is a PTAS, and its running time is polynomial both in n and $1/\varepsilon$.

Example 8.2.5. A PTAS with running time $O(n^{1/\varepsilon})$ is not a FPTAS, while a PTAS with running time $O(n^2/\varepsilon^3)$ is a FPTAS.

8.2.2. Approximating subset-sum

Let $S = \{a_1, \dots, a_n\}$ be a set of numbers. For a number x , let $x + S$ denote the translation of S by x ; namely, $x + S = \{a_1 + x, a_2 + x, \dots, a_n + x\}$. Our first step in deriving an approximation algorithm for **Subset Sum** is to come up with a slightly different algorithm for solving the problem exactly. The algorithm is depicted on the right.

Note, that while **ExactSubsetSum** performs only n iterations, the lists P_i that it constructs might have exponential size.

```

ExactSubsetSum( $S, t$ )
 $n \leftarrow |S|$ 
 $P_0 \leftarrow \{0\}$ 
for  $i = 1 \dots n$  do
     $P_i \leftarrow P_{i-1} \cup (P_{i-1} + x_i)$ 
    Remove from  $P_i$  all elements  $> t$ 

return largest element in  $P_n$ 

```

```

Trim( $L', \delta$ )
//  $L'$ : inc. sorted list of #s
 $L = \langle y_1 \dots y_m \rangle$ 
    //  $y_i \leq y_{i+1}$ , for  $i = 1, \dots, m-1$ .
 $curr \leftarrow y_1$ 
 $L_{out} \leftarrow \{y_1\}$ 
for  $i = 2 \dots m$  do
    if  $y_i > curr \cdot (1 + \delta)$ 
        Append  $y_i$  to  $L_{out}$ 
         $curr \leftarrow y_i$ 
return  $L_{out}$ 

```

Thus, if we would like to turn **ExactSubsetSum** into a faster algorithm, we need to somehow to make the lists L_i smaller. This would be done by removing numbers which are very close together.

Definition 8.2.6. For two positive real numbers $z \leq y$, the number y is a δ -approximation to z if $\frac{y}{1 + \delta} \leq z \leq y$.

The procedure **Trim** that trims a list L' so that it removes close numbers is depicted on the left.

Observation 8.2.7. If $x \in L'$ then there exists a number $y \in L_{out}$ such that $y \leq x \leq y(1 + \delta)$, where $L_{out} \leftarrow \text{Trim}(L', \delta)$.

We can now modify **ExactSubsetSum** to use **Trim** to keep the candidate list shorter. The resulting algorithm **ApproxSubsetSum** is depicted on the right. Note, that computing E_i requires merging two sorted lists, which can be done in linear time in the size of the lists (i.e., we can keep all the lists sorted, without sorting the lists repeatedly).

Let E_i be the list generated by the algorithm in the i th iteration, and P_i be the list of numbers without any trimming (i.e., the set generated by **ExactSubsetSum** algorithm) in the i th iteration.

```

ApproxSubsetSum( $S, t$ )
// Assume  $S = \{x_1, \dots, x_n\}$ , where
//  $x_1 \leq x_2 \leq \dots \leq x_n$ 
 $n \leftarrow |S|$ ,  $L_0 \leftarrow \{0\}$ ,  $\delta = \varepsilon/2n$ 
for  $i = 1 \dots n$  do
     $E_i \leftarrow L_{i-1} \cup (L_{i-1} + x_i)$ 
     $L_i \leftarrow \text{Trim}(E_i, \delta)$ 
    Remove from  $L_i$  all elements  $> t$ .

return largest element in  $L_n$ 

```

Claim 8.2.8. For any $x \in P_i$ there exists $y \in L_i$ such that $y \leq x \leq (1 + \delta)^i y$.

Proof: If $x \in P_1$ the claim follows by **Observation 8.2.7** above. Otherwise, if $x \in P_{i-1}$, then, by induction, there is $y' \in L_{i-1}$ such that $y' \leq x \leq (1 + \delta)^{i-1} y'$. **Observation 8.2.7** implies that there exists $y \in L_i$ such that $y \leq y' \leq (1 + \delta)y$. As such,

$$y \leq y' \leq x \leq (1 + \delta)^{i-1} y' \leq (1 + \delta)^i y$$

as required.

The other possibility is that $x \in P_i \setminus P_{i-1}$. But then $x = \alpha + x_i$, for some $\alpha \in P_{i-1}$. By induction, there exists $\alpha' \in L_{i-1}$ such that

$$\alpha' \leq \alpha \leq (1 + \delta)^{i-1} \alpha'.$$

Thus, $\alpha' + x_i \in E_i$ and by **Observation 8.2.7**, there is a $x' \in L_i$ such that

$$x' \leq \alpha' + x_i \leq (1 + \delta)x'.$$

Thus,

$$x' \leq \alpha' + x_i \leq \alpha + x_i = x \leq (1 + \delta)^{i-1} \alpha' + x_i \leq (1 + \delta)^{i-1} (\alpha' + x_i) \leq (1 + \delta)^i x'.$$

Namely, for any $x \in P_i \setminus P_{i-1}$, there exists $x' \in L_i$, such that $x' \leq x \leq (1 + \delta)^i x'$. ■

8.2.2.1. Bounding the running time of **ApproxSubsetSum**

We need the following two easy technical lemmas. We include their proofs here only for the sake of completeness.

Lemma 8.2.9. *For $x \in [0, 1]$, it holds $\exp(x/2) \leq (1 + x)$.*

Proof: Let $f(x) = \exp(x/2)$ and $g(x) = 1 + x$. We have $f'(x) = \exp(x/2)/2$ and $g'(x) = 1$. As such,

$$f'(x) = \frac{\exp(x/2)}{2} \leq \frac{\exp(1/2)}{2} \leq 1 = g'(x), \quad \text{for } x \in [0, 1].$$

Now, $f(0) = g(0) = 1$, which immediately implies the claim. ■

Lemma 8.2.10. *For $0 < \delta < 1$, and $x \geq 1$, we have $\log_{1+\delta} x \leq \frac{2 \ln x}{\delta} = O\left(\frac{\ln x}{\delta}\right)$.*

Proof: We have, by **Lemma 8.2.9**, that $\log_{1+\delta} x = \frac{\ln x}{\ln(1 + \delta)} \leq \frac{\ln x}{\ln \exp(\delta/2)} = \frac{2 \ln x}{\delta}$. ■

Observation 8.2.11. *In a list generated by **Trim**, for any number x , there are no two numbers in the trimmed list between x and $(1 + \delta)x$.*

Lemma 8.2.12. *We have $|L_i| = O\left(\frac{n^2}{\varepsilon} \log n\right)$, for $i = 1, \dots, n$.*

Proof: The set $L_{i-1} + x_i$ is a set of numbers between x_i and ix_i , because x_i is larger than $x_1 \dots x_{i-1}$ and L_{i-1} contains subset sums of at most $i - 1$ numbers, each one of them smaller than x_i . As such, the number of different values in this range, stored in the list L_i , after trimming is at most

$$\log_{1+\delta} \frac{ix_i}{x_i} = O\left(\frac{\ln i}{\delta}\right) = O\left(\frac{\ln n}{\delta}\right),$$

by **Lemma 8.2.10**. Thus, as $\delta = \varepsilon/2n$, we have

$$|L_i| \leq |L_{i-1}| + O\left(\frac{\ln n}{\delta}\right) \leq |L_{i-1}| + O\left(\frac{n \ln n}{\varepsilon}\right) = O\left(\frac{n^2 \log n}{\varepsilon}\right). \quad \blacksquare$$

Lemma 8.2.13. *The running time of **ApproxSubsetSum** is $O\left(\frac{n^3}{\varepsilon} \log^2 n\right)$.*

Proof: Clearly, the running time of **ApproxSubsetSum** is dominated by the total length of the lists L_1, \dots, L_n it creates. **Lemma 8.2.12** implies that $\sum_i |L_i| = O\left(\frac{n^3}{\varepsilon} \log n\right)$. The running time of **Trim** is proportional to the size of the lists, implying the claimed running time. ■

8.2.2.2. The result

Theorem 8.2.14. *ApproxSubsetSum* returns a number $u \leq t$, such that

$$\frac{\gamma_{\text{opt}}}{1 + \varepsilon} \leq u \leq \gamma_{\text{opt}} \leq t,$$

where γ_{opt} is the optimal solution (i.e., largest realizable subset sum smaller than t).

The running time of *ApproxSubsetSum* is $O((n^3/\varepsilon) \log n)$.

Proof: The running time bound is by [Lemma 8.2.13](#).

As for the other claim, consider the optimal solution $\text{opt} \in P_n$. By [Claim 8.2.8](#), there exists $z \in L_n$ such that $z \leq \text{opt} \leq (1 + \delta)^n z$. However,

$$(1 + \delta)^n = (1 + \varepsilon/2n)^n \leq \exp\left(\frac{\varepsilon}{2}\right) \leq 1 + \varepsilon,$$

since $1 + x \leq e^x$ for $x \geq 0$. Thus, $\text{opt}/(1 + \varepsilon) \leq z \leq \text{opt} \leq t$, implying that the output of *ApproxSubsetSum* is within the required range. ■

8.3. Approximate Bin Packing

Consider the following problem.

Min Bin Packing

Instance: $s_1 \dots s_n - n$ numbers in $[0, 1]$

Question: Q: What is the minimum number of unit bins do you need to use to store all the numbers in S ?

Bin Packing is **NP-COMplete** because you can reduce **Partition** to it. Its natural to ask how one can approximate the optimal solution to **Bin Packing**.

One such algorithm is **next fit**. Here, we go over the numbers one by one, and put a number in the current bin if that bin can contain it. Otherwise, we create a new bin and put the number in this bin. Clearly, we need at least

$$\lceil S \rceil \text{ bins where } S = \sum_{i=1}^n s_i.$$

Every two consecutive bins contain numbers that add up to more than 1, since otherwise we would have not created the second bin. As such, the number of bins used is $\leq 2 \lceil S \rceil$. As such, the next fit algorithm for bin packing achieves a $\leq 2 \lceil S \rceil / \lceil S \rceil = 2$ approximation.

A better strategy, is to sort the numbers from largest to smallest and insert them in this order, where in each stage, we scan all current bins, and see if can insert the current number into one of those bins. If we can not, we create a new bin for this number. This is known as **first fit decreasing**. We state the approximation ratio for this algorithm without proof.

Theorem 8.3.1 ([DLHT13]). *Decreasing first fit is a 11/9-approximation to Min Bin Packing. More precisely, for any instance I of the problem, one has*

$$\text{FFD}(I) \leq \frac{11}{9} \text{opt}(I) + \frac{2}{3},$$

and this is tight in the worst case. Here $\text{FFD}(I)$ and $\text{opt}(I)$ are the number of bins used by the first-fit decreasing algorithm and optimal solution, respectively.

Remark 8.3.2. Note that if $\text{opt}(I) = 2$, then the above bound is $\text{FFD}(I) \leq \frac{11}{9} \cdot 2 + \frac{2}{3} = \frac{28}{9} = 3\frac{1}{9}$. Which means that in this case this approach could yield a solution with three bins, which is not exciting.

The above paper is almost 50 pages long, and is not easy. The coefficient $11/9$ was proved by David S. Johnson in his PhD thesis in 1973 (who also authored [GJ90]), but the exact value of the additive constant was only settled by [DLHT13].

Remark 8.3.3. Note, that the above algorithm is not a multiplicative approximation (note the $+2/3$ term). In particular, getting a $3/2$ -approximation is hard because of the reduction from **Partition** – there the decision boils down to whether the instance generated from partition requires two bins or three bins. As such, any multiplicative approximation better than $3/2$ is impossible unless $P = NP$.

8.4. Bibliographical notes

One can do 2-approximation for the k -center clustering in low dimensional Euclidean space can be done in $\Theta(n \log k)$ time [FG88]. In fact, it can be solved in linear time [Har04].

Part IV

Randomized algorithms

Chapter 9

Randomized Algorithms

9.1. Some Probability

Definition 9.1.1. (Informal.) A *random variable* is a measurable function from a probability space to (usually) real numbers. It associates a value with each possible atomic event in the probability space.

Definition 9.1.2. The *conditional probability* of X given Y is

$$\mathbb{P}[X = x | Y = y] = \frac{\mathbb{P}[(X = x) \cap (Y = y)]}{\mathbb{P}[Y = y]}.$$

An equivalent and useful restatement of this is that

$$\mathbb{P}[(X = x) \cap (Y = y)] = \mathbb{P}[X = x | Y = y] * \mathbb{P}[Y = y].$$

Definition 9.1.3. Two events X and Y are *independent*, if $\mathbb{P}[X = x \cap Y = y] = \mathbb{P}[X = x] \cdot \mathbb{P}[Y = y]$. In particular, if X and Y are independent, then

$$\mathbb{P}[X = x | Y = y] = \mathbb{P}[X = x].$$

Definition 9.1.4. The *expectation* of a random variable X is the average value of this random variable. Formally, if X has a finite (or countable) set of values, it is

$$\mathbb{E}[X] = \sum_x x \cdot \mathbb{P}[X = x],$$

where the summation goes over all the possible values of X .

One of the most powerful properties of expectations is that an expectation of a sum is the sum of expectations.

Lemma 9.1.5 (Linearity of expectation.). For any two random variables X and Y , we have $\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]$.

Proof: For the simplicity of exposition, assume that X and Y receive only integer values. We have that

$$\begin{aligned}
\mathbb{E}[X + Y] &= \sum_x \sum_y (x + y) \mathbb{P}[(X = x) \cap (Y = y)] \\
&= \sum_x \sum_y x * \mathbb{P}[(X = x) \cap (Y = y)] + \sum_x \sum_y y * \mathbb{P}[(X = x) \cap (Y = y)] \\
&= \sum_x x * \sum_y \mathbb{P}[(X = x) \cap (Y = y)] + \sum_y y * \sum_x \mathbb{P}[(X = x) \cap (Y = y)] \\
&= \sum_x x * \mathbb{P}[X = x] + \sum_y y * \mathbb{P}[Y = y] \\
&= \mathbb{E}[X] + \mathbb{E}[Y]. \quad \blacksquare
\end{aligned}$$

Another interesting function is the conditional expectation – that is, it is the expectation of a random variable given some additional information.

Definition 9.1.6. Given random variables X and Y , the *conditional expectation* of X given Y , is the quantity $\mathbb{E}[X | Y]$. Specifically, you are given the value y of the random variable Y , and the $\mathbb{E}[X | Y] = \mathbb{E}[X | Y = y] = \sum_x x * \mathbb{P}[X = x | Y = y]$.

Note, that for a random variable X , the expectation $\mathbb{E}[X]$ is a number. On the other hand, the conditional probability $f(y) = \mathbb{E}[X | Y = y]$ is a function. The key insight why conditional probability is the following.

Lemma 9.1.7. For any two random variables X and Y (not necessarily independent), we have that $\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X | Y]]$.

Proof: We use the definitions carefully:

$$\begin{aligned}
\mathbb{E}[\mathbb{E}[X | Y]] &= \mathbb{E}_y[\mathbb{E}[X | Y = y]] = \mathbb{E}_y\left[\sum_x x * \mathbb{P}[X = x | Y = y]\right] \\
&= \sum_y \mathbb{P}[Y = y] * \left(\sum_x x * \mathbb{P}[X = x | Y = y]\right) \\
&= \sum_y \mathbb{P}[Y = y] * \left(\sum_x x * \frac{\mathbb{P}[(X = x) \cap (Y = y)]}{\mathbb{P}[Y = y]}\right) \\
&= \sum_y \sum_x x * \mathbb{P}[(X = x) \cap (Y = y)] = \sum_x \sum_y x * \mathbb{P}[(X = x) \cap (Y = y)] \\
&= \sum_x x * \left(\sum_y \mathbb{P}[(X = x) \cap (Y = y)]\right) = \sum_x x * \mathbb{P}[X = x] = \mathbb{E}[X]. \quad \blacksquare
\end{aligned}$$

9.2. Sorting Nuts and Bolts

Problem 9.2.1 (Sorting Nuts and Bolts). You are given a set of n nuts and n bolts. Every nut have a matching bolt, and all the n pairs of nuts and bolts have different sizes. Unfortunately, you get the nuts and bolts separated from each other and you have to match the nuts to the bolts. Furthermore, given a nut and a bolt, all you can do is to try and match one bolt against a nut (i.e., you can not compare two nuts to each other, or two bolts to each other).

When comparing a nut to a bolt, either they match, or one is smaller than other (and you know the relationship after the comparison).

How to match the n nuts to the n bolts quickly? Namely, while performing a small number of comparisons.

The naive algorithm is of course to compare each nut to each bolt, and match them together. This would require a quadratic number of comparisons. Another option is to sort the nuts by size, and the bolts by size and then “merge” the two ordered sets, matching them by size. The only problem is that we can not sort only the nuts, or only the bolts, since we can not compare them to each other. Indeed, we sort the two sets simultaneously, by simulating **QuickSort**. The resulting algorithm is depicted on the right.

```

MatchNutsAndBolts( $N$ : nuts,  $B$ : bolts)
  Pick a random nut  $n_{pivot}$  from  $N$ 
  Find its matching bolt  $b_{pivot}$  in  $B$ 
   $B_L \leftarrow$  All bolts in  $B$  smaller than  $n_{pivot}$ 
   $N_L \leftarrow$  All nuts in  $N$  smaller than  $b_{pivot}$ 
   $B_R \leftarrow$  All bolts in  $B$  larger than  $n_{pivot}$ 
   $N_R \leftarrow$  All nuts in  $N$  larger than  $b_{pivot}$ 
  MatchNutsAndBolts( $N_R, B_R$ )
  MatchNutsAndBolts( $N_L, B_L$ )
  
```

9.2.1. Running time analysis

Definition 9.2.2. Let \mathcal{RT} denote the random variable which is the running time of the algorithm. Note, that the running time is a random variable as it might be different between different executions on the *same* input.

Definition 9.2.3. For a randomized algorithm, we can speak about the expected running time. Namely, we are interested in bounding the quantity $\mathbb{E}[\mathcal{RT}]$ for the worst input.

Definition 9.2.4. The *expected running-time* of a randomized algorithm for input of size n is

$$T(n) = \max_{U \text{ is an input of size } n} \mathbb{E}[\mathcal{RT}(U)],$$

where $\mathcal{RT}(U)$ is the running time of the algorithm for the input U .

Definition 9.2.5. The *rank* of an element x in a set S , denoted by $\text{rank}(x)$, is the number of elements in S of size smaller or equal to x . Namely, it is the location of x in the sorted list of the elements of S .

Theorem 9.2.6. *The expected running time of **MatchNutsAndBolts** (and thus also of **QuickSort**) is $T(n) = O(n \log n)$, where n is the number of nuts and bolts. The worst case running time of this algorithm is $O(n^2)$.*

Proof: Clearly, we have that $\mathbb{P}[\text{rank}(n_{pivot}) = k] = \frac{1}{n}$. Furthermore, if the rank of the pivot is k then

$$\begin{aligned} T(n) &= \mathbb{E}_{k=\text{rank}(n_{pivot})} [O(n) + T(k-1) + T(n-k)] = O(n) + \mathbb{E}_k [T(k-1) + T(n-k)] \\ &= T(n) = O(n) + \sum_{k=1}^n \mathbb{P}[\text{Rank}(\text{Pivot}) = k] \cdot (T(k-1) + T(n-k)) \\ &= O(n) + \sum_{k=1}^n \frac{1}{n} \cdot (T(k-1) + T(n-k)), \end{aligned}$$

by the definition of expectation. It is not easy to verify that the solution to the recurrence $T(n) = O(n) + \sum_{k=1}^n \frac{1}{n} \cdot (T(k-1) + T(n-k))$ is $O(n \log n)$. ■

9.2.1.1. Alternative incorrect solution

The algorithm `MatchNutsAndBolts` is lucky if $\frac{n}{4} \leq \text{rank}(n_{pivot}) \leq \frac{3}{4}n$. Thus, $\mathbb{P}[\text{“lucky”}] = 1/2$. Intuitively, for the algorithm to be fast, we want the split to be as balanced as possible. The less balanced the cut is, the worst the expected running time. As such, the “Worst” lucky position is when $\text{rank}(n_{pivot}) = n/4$ and we have that

$$T(n) \leq O(n) + \mathbb{P}[\text{“lucky”}] * (T(n/4) + T(3n/4)) + \mathbb{P}[\text{“unlucky”}] * T(n).$$

Namely, $T(n) = O(n) + \frac{1}{2} * (T(\frac{n}{4}) + T(\frac{3}{4}n)) + \frac{1}{2}T(n)$. Rewriting, we get the recurrence $T(n) = O(n) + T(n/4) + T((3/4)n)$, and its solution is $O(n \log n)$.

While this is a very intuitive and elegant solution that bounds the running time of `QuickSort`, it is also incomplete. The interested reader should try and make this argument complete. After completion the argument is as involved as the previous argument. Nevertheless, this argumentation gives a good back of the envelope analysis for randomized algorithms which can be applied in a lot of cases.

9.2.2. What are randomized algorithms?

Randomized algorithms are algorithms that use random numbers (retrieved usually from some unbiased source of randomness [say a library function that returns the result of a random coin flip]) to make decisions during the executions of the algorithm. The running time becomes a random variable. Analyzing the algorithm would now boil down to analyzing the behavior of the random variable $\mathcal{RT}(n)$, where n denotes the size of the input. In particular, the expected running time $\mathbb{E}[\mathcal{RT}(n)]$ is a quantity that we would be interested in.

It is useful to compare the expected running time of a randomized algorithm, which is

$$T(n) = \max_{U \text{ is an input of size } n} \mathbb{E}[\mathcal{RT}(U)],$$

to the worst case running time of a deterministic (i.e., not randomized) algorithm, which is

$$T(n) = \max_{U \text{ is an input of size } n} \mathcal{RT}(U),$$

Caveat Emptor:^①Note, that a randomized algorithm might have exponential running time in the worst case (or even unbounded) while having good expected running time. For example, consider the algorithm `FlipCoins` depicted on the right. The expected running time of `FlipCoins` is a geometric random variable with probability 1/2, as such we have that $\mathbb{E}[\mathcal{RT}(\text{FlipCoins})] = O(2)$. However, `FlipCoins` can run forever if it always gets 1 from the `RandBit` function.

```
FlipCoins
while RandBit = 1 do
    nothing;
```

This is of course a ludicrous argument. Indeed, the probability that `FlipCoins` runs for long decreases very quickly as the number of steps increases. It can happen that it runs for long, but it is extremely unlikely.

Definition 9.2.7. The running time of a randomized algorithm `Alg` is $O(f(n))$ with *high probability* if

$$\mathbb{P}[\mathcal{RT}(\text{Alg}(n)) \geq c \cdot f(n)] = o(1).$$

Namely, the probability of the algorithm to take more than $O(f(n))$ time decreases to 0 as n goes to infinity. In our discussion, we would use the following (considerably more restrictive definition), that requires that

$$\mathbb{P}[\mathcal{RT}(\text{Alg}(n)) \geq c \cdot f(n)] \leq \frac{1}{n^d},$$

where c and d are appropriate constants. For technical reasons, we also require that $\mathbb{E}[\mathcal{RT}(\text{Alg}(n))] = O(f(n))$.

^①Caveat Emptor - let the buyer beware (i.e., one buys at one’s own risk)

9.3. Analyzing QuickSort

The previous analysis works also for **QuickSort**. However, there is an alternative analysis which is also very interesting and elegant. Let a_1, \dots, a_n be the n given numbers (in sorted order – as they appear in the output).

It is enough to bound the number of comparisons performed by **QuickSort** to bound its running time, as can be easily verified. Observe, that two specific elements are compared to each other by **QuickSort** at most once, because **QuickSort** performs only comparisons against the pivot, and after the comparison happen, the pivot does not being passed to the two recursive subproblems.

Let X_{ij} be an indicator variable if **QuickSort** compared a_i to a_j in the current execution, and zero otherwise. The number of comparisons performed by **QuickSort** is **exactly** $Z = \sum_{i < j} X_{ij}$.

Observation 9.3.1. *The element a_i is compared to a_j iff one of them is picked to be the pivot and they are still in the same subproblem.*

Also, we have that $\mu = \mathbb{E}[X_{ij}] = \mathbb{P}[X_{ij} = 1]$. To quantify this probability, observe that if the pivot is smaller than a_i or larger than a_j then the subproblem still contains the block of elements a_i, \dots, a_j . Thus, we have that

$$\mu = \mathbb{P}[a_i \text{ or } a_j \text{ is first pivot} \in a_i, \dots, a_j] = \frac{2}{j - i + 1}.$$

Another (and hopefully more intuitive) explanation for the above phenomena is the following: Imagine, that before running **QuickSort** we choose for every element a random priority, which is a real number in the range $[0, 1]$. Now, we reimplement **QuickSort** such that it always pick the element with the lowest random priority (in the given subproblem) to be the pivot. One can verify that this variant and the standard implementation have the same running time. Now, a_i gets compares to a_j if and only if all the elements a_{i+1}, \dots, a_{j-1} have random priority larger than both the random priority of a_i and the random priority of a_j . But the probability that one of two elements would have the lowest random-priority out of $j - i + 1$ elements is $2 * 1/(j - i + 1)$, as claimed.

Thus, the running time of **QuickSort** is

$$\begin{aligned} \mathbb{E}[\mathcal{RT}(n)] &= \mathbb{E}\left[\sum_{i < j} X_{ij}\right] = \sum_{i < j} \mathbb{E}[X_{ij}] = \sum_{i < j} \frac{2}{j - i + 1} = 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{1}{j - i + 1} \\ &= 2 \sum_{i=1}^{n-1} \sum_{\Delta=2}^{n-i+1} \frac{1}{\Delta} \leq 2 \sum_{i=1}^{n-1} \sum_{\Delta=1}^n \frac{1}{\Delta} \leq 2 \sum_{i=1}^{n-1} H_n = 2nH_n. \end{aligned}$$

by linearity of expectations, where $H_n = \sum_{i=1}^n \frac{1}{i} \leq \ln n + 1$ is the n th harmonic number,

As we will see in the near future, the running time of **QuickSort** is $O(n \log n)$ with high-probability. We need some more tools before we can show that.

9.4. QuickSelect – median selection in linear time

Consider the problem of given a set X of n numbers, and a parameter k , to output the k th smallest number (which is the number with **rank** k in X). This can be easily be done by modifying **QuickSort** only to perform one recursive call. See **Figure 9.1** for a pseud-code of the resulting algorithm.

Intuitively, at each iteration of **QuickSelect** the input size shrinks by a constant factor, leading to a linear time algorithm.

Theorem 9.4.1. *Given a set X of n numbers, and any integer k , the expected running time of **QuickSelect**(X, n) is $O(n)$.*

```

QuickSelect( $X, k$ )
// Input:  $X = \{x_1, \dots, x_n\}$  numbers,  $k$ .
// Assume  $x_1, \dots, x_n$  are all distinct.
// Task: Return  $k$ th smallest number in  $X$ .
 $y \leftarrow$  random element of  $X$ .
 $r \leftarrow$  rank of  $y$  in  $X$ .
if  $r = k$  then return  $y$ 
 $X_{<} =$  all elements in  $X <$  than  $y$ 
 $X_{>} =$  all elements in  $X >$  than  $y$ 
// By assumption  $|X_{<}| + |X_{>}| + 1 = |X|$ .
if  $r < k$  then
    return QuickSelect( $X_{>}, k - r$ )
else
    return QuickSelect( $X_{\leq}, k$ )

```

Figure 9.1: QuickSelect pseudo-code.

Proof: Let $X_1 = X$, and X_i be the set of numbers in the i th level of the recursion. Let y_i and r_i be the random element and its rank in X_i , respectively, in the i th iteration of the algorithm. Finally, let $n_i = |X_i|$. Observe that the probability that the pivot y_i is in the “middle” of its subproblem is

$$\alpha = \mathbb{P}\left[\frac{n_i}{4} \leq r_i \leq \frac{3}{4}n_i\right] \geq \frac{1}{2},$$

and if this happens then

$$n_{i+1} \leq \max(r_i - 1, n_i - r_i) \leq \frac{3}{4}n_i.$$

We conclude that

$$\begin{aligned} \mathbb{E}[n_{i+1} \mid n_i] &\leq \mathbb{P}[y_i \text{ in the middle}] \frac{3}{4}n_i + \mathbb{P}[y_i \text{ not in the middle}]n_i \\ &\leq \alpha \frac{3}{4}n_i + (1 - \alpha)n_i = n_i(1 - \alpha/4) \leq n_i(1 - (1/2)/4) = (7/8)n_i. \end{aligned}$$

Now, we have that

$$\begin{aligned} m_{i+1} = \mathbb{E}[n_{i+1}] &= \mathbb{E}\left[\mathbb{E}[n_{i+1} \mid n_i]\right] \leq \mathbb{E}[(7/8)n_i] = (7/8)\mathbb{E}[n_i] = (7/8)m_i \\ &= (7/8)^i m_0 = (7/8)^i n, \end{aligned}$$

since for any two random variables we have that $\mathbb{E}[X] = \mathbb{E}[\mathbb{E}[X \mid Y]]$. In particular, the expected running time of QuickSelect is proportional to

$$\mathbb{E}\left[\sum_i n_i\right] = \sum_i \mathbb{E}[n_i] \leq \sum_i m_i = \sum_i (7/8)^i n = O(n),$$

as desired. ■

Chapter 10

Randomized Algorithms II

10.1. QuickSort and Treaps with High Probability

You must be asking yourself what are treaps. For the answer, see [Section 10.3_{p75}](#).

One can think about **QuickSort** as playing a game in rounds. Every round, **QuickSort** picks a pivot, splits the problem into two subproblems, and continue playing the game recursively on both subproblems.

If we track a single element in the input, we see a sequence of rounds that involve this element. The game ends, when this element find itself alone in the round (i.e., the subproblem is to sort a single element).

Thus, to show that **QuickSort** takes $O(n \log n)$ time, it is enough to show, that every element in the input, participates in at most $32 \ln n$ rounds with high enough probability.

Indeed, let X_i be the event that the i th element participates in more than $32 \ln n$ rounds.

Let C_{QS} be the number of comparisons performed by **QuickSort**. A comparison between a pivot and an element will be always charged to the element. And as such, the number of comparisons overall performed by **QuickSort** is bounded by $\sum_i r_i$, where r_i is the number of rounds the i th element participated in (the last round where it was a pivot is ignored). We have that

$$\alpha = \mathbb{P}[C_{QS} \geq 32n \ln n] \leq \mathbb{P}\left[\bigcup_i X_i\right] \leq \sum_{i=1}^n \mathbb{P}[X_i].$$

Here, we used the **union bound**^①, that states that for any two events A and B , we have that $\mathbb{P}[A \cup B] \leq \mathbb{P}[A] + \mathbb{P}[B]$. Assume, for the time being, that $\mathbb{P}[X_i] \leq 1/n^3$. This implies that

$$\alpha \leq \sum_{i=1}^n \mathbb{P}[X_i] \leq \sum_{i=1}^n \frac{1}{n^3} = \frac{1}{n^2}.$$

Namely, **QuickSort** performs at most $32n \ln n$ comparisons with high probability. It follows, that **QuickSort** runs in $O(n \log n)$ time, with high probability, since the running time of **QuickSort** is proportional to the number of comparisons it performs.

To this end, we need to prove that $\mathbb{P}[X_i] \leq 1/n^3$.

10.1.1. Proving that an element participates in small number of rounds

Consider a run of **QuickSort** for an input made out of n numbers. Consider a specific element x in this input, and let S_1, S_2, \dots be the subsets of the input that are in the recursive calls that include the element x . Here S_j is the set of numbers in the j th round (i.e., this is the recursive call at depth j which includes x among the numbers it needs to sort).

The element x would be considered to be **lucky**, in the j th iteration, if the call to the **QuickSort**, splits the current set S_j into two parts, where both parts contains at most $(3/4)|S_j|$ of the elements.

Let Y_j be an indicator variable which is 1 if and only if x is lucky in j th round. Formally, $Y_j = 1$ if and only if $|S_j|/4 \leq |S_{j+1}| \leq 3|S_j|/4$. By definition, we have that

$$\mathbb{P}[Y_j] = \frac{1}{2}.$$

^①Also known as Boole's inequality.

Furthermore, Y_1, Y_2, \dots, Y_m are all independent variables.

Note, that x can participate in at most

$$\rho = \log_{4/3} n \leq 3.5 \ln n \tag{10.1}$$

rounds, since at each successful round, the number of elements in the subproblem shrinks by at least a factor $3/4$, and $|S_1| = n$. As such, if there are ρ successful rounds in the first k rounds, then $|S_k| \leq (3/4)^\rho n \leq 1$.

Thus, the question of how many rounds x participates in, boils down to how many coin flips one needs to perform till one gets ρ heads. Of course, in expectation, we need to do this 2ρ times. But what if we want a bound that holds with high probability, how many rounds are needed then?

In the following, we require the following lemma, which we will prove in [Section 10.2](#).

Lemma 10.1.1. *In a sequence of M coin flips, the probability that the number of ones is smaller than $L \leq M/4$ is at most $\exp(-M/8)$.*

To use [Lemma 10.1.1](#), we set

$$M = 32 \ln n \geq 8\rho,$$

see [Eq. \(10.1\)](#). Let Y_j be the variable which is one if x is lucky in the j th level of recursion, and zero otherwise. We have that $\mathbb{P}[Y_j = 0] = \mathbb{P}[Y_j = 1] = 1/2$ and that Y_1, Y_2, \dots, Y_M are independent. By [Lemma 10.1.1](#), we have that the probability that there are only $\rho \leq M/4$ ones in Y_1, \dots, Y_M , is smaller than

$$\exp\left(-\frac{M}{8}\right) \leq \exp(-\rho) \leq \frac{1}{n^3}.$$

We have that the probability that x participates in M recursive calls of [QuickSort](#) to be at most $1/n^3$.

There are n input elements. Thus, the probability that depth of the recursion in [QuickSort](#) exceeds $32 \ln n$ is smaller than $(1/n^3) * n = 1/n^2$. We thus established the following result.

Theorem 10.1.2. *With high probability (i.e., $1 - 1/n^2$) the depth of the recursion of [QuickSort](#) is $\leq 32 \ln n$. Thus, with high probability, the running time of [QuickSort](#) is $O(n \log n)$.*

More generally, for any constant c , there exist a constant d , such that the probability that [QuickSort](#) recursion depth for any element exceeds $d \ln n$ is smaller than $1/n^c$.

Specifically, for any $t \geq 1$, we have that probability that the recursion depth for any element exceeds $t \cdot d \ln n$ is smaller than $1/n^{t^c}$.

Proof: Let us do the last part (but the reader is encouraged to skip this on first reading). Setting $M = 32t \ln n$, we get that the probability that an element has depth exceeds M , requires that in M coin flips we get at most $h = 4 \ln n$ heads. That is, if Y is the sum of the coin flips, where we get $+1$ for head, and -1 for tails, then Y needs to be smaller than $-(M - h) + h = -M + 2h$. By symmetry, this is equal to the probability that $Y \geq \Delta = M - 2h$. By [Theorem 10.2.3](#) below, the probability for that is

$$\begin{aligned} \mathbb{P}[Y \geq \Delta] &\leq \exp\left(-\Delta^2/2M\right) = \exp\left(-\frac{(M - 2h)^2}{2M}\right) = \exp\left(-\frac{(32t - 8)^2 \ln^2 n}{128t \ln n}\right) \\ &= \exp\left(-\frac{(4t - 1)^2 \ln n}{2t}\right) \leq \exp\left(-\frac{3t^2 \ln n}{t}\right) \leq \frac{1}{n^{3t}}. \quad \blacksquare \end{aligned}$$

Of course, the same result holds for the algorithm [MatchNutsAndBolts](#) for matching nuts and bolts.

10.1.2. An alternative proof of the high probability of QuickSort

Consider a set T of the n items to be sorted, and consider a specific element $t \in T$. Let X_i be the size of the input in the i th level of recursion that contains t . We know that $X_0 = n$, and

$$\mathbb{E}[X_i \mid X_{i-1}] \leq \frac{1}{2} \frac{3}{4} X_{i-1} + \frac{1}{2} X_{i-1} \leq \frac{7}{8} X_{i-1}.$$

Indeed, with probability $1/2$ the pivot is the middle of the subproblem; that is, its rank is between $X_{i-1}/4$ and $(3/4)X_{i-1}$ (and then the subproblem has size $\leq X_{i-1}(3/4)$), and with probability $1/2$ the subproblem might not shrink significantly (i.e., we pretend it did not shrink at all).

Now, observe that for any two random variables we have that $\mathbb{E}[X] = \mathbb{E}_y[\mathbb{E}[X \mid Y = y]]$, see [Lemma 9.1.7](#)_{p65}.. As such, we have that

$$\mathbb{E}[X_i] = \mathbb{E}_y[\mathbb{E}[X_i \mid X_{i-1} = y]] \leq \mathbb{E}_{X_{i-1}=y} \left[\frac{7}{8} y \right] = \frac{7}{8} \mathbb{E}[X_{i-1}] \leq \left(\frac{7}{8} \right)^i \mathbb{E}[X_0] = \left(\frac{7}{8} \right)^i n.$$

In particular, consider $M = 8 \log_{8/7} n$. We have that

$$\mu = \mathbb{E}[X_M] \leq \left(\frac{7}{8} \right)^M n \leq \frac{1}{n^8} n = \frac{1}{n^7}.$$

Of course, t participates in more than M recursive calls, if and only if $X_M \geq 1$. However, by Markov's inequality ([Theorem 10.2.1](#)), we have that

$$\mathbb{P} \left[\begin{array}{c} \text{element } t \text{ participates} \\ \text{in more than } M \text{ recursive calls} \end{array} \right] \leq \mathbb{P}[X_M \geq 1] \leq \frac{\mathbb{E}[X_M]}{1} \leq \frac{1}{n^7},$$

as desired. That is, we proved that the probability that any element of the input T participates in more than M recursive calls is at most $n(1/n^7) \leq 1/n^6$.

10.2. Chernoff inequality

10.2.1. Preliminaries

Theorem 10.2.1 (*Markov's Inequality*). For a non-negative variable X , and $t > 0$, we have:

$$\mathbb{P}[X \geq t] \leq \frac{\mathbb{E}[X]}{t}.$$

Proof: Assume that this is false, and there exists $t_0 > 0$ such that $\mathbb{P}[X \geq t_0] > \frac{\mathbb{E}[X]}{t_0}$. However,

$$\begin{aligned} \mathbb{E}[X] &= \sum_x x \cdot \mathbb{P}[X = x] = \sum_{x < t_0} x \cdot \mathbb{P}[X = x] + \sum_{x \geq t_0} x \cdot \mathbb{P}[X = x] \\ &\geq 0 + t_0 \cdot \mathbb{P}[X \geq t_0] > 0 + t_0 \cdot \frac{\mathbb{E}[X]}{t_0} = \mathbb{E}[X], \end{aligned}$$

a contradiction. ■

We remind the reader that two random variables X and Y are *independent* if for all x, y we have that

$$\mathbb{P}[(X = x) \cap (Y = y)] = \mathbb{P}[X = x] \cdot \mathbb{P}[Y = y].$$

The following claim is easy to verify, and we omit the easy proof.

Claim 10.2.2. If X and Y are independent, then $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$.

If X and Y are independent then $Z = e^X$ and $W = e^Y$ are also independent variables.

10.2.2. Chernoff inequality

Theorem 10.2.3 (Chernoff inequality). Let X_1, \dots, X_n be n independent random variables, such that $\mathbb{P}[X_i = 1] = \mathbb{P}[X_i = -1] = \frac{1}{2}$, for $i = 1, \dots, n$. Let $Y = \sum_{i=1}^n X_i$. Then, for any $\Delta > 0$, we have

$$\mathbb{P}[Y \geq \Delta] \leq \exp(-\Delta^2/2n).$$

Proof: Clearly, for an arbitrary t , to be specified shortly, we have

$$\mathbb{P}[Y \geq \Delta] = \mathbb{P}[tY \geq t\Delta] = \mathbb{P}[\exp(tY) \geq \exp(t\Delta)] \leq \frac{\mathbb{E}[\exp(tY)]}{\exp(t\Delta)}, \quad (10.2)$$

where the first part follows since $\exp(\cdot)$ preserve ordering, and the second part follows by Markov's inequality (Theorem 10.2.1).

Observe that, by the definition of $\mathbb{E}[\cdot]$ and by the Taylor expansion of $\exp(\cdot)$, we have

$$\begin{aligned} \mathbb{E}[\exp(tX_i)] &= \frac{1}{2}e^t + \frac{1}{2}e^{-t} = \frac{e^t + e^{-t}}{2} \\ &= \frac{1}{2} \left(1 + \frac{t}{1!} + \frac{t^2}{2!} + \frac{t^3}{3!} + \dots \right) \\ &\quad + \frac{1}{2} \left(1 - \frac{t}{1!} + \frac{t^2}{2!} - \frac{t^3}{3!} + \dots \right) \\ &= \left(1 + \frac{t^2}{2!} + \dots + \frac{t^{2k}}{(2k)!} + \dots \right). \end{aligned}$$

Now, $(2k)! = k!(k+1)(k+2)\dots 2k \geq k!2^k$, and thus

$$\mathbb{E}[\exp(tX_i)] = \sum_{i=0}^{\infty} \frac{t^{2i}}{(2i)!} \leq \sum_{i=0}^{\infty} \frac{t^{2i}}{2^i(i!)} = \sum_{i=0}^{\infty} \frac{1}{i!} \left(\frac{t^2}{2}\right)^i = \exp\left(\frac{t^2}{2}\right),$$

again, by the Taylor expansion of $\exp(\cdot)$. Next, by the independence of the X_i s, we have

$$\begin{aligned} \mathbb{E}[\exp(tY)] &= \mathbb{E}\left[\exp\left(\sum_i tX_i\right)\right] = \mathbb{E}\left[\prod_i \exp(tX_i)\right] = \prod_{i=1}^n \mathbb{E}[\exp(tX_i)] \\ &\leq \prod_{i=1}^n \exp\left(\frac{t^2}{2}\right) = \exp\left(\frac{nt^2}{2}\right). \end{aligned}$$

We have, by Eq. (10.2), that

$$\mathbb{P}[Y \geq \Delta] \leq \frac{\mathbb{E}[\exp(tY)]}{\exp(t\Delta)} \leq \frac{\exp\left(\frac{nt^2}{2}\right)}{\exp(t\Delta)} = \exp\left(\frac{nt^2}{2} - t\Delta\right).$$

Next, we select the value of t that minimizes the right term in the above inequality. Easy calculation shows that the right value is $t = \Delta/n$. We conclude that

$$\mathbb{P}[Y \geq \Delta] \leq \exp\left(\frac{n}{2}\left(\frac{\Delta}{n}\right)^2 - \frac{\Delta}{n}\Delta\right) = \exp\left(-\frac{\Delta^2}{2n}\right). \quad \blacksquare$$

Note, the above theorem states that

$$\mathbb{P}[Y \geq \Delta] = \sum_{i=\Delta}^n \mathbb{P}[Y = i] = \sum_{i=n/2+\Delta/2}^n \frac{\binom{n}{i}}{2^n} \leq \exp\left(-\frac{\Delta^2}{2n}\right),$$

since $Y = \Delta$ means that we got $n/2 + \Delta/2$ times +1s and $n/2 - \Delta/2$ times (-1)s.

By the symmetry of Y , we get the following corollary.

Corollary 10.2.4. Let X_1, \dots, X_n be n independent random variables, such that $\mathbb{P}[X_i = 1] = \mathbb{P}[X_i = -1] = \frac{1}{2}$, for $i = 1, \dots, n$. Let $Y = \sum_{i=1}^n X_i$. Then, for any $\Delta > 0$, we have

$$\mathbb{P}[|Y| \geq \Delta] \leq 2 \exp\left(-\frac{\Delta^2}{2n}\right).$$

By easy manipulation, we get the following result.

Corollary 10.2.5. Let X_1, \dots, X_n be n independent coin flips, such that $\mathbb{P}[X_i = 1] = \mathbb{P}[X_i = 0] = \frac{1}{2}$, for $i = 1, \dots, n$. Let $Y = \sum_{i=1}^n X_i$. Then, for any $\Delta > 0$, we have

$$\mathbb{P}\left[\frac{n}{2} - Y \geq \Delta\right] \leq \exp\left(-\frac{2\Delta^2}{n}\right) \quad \text{and} \quad \mathbb{P}\left[Y - \frac{n}{2} \geq \Delta\right] \leq \exp\left(-\frac{2\Delta^2}{n}\right).$$

In particular, we have $\mathbb{P}\left[\left|Y - \frac{n}{2}\right| \geq \Delta\right] \leq 2 \exp\left(-\frac{2\Delta^2}{n}\right)$.

Proof: Transform X_i into the random variable $Z_i = 2X_i - 1$, and now use [Theorem 10.2.3](#) on the new random variables Z_1, \dots, Z_n . ■

Lemma 10.1.1 (Restatement.) In a sequence of M coin flips, the probability that the number of ones is smaller than $L \leq M/4$ is at most $\exp(-M/8)$.

Proof: Let $Y = \sum_{i=1}^M X_i$ the sum of the M coin flips. By the above corollary, we have:

$$\mathbb{P}[Y \leq L] = \mathbb{P}\left[\frac{M}{2} - Y \geq \frac{M}{2} - L\right] = \mathbb{P}\left[\frac{M}{2} - Y \geq \Delta\right],$$

where $\Delta = M/2 - L \geq M/4$. Using the above Chernoff inequality, we get

$$\mathbb{P}[Y \leq L] \leq \exp\left(-\frac{2\Delta^2}{M}\right) \leq \exp(-M/8). \quad \blacksquare$$

10.2.2.1. The Chernoff Bound — General Case

Here we present the Chernoff bound in a more general settings.

Problem 10.2.6. Let X_1, \dots, X_n be n independent Bernoulli trials, where

$$\mathbb{P}[X_i = 1] = p_i \quad \text{and} \quad \mathbb{P}[X_i = 0] = 1 - p_i,$$

and let denote

$$Y = \sum_i X_i \quad \mu = \mathbb{E}[Y].$$

Question: what is the probability that $Y \geq (1 + \delta)\mu$.

Theorem 10.2.7 (Chernoff inequality). For any $\delta > 0$,

$$\mathbb{P}[Y > (1 + \delta)\mu] < \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}}\right)^\mu.$$

Or in a more simplified form, for any $\delta \leq 2e - 1$,

$$\mathbb{P}[Y > (1 + \delta)\mu] < \exp\left(-\mu\delta^2/4\right), \quad (10.3)$$

and

$$\mathbb{P}[Y > (1 + \delta)\mu] < 2^{-\mu(1+\delta)},$$

for $\delta \geq 2e - 1$.

Theorem 10.2.8. *Under the same assumptions as the theorem above, we have*

$$\mathbb{P}[Y < (1 - \delta)\mu] \leq \exp\left(-\mu \frac{\delta^2}{2}\right).$$

The proofs of those more general form, follows the proofs shown above, and are omitted. The interested reader can get the proofs from:

http://www.uiuc.edu/~sariel/teach/2002/a/notes/07_chernoff.ps

10.3. Treaps

Anybody that ever implemented a balanced binary tree, knows that it can be very painful. A natural question, is whether we can use randomization to get a simpler data-structure with good performance.

10.3.1. Construction

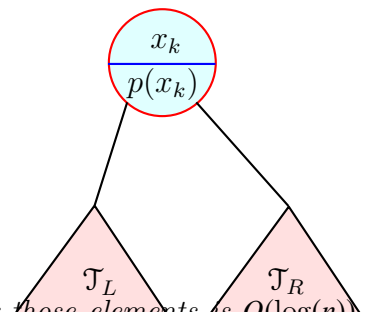
The key observation is that many of data-structures that offer good performance for balanced binary search trees, do so by storing additional information to help in how to balance the tree. As such, the key Idea is that for every element x inserted into the data-structure, randomly choose a priority $p(x)$; that is, $p(x)$ is chosen uniformly and randomly in the range $[0, 1]$.

So, for the set of elements $X = \{x_1, \dots, x_n\}$, with (random) priorities $p(x_1), \dots, p(x_n)$, our purpose is to build a binary tree which is “balanced”. So, let us pick the element x_k with the lowest priority in X , and make it the root of the tree. Now, we partition X in the natural way:

- (A) L : set of all the numbers smaller than x_k in X , and
- (B) R : set of all the numbers larger than x_k in X .

We can now build recursively the trees for L and R , and let denote them by \mathcal{T}_L and \mathcal{T}_R . We build the natural tree, by creating a node for x_k , having \mathcal{T}_L its left child, and \mathcal{T}_R as its right child.

We call the resulting tree a **treap**. As it is a tree over the elements, and a heap over the priorities; that is, TREAP = TREE + HEAP.



Lemma 10.3.1. *Given n elements, the expected depth of a treap \mathcal{T} defined over those elements is $O(\log(n))$. Furthermore, this holds with high probability; namely, the probability that the depth of the treap would exceed $c \log n$ is smaller than $\delta = n^{-d}$, where d is an arbitrary constant, and c is a constant that depends on d .^②*

Furthermore, the probability that \mathcal{T} has depth larger than $ct \log(n)$, for any $t \geq 1$, is smaller than n^{-dt} .

Proof: Observe, that every element has equal probability to be in the root of the treap. Thus, the structure of a treap, is identical to the recursive tree of **QuickSort**. Indeed, imagine that instead of picking the pivot uniformly at random, we instead pick the pivot to be the element with the lowest (random) priority. Clearly, these two ways of choosing pivots are equivalent. As such, the claim follows immediately from our analysis of the depth of the recursion tree of **QuickSort**, see **Theorem 10.1.2**_{p71}. ■

10.3.2. Operations

The following innocent observation is going to be the key insight in implementing operations on treaps:

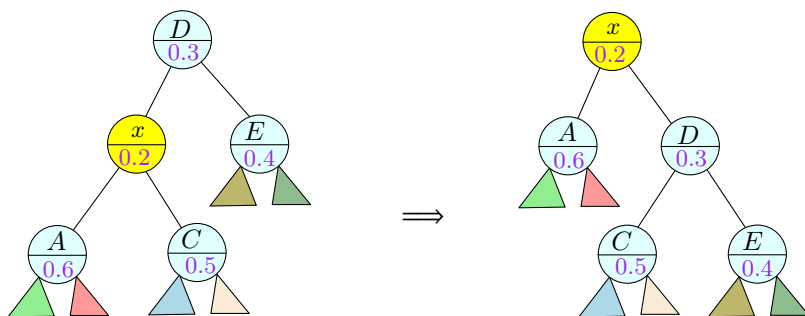
Observation 10.3.2. *Given n distinct elements, and their (distinct) priorities, the treap storing them is uniquely defined.*

^②That is, if we want to decrease the probability of failure, that is δ , we need to increase c .

10.3.2.1. Insertion

Given an element x to be inserted into an existing treap \mathcal{T} , insert it in the usual way into \mathcal{T} (i.e., treat it a regular search binary tree). This takes $O(\text{height}(\mathcal{T}))$. Now, x is a leaf in the treap. Set x priority $p(x)$ to some random number $[0, 1]$. Now, while the new tree is a valid search tree, it is not necessarily still a valid treap, as x 's priority might be smaller than its parent. So, we need to fix the tree around x , so that the priority property holds.

We call **RotateUp**(x) to do so. Specifically, if x parent is y , and $p(x) < p(y)$, we will rotate x up so that it becomes the parent of y . We repeatedly do it till x has a larger priority than its parent. The rotation operation takes constant time and plays around with priorities, and importantly, it preserves the binary search tree order. Here is a rotate right operation **RotateRight**(D):



```

RotateUp( $x$ )
 $y \leftarrow \text{parent}(x)$ 
while  $p(y) > p(x)$  do
    if  $y.\text{left\_child} = x$  then
        RotateRight( $y$ )
    else
        RotateLeft( $y$ )
 $y \leftarrow \text{parent}(x)$ 
    
```

RotateLeft is the same tree rewriting operation done in the other direction.

In the end of this process, both the ordering property and the priority property holds. That is, we have a valid treap that includes all the old elements, and the new element. By **Observation 10.3.2**, since the treap is uniquely defined, we have updated the treap correctly. Since every time we do a rotation the distance of x from the root decrease by one, it follows that insertions takes $O(\text{height}(\mathcal{T}))$.

10.3.2.2. Deletion

Deletion is just an insertion done in reverse. Specifically, to delete an element x from a treap \mathcal{T} , set its priority to $+\infty$, and rotate it down it becomes a leaf. The only tricky observation is that you should rotate always so that the child with the lower priority becomes the new parent. Once x becomes a leaf deleting it is trivial - just set the pointer pointing to it in the tree to null.

10.3.2.3. Split

Given an element x stored in a treap \mathcal{T} , we would like to split \mathcal{T} into two treaps - one treap \mathcal{T}_{\leq} for all the elements smaller or equal to x , and the other treap $\mathcal{T}_{>}$ for all the elements larger than x . To this end, we set x priority to $-\infty$, fix the priorities by rotating x up so it becomes the root of the treap. The right child of x is the treap $\mathcal{T}_{>}$, and we disconnect it from \mathcal{T} by setting x right child pointer to null. Next, we restore x to its real priority, and rotate it down to its natural location. The resulting treap is \mathcal{T}_{\leq} . This again takes time that is proportional to the depth of the treap.

10.3.2.4. Meld

Given two treaps \mathcal{T}_L and \mathcal{T}_R such that all the elements in \mathcal{T}_L are smaller than all the elements in \mathcal{T}_R , we would like to merge them into a single treap. Find the largest element x stored in \mathcal{T}_L (this is just the element stored in the path going only right from the root of the tree). Set x priority to $-\infty$, and rotate it up the treap so that

it becomes the root. Now, x being the largest element in \mathcal{T}_L has no right child. Attach \mathcal{T}_R as the right child of x . Now, restore x priority to its original priority, and rotate it back so the priorities properties hold.

10.3.3. Summery

Theorem 10.3.3. *Let \mathcal{T} be a treap, initialized to an empty treap, and undergoing a sequence of $m = n^c$ insertions, where c is some constant. The probability that the depth of the treap in any point in time would exceed $d \log n$ is $\leq 1/n^f$, where d is an arbitrary constant, and f is a constant that depends only c and d .*

In particular, a treap can handle insertion/deletion in $O(\log n)$ time with high probability.

Proof: Since the first part of the theorem implies that with high probability all these treaps have logarithmic depth, then this implies that all operations takes logarithmic time, as an operation on a treap takes at most the depth of the treap.

As for the first part, let $\mathcal{T}_1, \dots, \mathcal{T}_m$ be the sequence of treaps, where \mathcal{T}_i is the treap after the i th operation. Similarly, let X_i be the set of elements stored in \mathcal{T}_i . By [Lemma 10.3.1](#), the probability that \mathcal{T}_i has large depth is tiny. Specifically, we have that

$$\alpha_i = \mathbb{P}[\text{depth}(\mathcal{T}_i) > tc' \log n^c] = \mathbb{P}\left[\text{depth}(\mathcal{T}_i) > c't \left(\frac{\log n^c}{\log |\mathcal{T}_i|}\right) \cdot \log |\mathcal{T}_i|\right] \leq \frac{1}{n^{t \cdot c}},$$

as a tedious and boring but straightforward calculation shows. Picking t to be sufficiently large, we have that the probability that the i th treap is too deep is smaller than $1/n^{f+c}$. By the union bound, since there are n^c treaps in this sequence of operations, it follows that the probability of any of these treaps to be too deep is at most $1/n^f$, as desired. ■

10.4. Bibliographical Notes

Chernoff inequality was a rediscovery of Bernstein inequality, which was published in 1924 by Sergei Bernstein. Treaps were invented by Siedel and Aragon [[SA96](#)]. Experimental evidence suggests that Treaps performs reasonably well in practice, despite their simplicity, see for example the comparison carried out by Cho and Sahni [[CS00](#)]. Implementations of treaps are readily available. An old implementation I wrote in C is available here: <http://valis.cs.uiuc.edu/blog/?p=6060>.

Chapter 11

Hashing

“I tried to read this book, Huckleberry Finn, to my grandchildren, but I couldn’t get past page six because the book is fraught with the ‘n-word.’ And although they are the deepest-thinking, combat-ready eight- and ten-year-olds I know, I knew my babies weren’t ready to comprehend Huckleberry Finn on its own merits. That’s why I took the liberty to rewrite Mark Twain’s masterpiece. Where the repugnant ‘n-word’ occurs, I replaced it with ‘warrior’ and the word ‘slave’ with ‘dark-skinned volunteer.’”

Paul Beatty, *The Sellout*

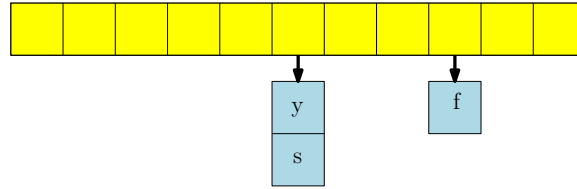


Figure 11.1: Open hashing.

11.1. Introduction

We are interested here in dictionary data structure. The settings for such a data-structure:

- (A) \mathcal{U} : universe of keys with total order: numbers, strings, etc.
- (B) Data structure to store a subset $S \subseteq \mathcal{U}$
- (C) **Operations:**
 - (A) **search/lookup**: given $x \in \mathcal{U}$ is $x \in S$?
 - (B) **insert**: given $x \notin S$ add x to S .
 - (C) **delete**: given $x \in S$ delete x from S
- (D) **Static** structure: S given in advance or changes very infrequently, main operations are lookups.
- (E) **Dynamic** structure: S changes rapidly so inserts and deletes as important as lookups.

Common constructions for such data-structures, include using a static sorted array, where the lookup is a binary search. Alternatively, one might use a *balanced* search tree (i.e., red-black tree). The time to perform an operation like lookup, insert, delete take $O(\log |S|)$ time (comparisons).

Naturally, the above are potentially an “overkill”, in the sense that sorting is unnecessary. In particular, the universe \mathcal{U} may not be (naturally) totally ordered. The keys correspond to large objects (images, graphs etc) for which comparisons are expensive. Finally, we would like to improve “average” performance of lookups to $O(1)$ time, even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Hashing and Hash Tables. The hash-table data structure has an associated (hash) table/array T of size m (the table *size*). A hash function $h : \mathcal{U} \rightarrow \{0, \dots, m - 1\}$. An item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T .

Given a set $S \subseteq \mathcal{U}$, in a perfect ideal situation, each element $x \in S$ hashes to a distinct slot in T , and we store x in the slot $h(x)$. The **Lookup** for an item $y \in \mathcal{U}$, is to check if $T[h(y)] = y$. This takes constant time.

Unfortunately, *collisions* are unavoidable, and several different techniques to handle them. Formally, two items $x \neq y$ *collide* if $h(x) = h(y)$.

A standard technique to handle collisions is to use **chaining** (aka *open hashing*). Here, we handle collisions as follows:

- (A) For each slot i store all items hashed to slot i in a linked list. $T[i]$ points to the linked list.
- (B) **Lookup**: to find if $y \in \mathcal{U}$ is in T , check the linked list at $T[h(y)]$. Time proportion to size of linked list.

Other techniques for handling collisions include associating a list of locations where an element can be (in certain order), and check these locations in this order. Another useful technique is **cuckoo hashing** which we will discuss later on: Every value has two possible locations. When inserting, insert in one of the locations, otherwise, kick out the stored value to its other location. Repeat till stable. if no stability then rebuild table.

The relevant questions when designing a hashing scheme, include: (I) Does hashing give $O(1)$ time per operation for dictionaries? (II) Complexity of evaluating h on a given element? (III) Relative sizes of the universe \mathcal{U} and the set to be stored S . (IV) Size of table relative to size of S . (V) Worst-case vs average-case vs randomized (expected) time? (VI) How do we choose h ?

The **load factor** of the array T is the ratio n/t where $n = |S|$ is the number of elements being stored and $m = |T|$ is the size of the array being used. Typically n/t is a small constant smaller than 1.

In the following, we assume that \mathcal{U} (the universe the keys are taken from) is large – specifically, $N = |\mathcal{U}| \gg m^2$, where m is the size of the table. Consider a hash function $h : \mathcal{U} \rightarrow \{0, \dots, m-1\}$. If hash N items to the m slots, then by the pigeon hole principle, there is some $i \in \{0, \dots, m-1\}$ such that $N/m \geq m$ elements of \mathcal{U} get hashed to i . In particular, this implies that there is set $S \subseteq \mathcal{U}$, where $|S| = m$ such that all of S hashes to same slot. Oops.

Namely, for every hash function there is a *bad set* with many collisions.

Observation 11.1.1. *Let \mathcal{H} be the set of all functions from $\mathcal{U} = \{1, \dots, U\}$ to $\{1, \dots, m\}$. The number of functions in \mathcal{H} is m^U . As such, specifying a function in \mathcal{H} would require $\log_2 |\mathcal{H}| = O(U \log m)$.*

As such, picking a truly random hash function requires many random bits, and furthermore, it is not even clear how to evaluate it efficiently (which is the whole point of hashing).

Picking a hash function. Picking a good hash function in practice is a dark art involving many non-trivial considerations and ideas. For parameters $N = |\mathcal{U}|$, $m = |T|$, and $n = |S|$, we require the following:

- (A) \mathcal{H} is a *family* of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$).
- (B) h is chosen *randomly* from \mathcal{H} (typically uniformly at random). Implicitly assumes that \mathcal{H} allows an efficient sampling.
- (C) Require that for any *fixed* set $S \subseteq \mathcal{U}$, of size m , the expected number of collisions for a function chosen from \mathcal{H} should be “small”. Here the expectation is over the randomness in choice of h .

11.2. Universal Hashing

We would like the hash function to have the following property – For any element $x \in \mathcal{U}$, and a random $h \in \mathcal{H}$, then $h(x)$ should have a uniform distribution. That is $\Pr[h(x) = i] = 1/m$, for every $0 \leq i < m$. A somewhat stronger property is that for any two distinct elements $x, y \in \mathcal{U}$, for a random $h \in \mathcal{H}$, the probability of a collision between x and y should be at most $1/m$. $\mathbb{P}[h(x) = h(y)] = 1/m$.

Definition 11.2.1. A family \mathcal{H} of hash functions is **2-universal** if for all distinct $x, y \in \mathcal{U}$, we have $\mathbb{P}[h(x) = h(y)] \leq 1/m$.

Applying a 2-universal family hash function to a set of distinct numbers, results in a 2-wise independent sequence of numbers.

Lemma 11.2.2. *Let S be a set of n elements stored using open hashing in a hash table of size m , using open hashing, where the hash function is picked from a 2-universal family. Then, the expected lookup time, for any element $x \in \mathcal{U}$ is $O(n/m)$.*

Proof: The number of elements colliding with x is $\ell(x) = \sum_{y \in S} D_y$, where $D_y = 1 \iff x$ and y collide under the hash function h . As such, we have

$$\mathbb{E}[\ell(x)] = \sum_{y \in S} \mathbb{E}[D_y] = \sum_{y \in S} \mathbb{P}[h(x) = h(y)] = \sum_{y \in S} \frac{1}{m} = |S|/m = n/m. \quad \blacksquare$$

Remark 11.2.3. The above analysis holds even if we perform a sequence of $O(n)$ insertions/deletions operations. Indeed, just repeat the analysis with the set of elements being all elements encountered during these operations.

The worst-case bound is of course much worse – it is not hard to show that in the worst case, the load of a single hash table entry might be $\Omega(\log n / \log \log n)$ (as we seen in the occupancy problem).

Rehashing, amortization, etc. The above assumed that the set S is fixed. If items are inserted and deleted, then the hash table might become much worse. In particular, $|S|$ grows to more than cm , for some constant c , then hash table performance start degrading. Furthermore, if many insertions and deletions happen then the initial random hash function is no longer random enough, and the above analysis no longer holds.

A standard solution is to rebuild the hash table periodically. We choose a new table size based on current number of elements in table, and a new random hash function, and rehash the elements. And then discard the old table and hash function. In particular, if $|S|$ grows to more than twice current table size, then rebuild new hash table (choose a new random hash function) with double the current number of elements. One can do a similar shrinking operation if the set size falls below quarter the current hash table size.

If the working $|S|$ stays roughly the same but more than $c|S|$ operations on table for some chosen constant c (say 10), rebuild.

The *amortize* cost of rebuilding to previously performed operations. Rebuilding ensures $O(1)$ expected analysis holds even when S changes. Hence $O(1)$ expected look up/insert/delete time *dynamic* data dictionary data structure!

11.2.1. How to build a 2-universal family

11.2.1.1. On working modulo prime

Definition 11.2.4. For a number p , let $\mathbb{Z}_n = \{0, \dots, n-1\}$.

For two integer numbers x and y , the *quotient* of x/y is $x \text{ div } y = \lfloor x/y \rfloor$. The *remainder* of x/y is $x \bmod y = x - y \lfloor x/y \rfloor$. If the $x \bmod y = 0$, than y *divides* x , denoted by $y \mid x$. We use $\alpha \equiv \beta \pmod{p}$ or $\alpha \equiv_p \beta$ to denote that α and β are *congruent modulo p* ; that is $\alpha \bmod p = \beta \bmod p$ – equivalently, $p \mid (\alpha - \beta)$.

Lemma 11.2.5. *Let p be a prime number.*

- (A) *For any $\alpha, \beta \in \{1, \dots, p-1\}$, we have that $\alpha\beta \not\equiv 0 \pmod{p}$.*
- (B) *For any $\alpha, \beta, i \in \{1, \dots, p-1\}$, such that $\alpha \neq \beta$, we have that $\alpha i \not\equiv \beta i \pmod{p}$.*
- (C) *For any $x \in \{1, \dots, p-1\}$ there exists a unique y such that $xy \equiv 1 \pmod{p}$. The number y is the **inverse** of x , and is denoted by x^{-1} or $1/x$.*

Proof: (A) If $\alpha\beta \equiv 0 \pmod{p}$, then p must divide $\alpha\beta$, as it divides 0. But α, β are smaller than p , and p is prime. This implies that either $p \mid \alpha$ or $p \mid \beta$, which is impossible.

(B) Assume that $\alpha > \beta$. Furthermore, for the sake of contradiction, assume that $\alpha i \equiv \beta i \pmod{p}$. But then, $(\alpha - \beta)i \equiv 0 \pmod{p}$, which is impossible, by (A).

(C) For any $\alpha \in \{1, \dots, p-1\}$, consider the set $L_\alpha = \{\alpha * 1 \bmod p, \alpha * 2 \bmod p, \dots, \alpha * (p-1) \bmod p\}$. By (A), zero is not in L_α , and by (B), L_α must contain $p-1$ distinct values. It follows that $L_\alpha = \{1, 2, \dots, p-1\}$. As such, there exists exactly one number $y \in \{1, \dots, p-1\}$, such that $\alpha y \equiv 1 \pmod{p}$. ■

Lemma 11.2.6. *Consider a prime p , and any numbers $x, y \in \mathbb{Z}_p$. If $x \neq y$ then, for any $a, b \in \mathbb{Z}_p$, such that $a \neq 0$, we have $ax + b \not\equiv ay + b \pmod{p}$.*

Proof: Assume $y > x$ (the other case is handled similarly). If $ax + b \equiv ay + b \pmod{p}$ then $a(x - y) \pmod{p} = 0$ and $a \neq 0$ and $(x - y) \neq 0$. However, a and $x - y$ cannot divide p since p is prime and $a < p$ and $0 < x - y < p$. ■

Lemma 11.2.7. *Consider a prime p , and any numbers $x, y \in \mathbb{Z}_p$. If $x \neq y$ then, for each pair of numbers $r, s \in \mathbb{Z}_p = \{0, 1, \dots, p-1\}$, such that $r \neq s$, there is exactly one unique choice of numbers $a, b \in \mathbb{Z}_p$ such that $ax + b \pmod{p} = r$ and $ay + b \pmod{p} = s$.*

Proof: Solve the system of equations

$$ax + b \equiv r \pmod{p} \quad \text{and} \quad ay + b \equiv s \pmod{p}.$$

We get $a = \frac{r-s}{x-y} \pmod{p}$ and $b = r - ax \pmod{p}$. ■

11.2.1.2. Constructing a family of 2-universal hash functions

For parameters $N = |\mathcal{U}|$, $m = |T|$, $n = |S|$. Choose a *prime* number $p \geq N$. Let

$$\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p \text{ and } a \neq 0\},$$

where $h_{a,b}(x) = ((ax + b) \pmod{p}) \pmod{m}$. Note that $|\mathcal{H}| = p(p-1)$.

11.2.1.3. Analysis

Once we fix a and b , and we are given a value x , we compute the hash value of x in two stages:

- (A) **Compute:** $r \leftarrow (ax + b) \pmod{p}$.
- (B) **Fold:** $r' \leftarrow r \pmod{m}$

Lemma 11.2.8. *Assume that p is a prime, and $1 < m < p$. The number of pairs $(r, s) \in \mathbb{Z}_p \times \mathbb{Z}_p$, such that $r \neq s$, that are folded to the same number is $\leq p(p-1)/m$. Formally, the set of bad pairs*

$$B = \{(r, s) \in \mathbb{Z}_p \times \mathbb{Z}_p \mid r \equiv_m s\}$$

is of size at most $p(p-1)/m$.

Proof: Consider a pair $(x, y) \in \{0, 1, \dots, p-1\}^2$, such that $x \neq y$. For a fixed x , there are at most $\lceil p/m \rceil$ values of y that fold into x . Indeed, $x \equiv_m y$ if and only if

$$y \in L(x) = \{x + im \mid i \text{ is an integer}\} \cap \mathbb{Z}_p.$$

The size of $L(x)$ is maximized when $x = 0$. The number of such elements is at most $\lceil p/m \rceil$ (note, that since p is a prime, p/m is fractional). One of the numbers in $O(x)$ is x itself. As such, we have that

$$|B| \leq p(|L(x)| - 1) \leq p(\lceil p/m \rceil - 1) \leq p(p-1)/m,$$

since $\lceil p/m \rceil - 1 \leq (p-1)/m \iff m \lceil p/m \rceil - m \leq p-1 \iff m \lfloor p/m \rfloor \leq p-1 \iff m \lfloor p/m \rfloor < p$, which is true since p is a prime, and $1 < m < p$. ■

Claim 11.2.9. *For two distinct numbers $x, y \in \mathcal{U}$, a pair a, b is **bad** if $h_{a,b}(x) = h_{a,b}(y)$. The number of bad pairs is $\leq p(p-1)/m$.*

Proof: Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$. Let

$$r = (ax + b) \pmod{p} \quad \text{and} \quad s = (ay + b) \pmod{p}.$$

By **Lemma 11.2.6**, we have that $r \neq s$. As such, a collision happens if $r \equiv s \pmod{m}$. By **Lemma 11.2.8**, the number of such pairs (r, s) is at most $p(p-1)/m$. By **Lemma 11.2.7**, for each such pair (r, s) , there is a unique choice of a, b that maps x and y to r and s , respectively. As such, there are at most $p(p-1)/m$ bad pairs. ■

Theorem 11.2.10. *The hash family \mathcal{H} is a 2-universal hash family.*

Proof: Fix two distinct numbers $x, y \in \mathcal{U}$. We are interested in the probability they collide if h is picked randomly from \mathcal{H} . By **Claim 11.2.9** there are $M \leq p(p-1)/m$ bad pairs that causes such a collision, and since \mathcal{H} contains $N = p(p-1)$ functions, it follows the probability for collision is $M/N \leq 1/m$, which implies that \mathcal{H} is 2-universal. ■

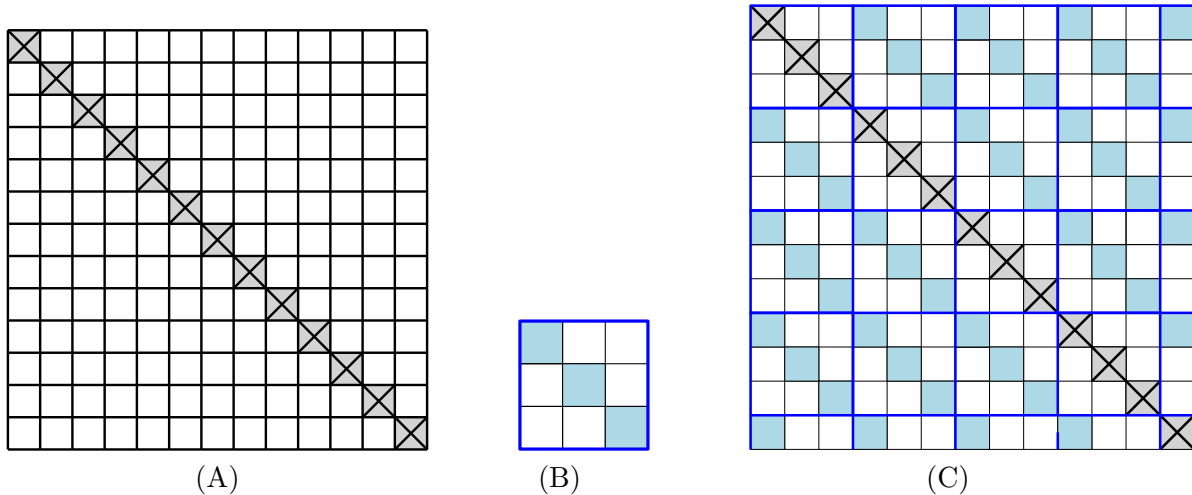


Figure 11.2: Explanation of the hashing scheme via figures.

11.2.1.4. Explanation via pictures

Consider a pair $(x, y) \in \mathbb{Z}_p^2$, such that $x \neq y$. This pair (x, y) corresponds to a cell in the natural “grid” \mathbb{Z}_p^2 that is off the main diagonal. See [Figure 11.2](#)

The mapping $f_{a,b}(x) = (ax + b) \bmod p$, takes the pair (x, y) , and maps it randomly and uniformly, to some other pair $x' = f_{a,b}(x)$ and $y' = f_{a,b}(y)$ (where x', y' are again off the main diagonal).

Now consider the smaller grid $\mathbb{Z}_m \times \mathbb{Z}_m$. The main diagonal of this subgrid is bad – it corresponds to a collision. One can think about the last step, of computing $h_{a,b}(x) = f_{a,b}(x) \bmod m$, as tiling the larger grid, by the smaller grid. in the natural way. Any diagonal that is in distance mi from the main diagonal get marked as bad. At most $1/m$ fraction of the off diagonal cells get marked as bad. See [Figure 11.2](#).

As such, the random mapping of (x, y) to (x', y') causes a collision only if we map the pair to a badly marked pair, and the probability for that $\leq 1/m$.

11.3. Perfect hashing

An interesting special case of hashing is the static case – given a set S of elements, we want to hash S so that we can answer membership queries efficiently (i.e., dictionary data-structures with no insertions). it is easy to come up with a hashing scheme that is optimal as far as space.

11.3.1. Some easy calculations

The first observation is that if the hash table is quadratically large, then there is a good (constant) probability to have no collisions (this is also the threshold for the birthday paradox).

Lemma 11.3.1. *Let $S \subseteq \mathcal{U}$ be a set of n elements, and let \mathcal{H} be a 2-universal family of hash functions, into a table of size $m \geq n^2$. Then with probability $\leq 1/2$, there is a pair of elements of S that collide under a random hash function $h \in \mathcal{H}$.*

Proof: For a pair $x, y \in S$, the probability they collide is at most $\leq 1/m$, by [definition](#). As such, by the union bound, the probability of any collision is $\binom{n}{2}/m = n(n-1)/2m \leq 1/2$. ■

We now need a second moment bound on the sizes of the buckets.

Lemma 11.3.2. *Let $S \subseteq \mathcal{U}$ be a set of n elements, and let \mathcal{H} be a 2-universal family of hash functions, into a table of size $m \geq cn$, where c is an arbitrary constant. Let $h \in \mathcal{H}$ be a random hash function, and*

let X_i be the number of elements of S mapped to the i th bucket by h , for $i = 0, \dots, m-1$. Then, we have $\mathbb{E}\left[\sum_{j=0}^{m-1} X_j^2\right] \leq (1 + 2/c)n$.

Proof: Let s_1, \dots, s_n be the n items in S , and let $Z_{i,j} = 1$ if $h(s_i) = h(s_j)$, for $i < j$. Observe that $\mathbb{E}[Z_{i,j}] = \mathbb{P}[h(s_i) = h(s_j)] \leq 1/m$ (this is the only place we use the property that \mathcal{H} is 2-universal). In particular, let $\mathcal{Z}(\alpha)$ be all the variables $Z_{i,j}$, for $i < j$, such that $Z_{i,j} = 1$ and $h(s_i) = h(s_j) = \alpha$.

If for some α we have that $X_\alpha = k$, then there are k indices $\ell_1 < \ell_2 < \dots < \ell_k$, such that $h(s_{\ell_1}) = \dots = h(s_{\ell_k}) = \alpha$. As such, $z(\alpha) = |\mathcal{Z}(\alpha)| = \binom{k}{2}$. In particular, we have

$$X_\alpha^2 = k^2 = 2\binom{k}{2} + k = 2z(\alpha) + X_\alpha$$

This implies that

$$\sum_{\alpha=0}^{m-1} X_\alpha^2 = \sum_{\alpha=0}^{m-1} (2z(\alpha) + X_\alpha) = 2 \sum_{\alpha=0}^{m-1} z(\alpha) + \sum_{\alpha=0}^{m-1} X_\alpha = 2 \sum_{i\alpha=0}^{m-1} z(\alpha) + \sum_{\alpha=0}^{m-1} X_\alpha = n + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^n Z_{ij}$$

Now, by linearity of expectations, we have

$$\mathbb{E}\left[\sum_{\alpha=0}^{m-1} X_\alpha^2\right] = \mathbb{E}\left[n + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^n Z_{ij}\right] = n + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[Z_{ij}] \leq n + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{1}{m} \leq n\left(1 + 2\frac{n}{m}\right) \leq n\left(1 + \frac{2}{c}\right)$$

since $m \geq cn$. ■

11.3.2. Construction of perfect hashing

Given a set S of n elements, we build a open hash table T of size, say, $2n$. We use a random hash function h that is 2-universal for this hash table, see [Theorem 11.2.10](#). Next, we map the elements of S into the hash table. Let S_j be the list of all the elements of S mapped to the j th bucket, and let $X_j = |S_j|$, for $j = 0, \dots, n-1$.

We compute $Y = \sum_{i=1}^n X_i^2$. If $Y > 6n$, then we reject h , and resample a hash function h . We repeat this process till success.

In the second stage, we build secondary hash tables for each bucket. Specifically, for $j = 0, \dots, 2n-1$, if the j th bucket contains $X_j > 0$ elements, then we construct a secondary hash table H_j to store the elements of S_j , and this secondary hash table has size X_j^2 , and again we use a random 2-universal hash function h_j for the hashing of S_j into H_j . If any pair of elements of S_j collide under h_j , then we resample the hash function h_j , and try again till success.

11.3.2.1. Analysis

Theorem 11.3.3. *Given a (static) set $S \subseteq \mathcal{U}$ of n elements, the above scheme, constructs, in expected linear time, a two level hash-table that can perform search queries in $O(1)$ time. The resulting data-structure uses $O(n)$ space.*

Proof: Given an element $x \in \mathcal{U}$, we first compute $j = h(x)$, and then $k = h_j(x)$, and we can check whether the element stored in the secondary hash table H_j at the entry k is indeed x . As such, the search time is $O(1)$.

The more interesting issue is the construction time. Let X_j be the number of elements mapped to the j th bucket, and let $Y = \sum_{i=1}^n X_i^2$. Observe, that $\mathbb{E}[Y] = (1 + 2/2)n = 2n$, by [Lemma 11.3.2](#) (here, $m = 2n$ and $c = 2$). As such, by Markov's inequality, $\mathbb{P}[Y > 6n] \leq 1/2$. In particular, picking a good top level hash function requires in expectation $1/(1/2) = 2$ iterations. Thus the first stage takes $O(n)$ time, in expectation.

For the j th bucket, with X_j entries, by [Lemma 11.3.1](#), the construction succeeds with probability $\geq 1/2$. As before, the expected number of iterations till success is at most 2. As such, the expected construction time of the secondary hash table for the j th bucket is $O(X_j^2)$.

We conclude that the overall expected construction time is $O(n + \sum_j X_j^2) = O(n)$.

As for the space used, observe that it is $O(n + \sum_j X_j^2) = O(n)$. ■

11.4. Bloom filters

Consider an application where we have a set $S \subseteq \mathcal{U}$ of n elements, and we want to be able to decide for a query $x \in \mathcal{U}$, whether or not $x \in S$. Naturally, we can use hashing. However, here we are interested in more efficient data-structure as far as space. We allow the data-structure to make a mistake (i.e., say that an element is in, when it is not in).

First try. So, let start silly. Let $B[0 \dots m]$ be an array of bits, and pick a random hash function $h : \mathcal{U} \rightarrow \mathbb{Z}_m$. Initialize B to 0. Next, for every element $s \in S$, set $B[h(s)]$ to 1. Now, given a query, return $B[h(x)]$ as an answer whether or not $x \in S$. Note, that B is an array of bits, and as such it can be bit-packed and stored efficiently.

For the sake of simplicity of exposition, assume that the hash functions picked is truly random. As such, we have that the probability for a false positive (i.e., a mistake) for a fixed $x \in \mathcal{U}$ is n/m . Since we want the size of the table m to be close to n , this is not satisfying.

Using k hash functions. Instead of using a single hash function, let us use k independent hash functions h_1, \dots, h_k . For an element $s \in S$, we set $B[h_i(s)]$ to 1, for $i = 1, \dots, k$. Given an query $x \in \mathcal{U}$, if $B[h_i(x)]$ is zero, for any $i = 1, \dots, k$, then $x \notin S$. Otherwise, if all these k bits are on, the data-structure returns that x is in S .

Clearly, if the data-structure returns that x is not in S , then it is correct. The data-structure might make a mistake (i.e., a false positive), if it returns that x is in S (when is not in S).

We interpret the storing of the elements of S in B , as an experiment of throwing kn balls into m bins. The probability of a bin to be empty is

$$p = p(m, n) = (1 - 1/m)^{kn} \approx \exp(-k(n/m)).$$

Since the number of empty bins is a martingale, we know the number of empty bins is strongly concentrated around the expectation pm , and we can treat p as the true probability of a bin to be empty.

The probability of a mistake is

$$f(k, m, n) = (1 - p)^k.$$

In particular, for $k = (m/n) \ln n$, we have that $p = p(m, n) \approx 1/2$, and $f(k, m, n) \approx 1/2^{(m/n) \ln 2} \approx 0.618^{m/n}$.

Example 11.4.1. Of course, the above is fictional, as k has to be an integer. But motivated by these calculations, let $m = 3n$, and $k = 4$. We get that $p(m, n) = \exp(-4/3) \approx 0.26359$, and $f(4, 3n, n) \approx (1 - 0.265)^4 \approx 0.294078$. This is better than the naive $k = 1$ scheme, where the probability of false positive is $1/3$.

Note, that this scheme gets exponentially better over the naive scheme as m/n grows.

Example 11.4.2. Consider the setting $m = 8n$ – this is when we allocate a byte for each element stored (the element of course might be significantly bigger). The above implies we should take $k = \lceil (m/n) \ln 2 \rceil = 6$. We then get $p(8n, n) = \exp(-6/8) \approx 0.5352$, and $f(6, 8n, n) \approx 0.0215$. Here, the naive scheme with $k = 1$, would give probability of false positive of $1/8 = 0.125$. So this is a significant improvement.

Remark 11.4.3. It is important to remember that Bloom filters are competing with direct hashing of the whole elements. Even if one allocates 8 bits per item, as in the example above, the space it uses is significantly smaller than regular hashing. A situation when such a Bloom filter makes sense is for a cache – we might want to decide if an element is in a slow external cache (say SSD drive). Retrieving item from the cache is slow, but not so slow we are not willing to have a small overhead because of false positives.

11.5. Bibliographical notes

Practical Issues Hashing used typically for integers, vectors, strings etc.

- Universal hashing is defined for integers. To implement it for other objects, one needs to map objects in some fashion to integers.
- Practical methods for various important cases such as vectors, strings are studied extensively. See http://en.wikipedia.org/wiki/Universal_hashing for some pointers.
- Recent important paper bridging theory and practice of hashing. “The power of simple tabulation hashing” by Mikkel Thorup and Mihai Patrascu, 2011. See http://en.wikipedia.org/wiki/Tabulation_hashing

Chapter 12

Min Cut

To acknowledge the corn - This purely American expression means to admit the losing of an argument, especially in regard to a detail; to retract; to admit defeat. It is over a hundred years old. Andrew Stewart, a member of Congress, is said to have mentioned it in a speech in 1828. He said that haystacks and cornfields were sent by Indiana, Ohio and Kentucky to Philadelphia and New York. Charles A. Wickliffe, a member from Kentucky questioned the statement by commenting that haystacks and cornfields could not walk. Stewart then pointed out that he did not mean literal haystacks and cornfields, but the horses, mules, and hogs for which the hay and corn were raised. Wickliffe then rose to his feet, and said, “Mr. Speaker, I acknowledge the corn”.

Funk, Earle, A Hog on Ice and Other Curious Expressions

12.1. Branching processes – Galton-Watson Process

12.1.1. The problem

In the 19th century, Victorians were worried that aristocratic surnames were disappearing, as family names passed on only through the male children. As such, a family with no male children had its family name disappear. So, imagine the number of male children of a person is an independent random variable $X \in \{0, 1, 2, \dots\}$. Starting with a single person, its family (as far as male children are concerned) is a random tree with the degree of a node being distributed according to X . We continue recursively in constructing this tree, again, sampling the number of children for each current leaf according to the distribution of X . It is not hard to see that a family disappears if $\mathbb{E}[X] \leq 1$, and it has a constant probability of surviving if $\mathbb{E}[X] > 1$.

Francis Galton asked the question of what is the probability of such a blue-blood family name to survive, and this question was answered by Henry William Watson [WG75]. The Victorians were worried about strange things, see [Gre69] for a provocatively titled article from the period, and [Ste12] for a more recent take on this issue.

Of course, since infant mortality is dramatically down (as is the number of aristocrat males dying to maintain the British empire), the probability of family names to disappear is now much lower than it was in the 19th century (not to mention that many women keep their original family name). Interestingly, countries with family names that were introduced long time ago have very few surnames (i.e., Korean have 250 surnames, and three surnames form 45% of the population). On the other hand, countries that introduced surnames more recently have dramatically more surnames (for example, the Dutch have surnames only for the last 200 years, and there are 68,000 different family names).

Here we are going to look on a very specific variant of this problem. Imagine that starting with a single male. A male has exactly two children, and each one of them is a male with probability half. As such, the natural question is what is the probability that h generations down, there is a male decedent that all his ancestors are male.

12.1.2. On coloring trees

Let T_h be a complete binary tree of height h . We randomly color its edges by black and white. Namely, for each edge we independently choose its color to be either black or white, with equal probability (say, black indicates

the child is male). We are interested in the event that there exists a path from the root of T_h to one of its leaves, that is all black. Let \mathcal{E}_h denote this event, and let $\rho_h = \mathbb{P}[\mathcal{E}_h]$. Observe that $\rho_0 = 1$ and $\rho_1 = 3/4$ (see below).

To bound this probability, consider the root u of T_h and its two children u_l and u_r . The probability that there is a black path from u_l to one of its children is ρ_{h-1} , and as such, the probability that there is a black path from u through u_l to a leaf of the subtree of u_l is $\mathbb{P}[\text{the edge } uu_l \text{ is colored black}] \cdot \rho_{h-1} = \rho_{h-1}/2$. As such, the probability that there is no black path through u_l is $1 - \rho_{h-1}/2$. As such, the probability of not having a black path from u to a leaf (through either children) is $(1 - \rho_{h-1}/2)^2$. In particular, there desired probability, is the complement; that is

$$\rho_h = 1 - \left(1 - \frac{\rho_{h-1}}{2}\right)^2 = \frac{\rho_{h-1}}{2} \left(2 - \frac{\rho_{h-1}}{2}\right) = \rho_{h-1} - \frac{\rho_{h-1}^2}{4} = f(\rho_{h-1}) \quad \text{for} \quad f(x) = x - x^2/4.$$

The starting values are $\rho_0 = 1$, and $\rho_1 = 3/4$. Formally, we have the sequence:

$$\rho_0 = 1, \quad \rho_1 = 3/4, \quad \rho_h = \rho_{h-1} - \frac{\rho_{h-1}^2}{4}.$$

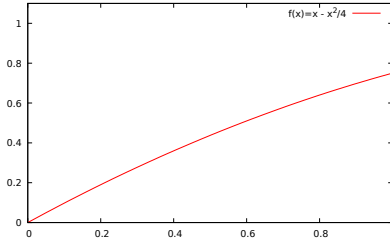


Figure 12.1: A graph of the function $f(x) = x - x^2/4$.

Lemma 12.1.1. *We have that $\rho_h \geq 1/(h+1)$.*

Proof: The proof is by induction. For $h = 1$, we have $\rho_1 = 3/4 \geq 1/(1+1)$.

Observe that $\rho_h = f(\rho_{h-1})$ for $f(x) = x - x^2/4$, and $f'(x) = 1 - x/2$. As such, $f'(x) > 0$ for $x \in [0, 1]$ and $f(x)$ is increasing in the range $[0, 1]$. As such, by induction, we have that $\rho_h = f(\rho_{h-1}) \geq f\left(\frac{1}{(h-1)+1}\right) = \frac{1}{h} - \frac{1}{4h^2}$.

We need to prove that $\rho_h \geq 1/(h+1)$, which is implied by the above if

$$\frac{1}{h} - \frac{1}{4h^2} \geq \frac{1}{h+1} \quad \Leftrightarrow \quad 4h(h+1) - (h+1) \geq 4h^2 \quad \Leftrightarrow \quad 4h^2 + 4h - h - 1 \geq 4h^2 \quad \Leftrightarrow \quad 3h \geq 1,$$

which trivially holds. ■

One can also prove an upper bound on this probability, showing that $\rho_h = \Theta(1/h)$. We provide the proof here for the sake of completeness, but the reader is encouraged to skip reading its proof, as we do not need this result.

Lemma 12.1.2. *We have that $\rho_h = O(1/h)$.*

Proof: The claim trivially holds for small values of h . For any $j > 0$, let h_j be the minimal index such that $\rho_{h_j} \leq 1/2^j$. It is easy to verify that $\rho_{h_j} \geq 1/2^{j+1}$. We claim (mysteriously) that $h_{j+1} - h_j \leq \frac{\rho_{h_j} - \rho_{h_{j+1}}}{(\rho_{h_{j+1}})^2/4}$. Indeed, ρ_{k+1} is the number resulting from removing $\rho_k^2/4$ from ρ_k . Namely, the sequence ρ_1, ρ_2, \dots is a monotonically decreasing sequence of numbers in the interval $[0, 1]$, where the gaps between consecutive numbers decreases.

In particular, to get from ρ_{h_j} to $\rho_{h_{j+1}}$, the gaps used were of size at least $\Delta = (\rho_{h_{j+1}})^2$, which means that there are at least $(\rho_{h_j} - \rho_{h_{j+1}})/\Delta - 1$ numbers in the series between these two elements. As such, we have

$$h_{j+1} - h_j \leq \frac{\rho_{h_j} - \rho_{h_{j+1}}}{(\rho_{h_{j+1}})^2/4} \leq \frac{1/2^j - 1/2^{j+2}}{1/2^{2(j+2)+2}} = 2^{j+6} + 2^{j+4} = O(2^j).$$

Arguing similarly, we have

$$h_{j+2} - h_j \geq \frac{\rho_{h_j} - \rho_{h_{j+2}}}{(\rho_{h_j})^2/4} \geq \frac{1/2^{j+1} - 1/2^{j+2}}{1/2^{2j+2}} = 2^{j+1} + 2^j = \Omega(2^j).$$

We conclude that $h_j = (h_j - h_{j-2}) + (h_{j-2} - h_{j-4}) + \dots = 2^{j-1} - O(1)$, implying the claim. ■

12.2. Min Cut

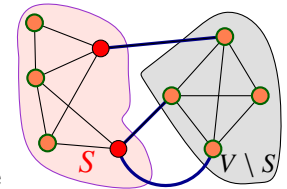
12.2.1. Problem Definition

Let $G = (V, E)$ be an undirected graph with n vertices and m edges. We are interested in **cuts** in G .

Definition 12.2.1. A **cut** in G is a partition of the vertices of V into two sets S and $V \setminus S$, where the edges of the cut are

$$(S, V \setminus S) = \{uv \in E \mid u \in S, v \in V \setminus S\},$$

where $S \neq \emptyset$ and $V \setminus S \neq \emptyset$. The number of edges in the cut $(S, V \setminus S)$ is the **size** of the cut. For an example of a cut, see figure on the right.



We are interested in the problem of computing the **minimum cut** (i.e., **mincut**), that is, the cut in the graph with minimum cardinality. Specifically, we would like to find the set $S \subseteq V$ such that $(S, V \setminus S)$ is as small as possible, and S is neither empty nor $V \setminus S$ is empty.

12.2.2. Some Definitions

We remind the reader of the following concepts. The **conditional probability** of X given Y is $\mathbb{P}[X = x \mid Y = y] = \mathbb{P}[(X = x) \cap (Y = y)]/\mathbb{P}[Y = y]$. An equivalent, useful restatement of this is that

$$\mathbb{P}[(X = x) \cap (Y = y)] = \mathbb{P}[X = x \mid Y = y] \cdot \mathbb{P}[Y = y]. \quad (12.1)$$

The following is easy to prove by induction using Eq. (12.1).

Lemma 12.2.2. *Let $\mathcal{E}_1, \dots, \mathcal{E}_n$ be n events which are not necessarily independent. Then,*

$$\mathbb{P}[\bigcap_{i=1}^n \mathcal{E}_i] = \mathbb{P}[\mathcal{E}_1] * \mathbb{P}[\mathcal{E}_2 \mid \mathcal{E}_1] * \mathbb{P}[\mathcal{E}_3 \mid \mathcal{E}_1 \cap \mathcal{E}_2] * \dots * \mathbb{P}[\mathcal{E}_n \mid \mathcal{E}_1 \cap \dots \cap \mathcal{E}_{n-1}].$$

12.3. The Algorithm

The basic operation used by the algorithm is **edge contraction**, depicted in Figure 12.2. We take an edge $e = xy$ in G and merge the two vertices into a single vertex. The new resulting graph is denoted by G/xy . Note, that we remove self loops created by the contraction. However, since the resulting graph is no longer a regular graph, it has parallel edges – namely, it is a multi-graph. We represent a multi-graph, as a regular graph with multiplicities on the edges. See Figure 12.3.

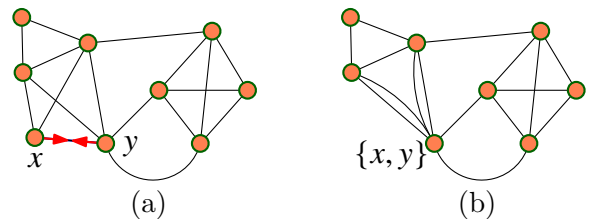


Figure 12.2: (a) A contraction of the edge xy . (b) The resulting graph.

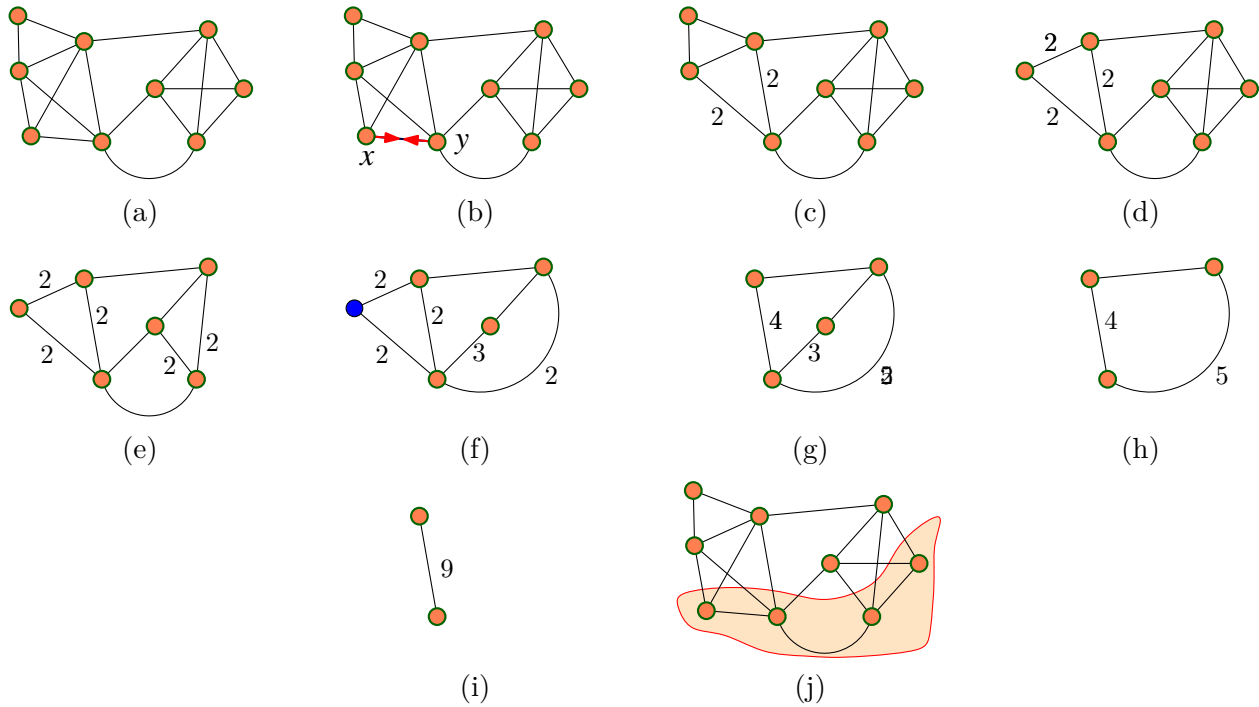


Figure 12.4: (a) Original graph. (b)–(j) a sequence of contractions in the graph, and (h) the cut in the original graph, corresponding to the single edge in (h). Note that the cut of (h) is not a mincut in the original graph.

The edge contraction operation can be implemented in $O(n)$ time for a graph with n vertices. This is done by merging the adjacency lists of the two vertices being contracted, and then using hashing to do the fix-ups (i.e., we need to fix the adjacency list of the vertices that are connected to the two vertices).

Note, that the cut is now computed counting multiplicities (i.e., if e is in the cut and it has weight w , then the contribution of e to the cut weight is w).

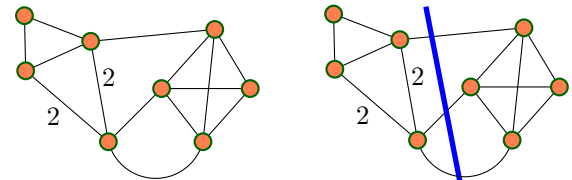


Figure 12.3: On the left a multi-graph, and on the right a minimum cut in the resulting multi-graph.

Observation 12.3.1. *A set of vertices in G/xy corresponds to a set of vertices in the graph G . Thus a cut in G/xy always corresponds to a valid cut in G . However, there are cuts in G that do not exist in G/xy . For example, the cut $S = \{x\}$, does not exist in G/xy . As such, the size of the minimum cut in G/xy is at least as large as the minimum cut in G (as long as G/xy has at least one edge). Since any cut in G/xy has a corresponding cut of the same cardinality in G .*

Our algorithm works by repeatedly performing edge contractions. This is beneficial as this shrinks the underlying graph, and we would compute the cut in the resulting (smaller) graph. An “extreme” example of this, is shown in Figure 12.4, where we contract the graph into a single edge, which (in turn) corresponds to a cut in the original graph. (It might help the reader to think about each vertex in the contracted graph, as corresponding to a connected component in the original graph.)

Figure 12.4 also demonstrates the problem with taking this approach. Indeed, the resulting cut is not the minimum cut in the graph.

So, why did the algorithm fail to find the minimum cut in this case?^① The failure occurs because of the contraction at Figure 12.4 (e), as we had contracted an edge in the minimum cut. In the new graph, depicted

^①Naturally, if the algorithm had succeeded in finding the minimum cut, this would have been our success.

```

Algorithm MinCut(G)
  G0 ← G
  i = 0
  while Gi has more than two vertices do
    Pick randomly an edge ei from the edges of Gi
    Gi+1 ← Gi/ei
    i ← i + 1
  Let (S, V \ S) be the cut in the original graph
    corresponding to the single edge in Gi
  return (S, V \ S).

```

Figure 12.5: The minimum cut algorithm.

in Figure 12.4 (f), there is no longer a cut of size 3, and all cuts are of size 4 or more. Specifically, the algorithm succeeds only if it does not contract an edge in the minimum cut.

Observation 12.3.2. *Let e_1, \dots, e_{n-2} be a sequence of edges in G , such that none of them is in the minimum cut, and such that $G' = G/\{e_1, \dots, e_{n-2}\}$ is a single multi-edge. Then, this multi-edge corresponds to a minimum cut in G .*

Note, that the claim in the above observation is only in one direction. We might be able to still compute a minimum cut, even if we contract an edge in a minimum cut, the reason being that a minimum cut is not unique. In particular, another minimum cut might survived the sequence of contractions that destroyed other minimum cuts.

Using Observation 12.3.2 in an algorithm is problematic, since the argumentation is circular, how can we find a sequence of edges that are not in the cut without knowing what the cut is? The way to slice the Gordian knot here, is to randomly select an edge at each stage, and contract this random edge.

See Figure 12.5 for the resulting algorithm **MinCut**.

12.3.1. Analysis

12.3.1.1. The probability of success

Naturally, if we are extremely lucky, the algorithm would never pick an edge in the mincut, and the algorithm would succeed. The ultimate question here is what is the probability of success. If it is relatively “large” then this algorithm is useful since we can run it several times, and return the best result computed. If on the other hand, this probability is tiny, then we are working in vain since this approach would not work.

Lemma 12.3.3. *If a graph G has a minimum cut of size k and G has n vertices, then $|E(G)| \geq kn/2$.*

Proof: Each vertex degree is at least k , otherwise the vertex itself would form a minimum cut of size smaller than k . As such, there are at least $\sum_{v \in V} \text{degree}(v)/2 \geq nk/2$ edges in the graph. ■

Lemma 12.3.4. *If we pick in random an edge e from a graph G , then with probability at most $2/n$ it belong to the minimum cut.*

Proof: There are at least $nk/2$ edges in the graph and exactly k edges in the minimum cut. Thus, the probability of picking an edge from the minimum cut is smaller then $k/(nk/2) = 2/n$. ■

The following lemma shows (surprisingly) that **MinCut** succeeds with reasonable probability.

Lemma 12.3.5. ***MinCut** outputs the mincut with probability $\geq \frac{2}{n(n-1)}$.*

Proof: Let \mathcal{E}_i be the event that e_i is not in the minimum cut of G_i . By [Observation 12.3.2](#), [MinCut](#) outputs the minimum cut if the events $\mathcal{E}_0, \dots, \mathcal{E}_{n-3}$ all happen (namely, all edges picked are outside the minimum cut).

By [Lemma 12.3.4](#), it holds $\mathbb{P}[\mathcal{E}_i \mid \mathcal{E}_0 \cap \mathcal{E}_1 \cap \dots \cap \mathcal{E}_{i-1}] \geq 1 - \frac{2}{|V(G_i)|} = 1 - \frac{2}{n-i}$. Implying that

$$\Delta = \mathbb{P}[\mathcal{E}_0 \cap \dots \cap \mathcal{E}_{n-3}] = \mathbb{P}[\mathcal{E}_0] \cdot \mathbb{P}[\mathcal{E}_1 \mid \mathcal{E}_0] \cdot \mathbb{P}[\mathcal{E}_2 \mid \mathcal{E}_0 \cap \mathcal{E}_1] \cdot \dots \cdot \mathbb{P}[\mathcal{E}_{n-3} \mid \mathcal{E}_0 \cap \dots \cap \mathcal{E}_{n-4}].$$

As such, we have

$$\Delta \geq \prod_{i=0}^{n-3} \left(1 - \frac{2}{n-i}\right) = \prod_{i=0}^{n-3} \frac{n-i-2}{n-i} = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \dots \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)}. \quad \blacksquare$$

12.3.1.2. Running time analysis.

Observation 12.3.6. *MinCut runs in $O(n^2)$ time.*

Observation 12.3.7. *The algorithm always outputs a cut, and the cut is not smaller than the minimum cut.*

Informally, [amplification](#) is the process of running an experiment again and again till the things we want to happen, with good probability, do happen.

Let [MinCutRep](#) be the algorithm that runs [MinCut](#) $n(n-1)$ times and return the minimum cut computed in all those independent executions of [MinCut](#).

Lemma 12.3.8. *The probability that [MinCutRep](#) fails to return the minimum cut is < 0.14 .*

Proof: The probability of failure of [MinCut](#) to output the mincut in each execution is at most $1 - \frac{2}{n(n-1)}$, by [Lemma 12.3.5](#). Now, [MinCutRep](#) fails, only if all the $n(n-1)$ executions of [MinCut](#) fail. But these executions are independent, as such, the probability to this happen is at most

$$\left(1 - \frac{2}{n(n-1)}\right)^{n(n-1)} \leq \exp\left(-\frac{2}{n(n-1)} \cdot n(n-1)\right) = \exp(-2) < 0.14,$$

since $1 - x \leq e^{-x}$ for $0 \leq x \leq 1$. \(\blacksquare\)

Theorem 12.3.9. *One can compute the minimum cut in $O(n^4)$ time with constant probability to get a correct result. In $O(n^4 \log n)$ time the minimum cut is returned with high probability.*

12.4. A faster algorithm

The algorithm presented in the previous section is extremely simple. Which raises the question of whether we can get a faster algorithm^②?

So, why [MinCutRep](#) needs so many executions? Well, the probability of success in the first ν iterations is

$$\mathbb{P}[\mathcal{E}_0 \cap \dots \cap \mathcal{E}_{\nu-1}] \geq \prod_{i=0}^{\nu-1} \left(1 - \frac{2}{n-i}\right) = \prod_{i=0}^{\nu-1} \frac{n-i-2}{n-i} = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \dots = \frac{(n-\nu)(n-\nu-1)}{n \cdot (n-1)}. \quad (12.2)$$

Namely, this probability deteriorates very quickly toward the end of the execution, when the graph becomes small enough. (To see this, observe that for $\nu = n/2$, the probability of success is roughly $1/4$, but for $\nu = n - \sqrt{n}$ the probability of success is roughly $1/n$.)

So, the key observation is that as the graph get smaller the probability to make a bad choice increases. So, instead of doing the amplification from the outside of the algorithm, we will run the new algorithm more times when the graph is smaller. Namely, we put the amplification directly into the algorithm.

The basic new operation we use is [Contract](#), depicted in [Figure 12.6](#), which also depict the new algorithm [FastCut](#).

^②This would require a more involved algorithm, that's life.

```

Contract ( G, t )
begin
  while |V(G)| > t do
    Pick a random edge e in G.
    G ← G/e
  return G
end

```

```

FastCut(G = (V, E))
  G – multi-graph
begin
  n ← |V(G)|
  if n ≤ 6 then
    Compute (via brute force) minimum cut
    of G and return cut.
  t ← ⌈1 + n/√2⌉
  H1 ← Contract(G, t)
  H2 ← Contract(G, t)
  /* Contract is randomized!!! */
  X1 ← FastCut(H1),
  X2 ← FastCut(H2)
  return minimum cut out of X1 and X2.
end

```

Figure 12.6: **Contract**(G, t) shrinks G till it has only t vertices. **FastCut** computes the minimum cut using **Contract**.

Lemma 12.4.1. *The running time of **FastCut**(G) is $O(n^2 \log n)$, where $n = |V(G)|$.*

Proof: Well, we perform two calls to **Contract**(G, t) which takes $O(n^2)$ time. And then we perform two recursive calls on the resulting graphs. We have

$$T(n) = O(n^2) + 2T(n/\sqrt{2}).$$

The solution to this recurrence is $O(n^2 \log n)$ as one can easily (and should) verify. ■

Exercise 12.4.2. *Show that one can modify **FastCut** so that it uses only $O(n^2)$ space.*

Lemma 12.4.3. *The probability that **Contract**(G, $n/\sqrt{2}$) had not contracted the minimum cut is at least 1/2.*

Namely, the probability that the minimum cut in the contracted graph is still a minimum cut in the original graph is at least 1/2.

Proof: Just plug in $v = n - t = n - \lceil 1 + n/\sqrt{2} \rceil$ into Eq. (12.2). We have

$$\mathbb{P}[\mathcal{E}_0 \cap \dots \cap \mathcal{E}_{n-t}] \geq \frac{t(t-1)}{n \cdot (n-1)} = \frac{\lceil 1 + n/\sqrt{2} \rceil (\lceil 1 + n/\sqrt{2} \rceil - 1)}{n(n-1)} \geq \frac{1}{2}. \quad \blacksquare$$

The following lemma bounds the probability of success.

Lemma 12.4.4. ***FastCut** finds the minimum cut with probability larger than $\Omega(1/\log n)$.*

Proof: Let T_h be the recursion tree of the algorithm of depth $h = \Theta(\log n)$. Color an edge of recursion tree by black if the contraction succeeded. Clearly, the algorithm succeeds if there is a path from the root to a leaf that is all black. This is exactly the settings of Lemma 12.1.1, and we conclude that the probability of success is at least $1/(h+1) = \Theta(1/\log n)$, as desired. ■

Exercise 12.4.5. *Prove, that running **FastCut** repeatedly $c \cdot \log^2 n$ times, guarantee that the algorithm outputs the minimum cut with probability $\geq 1 - 1/n^2$, say, for c a constant large enough.*

Theorem 12.4.6. *One can compute the minimum cut in a graph G with n vertices in $O(n^2 \log^3 n)$ time. The algorithm succeeds with probability $\geq 1 - 1/n^2$.*

Proof: We do amplification on **FastCut** by running it $O(\log^2 n)$ times. The running time bound follows from **Lemma 12.4.1**. The bound on the probability follows from **Lemma 12.4.4**, and using the amplification analysis as done in **Lemma 12.3.8** for **MinCutRep**. ■

12.5. Bibliographical Notes

The **MinCut** algorithm was developed by David Karger during his PhD thesis in Stanford. The fast algorithm is a joint work with Clifford Stein. The basic algorithm of the mincut is described in [MR95, pages 7–9], the faster algorithm is described in [MR95, pages 289–295].

Galton-Watson process. The idea of using coloring of the edges of a tree to analyze **FastCut** might be new (i.e., **Section 12.1.2**).

Part V

Network flow

Chapter 13

Network Flow

13.1. Network Flow

We would like to transfer as much “merchandise” as possible from one point to another. For example, we have a wireless network, and one would like to transfer a large file from s to t . The network have limited capacity, and one would like to compute the maximum amount of information one can transfer.

Specifically, there is a network and capacities associated with each connection in the network. The question is how much “flow” can you transfer from a source s into a sink t . Note, that here we think about the flow as being splittable, so that it can travel from the source to the sink along several parallel paths simultaneously. So, think about our network as being a network of pipe moving water from the source the sink (the capacities are how much water can a pipe transfer in a given unit of time). On the other hand, in the internet traffic is packet based and splitting is less easy to do.

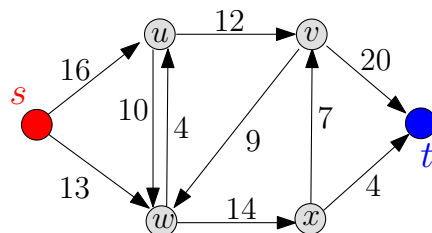
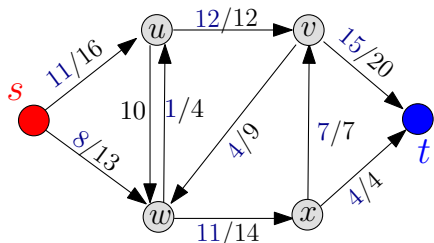


Figure 13.1: A network flow.

Definition 13.1.1. Let $G = (V, E)$ be a *directed* graph. For every edge $(u, v) \in E(G)$ we have an associated edge *capacity* $c(u, v)$, which is a non-negative number. If the edge $(u, v) \notin G$ then $c(u, v) = 0$. In addition, there is a *source* vertex s and a target *sink* vertex t .

The entities G , s , t and $c(\cdot)$ together form a *flow network* or simply a *network*. An example of such a flow network is depicted in [Figure 13.1](#).



We would like to transfer as much flow from the source s to the sink t . Specifically, all the flow starts from the source vertex, and ends up in the sink. The flow on an edge is a non-negative quantity that can not exceed the capacity constraint for this edge. One possible flow is depicted on the left figure, where the numbers a/b on an edge denote a flow of a units on an edge with capacity at most b .

We next formalize our notation of a flow.

Definition 13.1.2 (flow). A **flow** in a network is a function $f(\cdot, \cdot)$ on the edges of G such that:

(A) **Bounded by capacity**: For any edge $(u, v) \in E$, we have $f(u, v) \leq c(u, v)$.

Specifically, the amount of flow between u and v on the edge (u, v) never exceeds its capacity $c(u, v)$.

(B) **Anti symmetry**: For any u, v we have $f(u, v) = -f(v, u)$.

(C) There are two special vertices: (i) the **source** vertex s (all flow starts from the source), and the **sink** vertex t (all the flow ends in the sink).

(D) **Conservation of flow**: For any vertex $u \in V \setminus \{s, t\}$, we have $\sum_v f(u, v) = 0$.^① Namely, for any internal node, all the flow that flows into a vertex leaves this vertex.

The amount of flow (or simply **flow**) of f , called the **value** of f , is $|f| = \sum_{v \in V} f(s, v)$.

Note, that a flow on an edge can be negative (i.e., there is a positive flow flowing on this edge in the other direction).

Problem 13.1.3 (Maximum flow). Given a network G find the **maximum flow** in G . Namely, compute a legal flow f such that $|f|$ is maximized.

13.2. Some properties of flows and residual networks

For two sets $X, Y \subseteq V$, let $f(X, Y) = \sum_{x \in X, y \in Y} f(x, y)$. We will slightly abuse the notations and refer to $f(\{v\}, S)$ by $f(v, S)$, where $v \in V(G)$.

Observation 13.2.1. *By definition, we have $|f| = f(s, V)$.*

Lemma 13.2.2. *For a flow f , the following properties holds:*

(i) $\forall u \in V(G)$ we have $f(u, u) = 0$,

(ii) $\forall X \subseteq V$ we have $f(X, X) = 0$,

(iii) $\forall X, Y \subseteq V$ we have $f(X, Y) = -f(Y, X)$,

(iv) $\forall X, Y, Z \subseteq V$ such that $X \cap Y = \emptyset$ we have that $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ and $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$.

(v) For all $u \in V \setminus \{s, t\}$, we have $f(u, V) = f(V, u) = 0$.

Proof: Property (i) holds since (u, u) it not an edge in the graph, and as such its flow is zero. As for property (ii), we have

$$f(X, X) = \sum_{\{u, v\} \subseteq X, u \neq v} (f(u, v) + f(v, u)) + \sum_{u \in X} f(u, u) = \sum_{\{u, v\} \subseteq X, u \neq v} (f(u, v) - f(u, v)) + \sum_{u \in X} 0 = 0,$$

by the anti-symmetry property of flow (Definition 13.1.2 (B)).

Property (iii) holds immediately by the anti-symmetry of flow, as

$$f(X, Y) = \sum_{x \in X, y \in Y} f(x, y) = - \sum_{x \in X, y \in Y} f(y, x) = -f(Y, X).$$

(iv) This case follows immediately from definition.

Finally (v) is a restatement of the conservation of flow property. ■

Claim 13.2.3. $|f| = f(V, t)$.

^①This law for electric circuits is known as Kirchhoff's Current Law.



Figure 13.2: (i) A flow network, and (ii) the resulting residual network. Note, that $f(u, w) = -f(w, u) = -1$ and as such $c_f(u, w) = 10 - (-1) = 11$.

Proof: We have:

$$\begin{aligned}
 |f| &= f(s, V) = f(V \setminus (V \setminus \{s\}), V) \\
 &= f(V, V) - f(V \setminus \{s\}, V) \\
 &= -f(V \setminus \{s\}, V) = f(V, V \setminus \{s\}) \\
 &= f(V, t) + f(V, V \setminus \{s, t\}) \\
 &= f(V, t) + \sum_{u \in V \setminus \{s, t\}} f(V, u) \\
 &= f(V, t) + \sum_{u \in V \setminus \{s, t\}} 0 \\
 &= f(V, t),
 \end{aligned}$$

since $f(V, V) = 0$ by Lemma 13.2.2 (i) and $f(V, u) = 0$ by Lemma 13.2.2 (iv). ■

Definition 13.2.4. Given capacity c and flow f , the **residual capacity** of an edge (u, v) is

$$c_f(u, v) = c(u, v) - f(u, v).$$

Intuitively, the residual capacity $c_f(u, v)$ on an edge (u, v) is the amount of unused capacity on (u, v) . We can next construct a graph with all edges that are not being fully used by f , and as such can serve to improve f .

Definition 13.2.5. Given f , $G = (V, E)$ and c , as above, the **residual graph** (or **residual network**) of G and f is the graph $G_f = (V, E_f)$ where

$$E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}.$$

Note, that by the definition of E_f , it might be that an edge (u, v) that appears in E might induce two edges in E_f . Indeed, consider an edge (u, v) such that $f(u, v) < c(u, v)$ and (v, u) is not an edge of G . Clearly, $c_f(u, v) = c(u, v) - f(u, v) > 0$ and $(u, v) \in E_f$. Also,

$$c_f(v, u) = c(v, u) - f(v, u) = 0 - (-f(u, v)) = f(u, v),$$

since $c(v, u) = 0$ as (v, u) is not an edge of G . As such, $(v, u) \in E_f$. This states that we can always reduce the flow on the edge (u, v) and this is interpreted as pushing flow on the edge (v, u) . See Figure 13.2 for an example of a residual network.

Since every edge of G induces at most two edges in G_f , it follows that G_f has at most twice the number of edges of G ; formally, $|E_f| \leq 2|E|$.

Lemma 13.2.6. *Given a flow f defined over a network G , then the residual network G_f together with c_f form a flow network.*

Proof: One need to verify that $c_f(\cdot)$ is always a non-negative function, which is true by the definition of E_f . ■

The following lemma testifies that we can improve a flow f on G by finding a any legal flow h in the residual network G_f .

Lemma 13.2.7. *Given a flow network $G = (V, E)$, a flow f in G , and a flow h in G_f , where G_f is the residual network of f . Then $f + h$ is a (legal) flow in G and its capacity is $|f + h| = |f| + |h|$.*

Proof: By definition, we have $(f + h)(u, v) = f(u, v) + h(u, v)$ and thus $(f + h)(X, Y) = f(X, Y) + h(X, Y)$. We need to verify that $f + h$ is a legal flow, by verifying the properties required to it by **Definition 13.1.2**.

Anti symmetry holds since $(f + h)(u, v) = f(u, v) + h(u, v) = -f(v, u) - h(v, u) = -(f + h)(v, u)$.

Next, we verify that the flow $f + h$ is bounded by capacity. Indeed,

$$(f + h)(u, v) \leq f(u, v) + h(u, v) \leq f(u, v) + c_f(u, v) = f(u, v) + (c(u, v) - f(u, v)) = c(u, v).$$

For $u \in V - s - t$ we have $(f + h)(u, V) = f(u, V) + h(u, V) = 0 + 0 = 0$ and as such $f + h$ comply with the conservation of flow requirement.

Finally, the total flow is

$$|f + h| = (f + h)(s, V) = f(s, V) + h(s, V) = |f| + |h|. \quad \blacksquare$$

Definition 13.2.8. For G and a flow f , a path π in G_f between s and t is an **augmenting path**.

Note, that all the edges of π has positive capacity in G_f , since otherwise (by definition) they would not appear in E_f . As such, given a flow f and an augmenting path π , we can improve f by pushing a positive amount of flow along the augmenting path π . An augmenting path is depicted on the right, for the network flow of **Figure 13.2**.

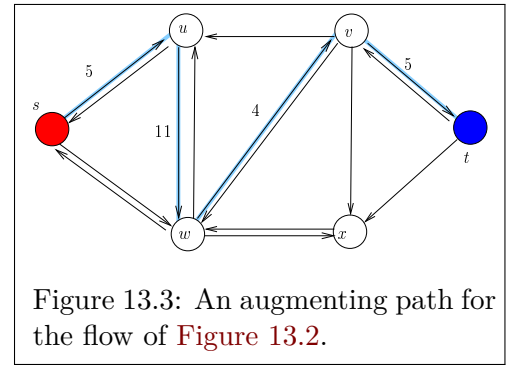


Figure 13.3: An augmenting path for the flow of **Figure 13.2**.

Definition 13.2.9. For an augmenting path π let $c_f(\pi)$ be the maximum amount of flow we can push through π . We call $c_f(\pi)$ the **residual capacity** of π . Formally,

$$c_f(\pi) = \min_{(u,v) \in \pi} c_f(u, v).$$

We can now define a flow that realizes the flow along π . Indeed:

$$f_\pi(u, v) = \begin{cases} c_f(\pi) & \text{if } (u, v) \text{ is in } \pi \\ -c_f(\pi) & \text{if } (v, u) \text{ is in } \pi \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 13.2.10. *For an augmenting path π , the flow f_π is a flow in G_f and $|f_\pi| = c_f(\pi) > 0$.*

We can now use such a path to get a larger flow.

Lemma 13.2.11. *Let f be a flow, and let π be an augmenting path for f . Then $f + f_\pi$ is a “better” flow. Namely, $|f + f_\pi| = |f| + |f_\pi| > |f|$.*

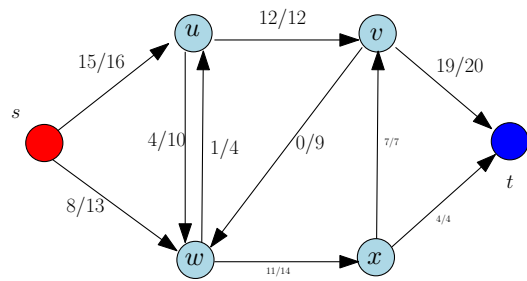
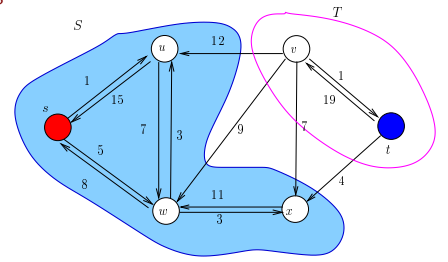


Figure 13.4: The flow resulting from applying the residual flow f_p of the path p of **Figure 13.3** to the flow of **Figure 13.2**.

Namely, $f + f_\pi$ is flow with larger value than f . Consider the flow in **Figure 13.4**.

Can we continue improving it? Well, if you inspect the residual network of this flow, depicted on the right. Observe that s is disconnected from t in this residual network. So, we are unable to push any more flow. Namely, we found a solution which is a local maximum solution for network flow. But is that a global maximum? Is this the maximum flow we are looking for?



13.3. The Ford-Fulkerson method

```

mtdFordFulkerson(G, c)
begin
  f ← Zero flow on G
  while (Gf has augmenting
         path p) do
    (* Recompute Gf for
       this check *)
    f ← f + fp
  return f
end

```

Given a network G with capacity constraints c , the above discussion suggest a simple and natural method to compute a maximum flow. This is known as the **Ford-Fulkerson** method for computing maximum flow, and is depicted on the left, we will refer to it as the **mtdFordFulkerson** method.

It is unclear that this method (and the reason we do not refer to it as an algorithm) terminates and reaches the global maximum flow. We address these problems shortly.

13.4. On maximum flows

We need several natural concepts.

Definition 13.4.1. A **directed cut** (S, T) in a flow network $G = (V, E)$ is a partition of V into S and $T = V \setminus S$, such that $s \in S$ and $t \in T$. We usually will refer to a directed cut as being a **cut**.

The net **flow of f across a cut** (S, T) is $f(S, T) = \sum_{s \in S, t \in T} f(s, t)$.

The **capacity** of (S, T) is $c(S, T) = \sum_{s \in S, t \in T} c(s, t)$.

The **minimum cut** is the cut in G with the minimum capacity.

Lemma 13.4.2. Let G, f, s, t be as above, and let (S, T) be a cut of G . Then $f(S, T) = |f|$.

Proof: We have

$$f(S, T) = f(S, V) - f(S, S) = f(S, V) = f(s, V) + f(S - s, V) = f(s, V) = |f|,$$

since $T = V \setminus S$, and $f(S - s, V) = \sum_{u \in S - s} f(u, V) = 0$ by **Lemma 13.2.2 (v)** (note that u can not be t as $t \in T$). ■

Claim 13.4.3. The flow in a network is upper bounded by the capacity of any cut (S, T) in G .

Proof: Consider a cut (S, T) . We have $|f| = f(S, T) = \sum_{u \in S, v \in T} f(u, v) \leq \sum_{u \in S, v \in T} c(u, v) = c(S, T)$. ■

In particular, the maximum flow is bounded by the capacity of the minimum cut. Surprisingly, the maximum flow is exactly the value of the minimum cut.

Theorem 13.4.4 (Max-flow min-cut theorem). If f is a flow in a flow network $G = (V, E)$ with source s and sink t , then the following conditions are equivalent:

- (A) f is a maximum flow in G .
- (B) The residual network G_f contains no augmenting paths.

(C) $|f| = c(S,T)$ for some cut (S,T) of G . And (S,T) is a minimum cut in G .

Proof: (A) \Rightarrow (B): By contradiction. If there was an augmenting path p then $c_f(p) > 0$, and we can generate a new flow $f + f_p$, such that $|f + f_p| = |f| + c_f(p) > |f|$. A contradiction as f is a maximum flow.

(B) \Rightarrow (C): Well, it must be that s and t are disconnected in G_f . Let

$$S = \left\{ v \mid \text{Exists a path between } s \text{ and } v \text{ in } G_f \right\}$$

and $T = V \setminus S$. We have that $s \in S$, $t \in T$, and for any $u \in S$ and $v \in T$ we have $f(u,v) = c(u,v)$. Indeed, if there were $u \in S$ and $v \in T$ such that $f(u,v) < c(u,v)$ then $(u,v) \in E_f$, and v would be reachable from s in G_f , contradicting the construction of T .

This implies that $|f| = f(S,T) = c(S,T)$. The cut (S,T) must be a minimum cut, because otherwise there would be cut (S',T') with smaller capacity $c(S',T') < c(S,T) = f(S,T) = |f|$, On the other hand, by [Lemma 13.4.3](#), we have $|f| = f(S',T') \leq c(S',T')$. A contradiction.

(C) \Rightarrow (A) Well, for any cut (U,V) , we know that $|f| \leq c(U,V)$. This implies that if $|f| = c(S,T)$ then the flow can not be any larger, and it is thus a maximum flow. ■

The above max-flow min-cut theorem implies that if `mtdFordFulkerson` terminates, then it had computed the maximum flow. What is still allusive is showing that the `mtdFordFulkerson` method always terminates. This turns out to be correct only if we are careful about the way we pick the augmenting path.

Chapter 14

Network Flow II - The Vengeance

14.1. Accountability

The comic in [Figure 14.1](#) is by Jonathan Shewchuk and is referring to the Calvin and Hobbes comics.

People that do not know maximum flows: essentially everybody.

Average salary on earth $<$ \$5,000

People that know maximum flow - most of them work in programming related jobs and make at least \$10,000 a year.

Salary of people that learned maximum flows: $>$ \$10,000

Salary of people that did not learn maximum flows: $<$ \$5,000

Salary of people that know Latin: 0 (unemployed).

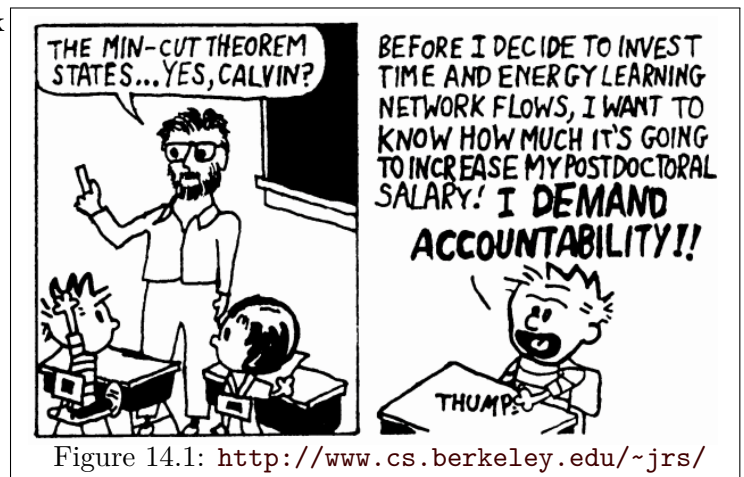


Figure 14.1: <http://www.cs.berkeley.edu/~jrs/>

Thus, by just learning maximum flows (and not knowing Latin) you can double your future salary!

14.2. The Ford-Fulkerson Method

The `mtdFordFulkerson` method is depicted on the right.

Lemma 14.2.1. *If the capacities on the edges of G are integers, then `mtdFordFulkerson` runs in $O(m|f^*|)$ time, where $|f^*|$ is the amount of flow in the maximum flow and $m = |E(G)|$.*

```

mtdFordFulkerson( $G, s, t$ )
  Initialize flow  $f$  to zero
  while  $\exists$  path  $\pi$  from  $s$  to  $t$  in  $G_f$  do
     $c_f(\pi) \leftarrow \min \{c_f(u, v) \mid (u, v) \in \pi\}$ 
    for  $\forall (u, v) \in \pi$  do
       $f(u, v) \leftarrow f(u, v) + c_f(\pi)$ 
       $f(v, u) \leftarrow f(v, u) - c_f(\pi)$ 

```

Proof: Observe that the `mtdFordFulkerson` method performs only subtraction, addition and min operations. Thus, if it finds an augmenting path π , then $c_f(\pi)$ must be a *positive* integer number. Namely, $c_f(\pi) \geq 1$. Thus, $|f^*|$ must be an integer number (by induction), and each iteration of the algorithm improves the flow by at least 1. It follows that after $|f^*|$ iterations the algorithm stops. Each iteration takes $O(m + n) = O(m)$ time, as can be easily verified. ■

The following observation is an easy consequence of our discussion.

Observation 14.2.2 (Integrality theorem). *If the capacity function c takes on only integral values, then the maximum flow f produced by the `mtdFordFulkerson` method has the property that $|f|$ is integer-valued. Moreover, for all vertices u and v , the value of $f(u, v)$ is also an integer.*

14.3. The Edmonds-Karp algorithm

The `Edmonds-Karp` algorithm works by modifying the `mtdFordFulkerson` method so that it always returns the shortest augmenting path in G_f (i.e., path with smallest number of edges). This is implemented by finding π using `BFS` in G_f .

Definition 14.3.1. For a flow f , let $\delta_f(v)$ be the length of the shortest path from the source s to v in the residual graph G_f . Each edge is considered to be of length 1.

We will shortly prove that, for any vertex $v \in V \setminus \{s, t\}$, the function $\delta_f(v)$, in the residual network G_f , increases monotonically with each flow augmentation. We delay proving this (key) technical fact (see [Lemma 14.3.5](#) below), and first show its implications.

Lemma 14.3.2. *During the execution of the `Edmonds-Karp` algorithm, an edge (u, v) might disappear (and thus reappear) from G_f at most $n/2$ times throughout the execution of the algorithm, where $n = |V(G)|$.*

Proof: Consider an iteration when the edge (u, v) disappears. Clearly, in this iteration the edge (u, v) appeared in the augmenting path π . Furthermore, this edge was fully utilized; namely, $c_f(\pi) = c_f(uv)$, where f is the flow in the beginning of the iteration when it disappeared. We continue running `Edmonds-Karp` till (u, v) “magically” reappears. This means that in the iteration before (u, v) reappeared in the residual graph, the algorithm handled an augmenting path σ that contained the *reverse* edge (v, u) . Let g be the flow used to compute σ . We have, by the monotonicity of $\delta(\cdot)$ [i.e., [Lemma 14.3.5](#) below], that

$$\delta_g(u) = \delta_g(v) + 1 \geq \delta_f(v) + 1 = \delta_f(u) + 2$$

as `Edmonds-Karp` is always augmenting along the shortest path. Namely, the distance of s to u had increased by 2 between its disappearance and its reappearance. Since $\delta_0(u) \geq 0$ and the maximum value of $\delta_f(u)$ is n , it follows that (u, v) can disappear and reappear at most $n/2$ times during the execution of the `Edmonds-Karp` algorithm. ■

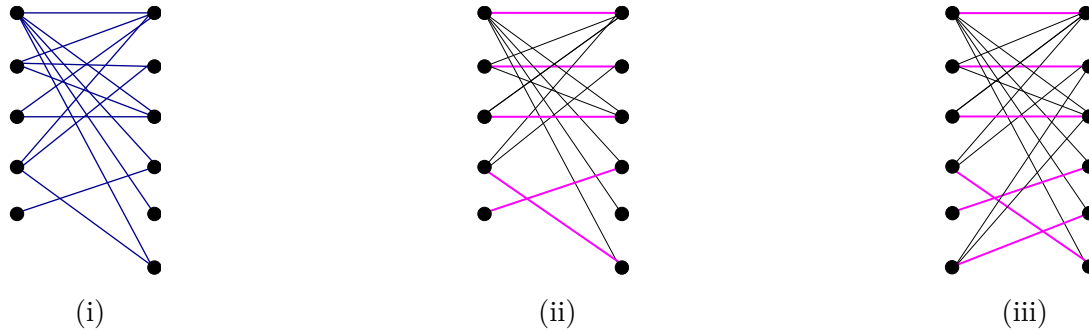


Figure 14.2: (i) A bipartite graph. (ii) A maximum matching in this graph. (iii) A perfect matching (in a different graph).

Observe that $\delta_f(u)$ might become infinity at some point during the algorithm execution (i.e., u is no longer reachable from s). If so, by monotonicity, the edge (u, v) would never appear again, in the residual graph, in any future iteration of the algorithm.

Observation 14.3.3. *Every time we add an augmenting path during the execution of the **Edmonds-Karp** algorithm, at least one edge disappears from the residual graph G_f . Indeed, every edge that realizes the residual capacity (of the augmenting path) will disappear once we push the maximum possible flow along this path.*

Lemma 14.3.4. *The **Edmonds-Karp** algorithm handles at most $O(nm)$ augmenting paths before it stops. Its running time is $O(nm^2)$, where $n = |V(G)|$ and $m = |E(G)|$.*

Proof: Every edge might disappear at most $n/2$ times during **Edmonds-Karp** execution, by **Lemma 14.3.2**. Thus, there are at most $nm/2$ edge disappearances during the execution of the **Edmonds-Karp** algorithm. At each iteration, we perform path augmentation, and at least one edge disappears along it from the residual graph. Thus, the **Edmonds-Karp** algorithm perform at most $O(mn)$ iterations.

Performing a single iteration of the algorithm boils down to computing an augmenting path. Computing such a path takes $O(m)$ time as we have to perform BFS to find the augmenting path. It follows, that the overall running time of the algorithm is $O(nm^2)$. ■

We still need to prove the aforementioned monotonicity property. (This is the only part in our discussion of network flow where the argument gets a bit tedious. So bear with us, after all, you are going to double your salary here.)

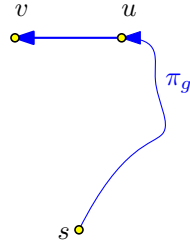
Lemma 14.3.5. *If the **Edmonds-Karp** algorithm is run on a flow network $G = (V, E)$ with source s and sink t , then for all vertices $v \in V \setminus \{s, t\}$, the shortest path distance $\delta_f(v)$ in the residual network G_f increases monotonically with each flow augmentation.*

Proof: Assume, for the sake of contradiction, that this is false. Consider the flow just after the first iteration when this claim failed. Let f denote the flow before this (fatal) iteration was performed, and let g be the flow after.

Let v be the vertex such that $\delta_g(v)$ is minimal, among all vertices for which the monotonicity fails. Formally, this is the vertex v where $\delta_g(v)$ is minimal and

$$\delta_g(v) < \delta_f(v). \quad (*)$$

Let $\pi_g = s \rightarrow \dots \rightarrow u \rightarrow v$ be the shortest path in G_g from s to v . Clearly, $(u, v) \in E(G_g)$, and thus $\delta_g(u) = \delta_g(v) - 1$.



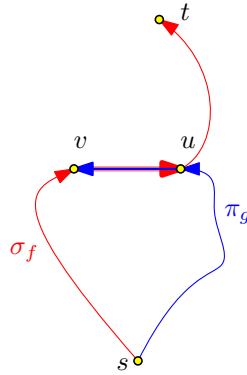
By the choice of v it must be that $\delta_g(u) \geq \delta_f(u)$, since otherwise the monotonicity property fails for u , and u is closer to s than v in G_g , and this, in turn, contradicts our choice of v as being the closest vertex to s that fails the monotonicity property. There are now two possibilities:

(i) If $(u, v) \in E(G_f)$ then

$$\delta_f(v) \leq \delta_f(u) + 1 \leq \delta_g(u) + 1 = \delta_g(v) - 1 + 1 = \delta_g(v).$$

This contradicts our assumptions that $\delta_f(v) > \delta_g(v)$.

(ii) If (u, v) is not in $E(G_f)$ then the augmenting path σ_f used in computing g from f contains the edge (v, u) . Indeed, the edge (u, v) reappeared in the residual graph G_g (while not being present in G_f). The only way this can happen is if the augmenting path σ_f pushed a flow in the other direction on the edge (u, v) . Namely, $(v, u) \in \sigma_f$.



However, the algorithm always augment along the shortest path. We have that

$$\delta_f(u) = \delta_f(v) + 1 \underbrace{>}_{(*)} \delta_g(v) + 1 > \delta_g(v) = \delta_g(u) + 1,$$

by the definition of u . Thus, $\delta_f(u) > \delta_g(u)$ (i.e., the monotonicity property fails for u) and $\delta_g(u) < \delta_g(v)$. A contradiction to the choice of v . ■

14.4. Applications and extensions for Network Flow

14.4.1. Maximum Bipartite Matching

Definition 14.4.1. For an undirected graph $G = (V, E)$ a **matching** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v .

A **maximum matching** is a matching M such that for any matching M' we have $|M| \geq |M'|$.

A matching is **perfect** if it involves all vertices. See Figure 14.2 for examples of these definitions.

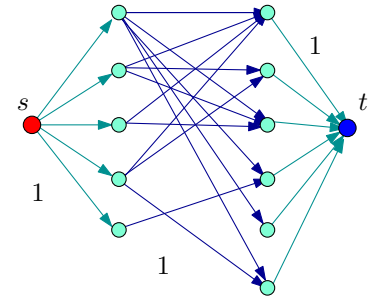


Figure 14.3

Theorem 14.4.2. One can compute maximum bipartite matching using network flow in $O(nm)$ time, for a bipartite graph with n vertices and m edges.

Proof: Given a bipartite graph G , we create a new graph with a new source on the left side and sink on the right, see Figure 14.3.

Direct all edges from left to right and set the capacity of all edges to 1. Let H be the resulting flow network. It is now easy to verify that by the Integrality theorem, a flow in H is either 0 or one on every edge, and thus a flow of value k in H is just a collection of k vertex disjoint paths between s and t in H , which corresponds to a matching in G of size k .

Similarly, given a matching of size k in G , it can be easily interpreted as realizing a flow in H of size k . Thus, computing a maximum flow in H results in computing a maximum matching in G . The running time of the algorithm is $O(nm^2)$. ■

14.4.2. Extension: Multiple Sources and Sinks

Given a flow network with several sources and sinks, how can we compute maximum flow on such a network?

The idea is to create a super source, that send all its flow to the old sources and similarly create a super sink that receives all the flow. See Figure 14.4. Clearly, computing flow in both networks is equivalent.

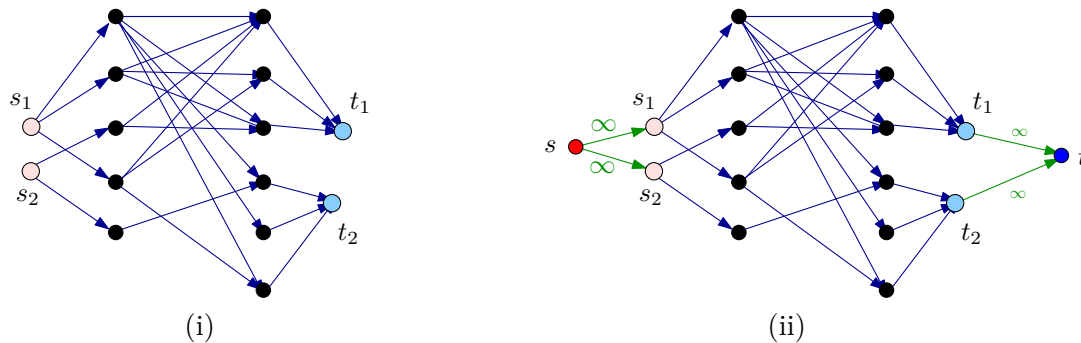


Figure 14.4: (i) A flow network with several sources and sinks, and (ii) an equivalent flow network with a single source and sink.

Chapter 15

Network Flow III - Applications

15.1. Edge disjoint paths

15.1.1. Edge-disjoint paths in a directed graphs

Question 15.1.1. *Given a graph G (either directed or undirected), two vertices s and t , and a parameter k , the task is to compute k paths from s to t in G , such that they are **edge disjoint**; namely, these paths do not share an edge.*

To solve this problem, we will convert G (assume G is a directed graph for the time being) into a network flow graph J , such that every edge has capacity 1. Find the maximum flow in J (between s and t). We claim that the value of the maximum flow in the network J , is equal to the number of edge disjoint paths in G .

Lemma 15.1.2. *If there are k edge disjoint paths in G between s and t , then the maximum flow value in J is at least k .*

Proof: Given k such edge disjoint paths, push one unit of flow along each such path. The resulting flow is legal in h and it has value k . ■

Definition 15.1.3 (0/1-flow). A flow f is a **0/1-flow** if every edge has either no flow on it, or one unit of flow.

Lemma 15.1.4. *Let f be a 0/1 flow in a network J with flow value μ . Then there are μ edge disjoint paths between s and t in J .*

Proof: By induction on the number of edges in J that has one unit of flow assigned to them by f . If $\mu = 0$ then there is nothing to prove.

Otherwise, start traversing the graph J from s traveling only along edges with flow 1 assigned to them by f . We mark such an edge as used, and do not allow one to travel on such an edge again. There are two possibilities:

(i) We reached the target vertex t . In this case, we take this path, add it to the set of output paths, and reduce the flow along the edges of the generated path π to 0. Let H' be the resulting flow network and f' the resulting flow. We have $|f'| = \mu - 1$, H' has less edges, and by induction, it has $\mu - 1$ edge disjoint paths in H' between s and t . Together with π this forms μ such paths.

(ii) We visit a vertex v for the second time. In this case, our traversal contains a cycle C , of edges in J that have flow 1 on them. We set the flow along the edges of C to 0 and use induction on the remaining graph (since it has less edges with flow 1 on them). The value of the flow f did not change by removing C , and as such it follows by induction that there are μ edge disjoint paths between s and t in J . ■

Since the graph G is simple, there are at most $n = |V(J)|$ edges that leave s . As such, the maximum flow in J is $\leq n$. Thus, applying the Ford-Fulkerson algorithm, takes $O(mn)$ time. The extraction of the paths can also be done in linear time by applying the algorithm in the proof of **Lemma 15.1.4**. As such, we get:

Theorem 15.1.5. *Given a directed graph G with n vertices and m edges, and two vertices s and t , one can compute the maximum number of edge disjoint paths between s and t in G , in $O(mn)$ time.*

As a consequence we get the following “cute” result.

Theorem 15.1.6 (Menger’s theorem). *In a directed graph G with nodes s and t the maximum number of edge disjoint $s - t$ paths is equal to the minimum number of edges whose removal separates s from t .*

Proof: Let U be a collection of edge-disjoint paths from s to t in G . If we remove a set F of edges from G and separate s from t , then it must be that every path in U uses at least one edge of F . Thus, the number of edge-disjoint paths is bounded by the number of edges needed to be removed to separate s and t . Namely, $|U| \leq |F|$.

As for the other direction, let F be a set of edges that its removal separates s and t . We claim that the set F form a cut in G between s and t . Indeed, let S be the set of all vertices in G that are reachable from s without using an edge of F . Clearly, if F is minimal then it must be all the edges of the cut (S, T) (in particular, if F contains some edge which is not in (S, T) we can remove it and get a smaller separating set of edges). In particular, the smallest set F with this separating property has the same size as the minimum cut between s and t in G , which is by the max-flow mincut theorem, also the maximum flow in the graph G (where every edge has capacity 1).

But then, by [Theorem 15.1.5](#), there are $|F|$ edge disjoint paths in G (since $|F|$ is the amount of the maximum flow). ■

15.1.2. Edge-disjoint paths in undirected graphs

We would like to solve the $s-t$ disjoint path problem for an undirected graph.

Problem 15.1.7. Given undirected graph G , s and t , find the maximum number of edge-disjoint paths in G between s and t .

The natural approach is to duplicate every edge in the undirected graph G , and get a (new) directed graph J . Next, apply the algorithm of [Section 15.1.1](#) to J .

So compute for J the maximum flow f (where every edge has capacity 1). The problem is the flow f might use simultaneously the two edges (u, v) and (v, u) . Observe, however, that in such case we can remove both edges from the flow f . In the resulting flow is legal and has the same value. As such, if we repeatedly remove those “double edges” from the flow f , the resulting flow f' has the same value. Next, we extract the edge disjoint paths from the graph, and the resulting paths are now edge disjoint in the original graph.

Lemma 15.1.8. *There are k edge-disjoint paths in an undirected graph G from s to t if and only if the maximum value of an $s - t$ flow in the directed version J of G is at least k . Furthermore, the Ford-Fulkerson algorithm can be used to find the maximum set of disjoint $s-t$ paths in G in $O(mn)$ time.*

15.2. Circulations with demands

15.2.1. Circulations with demands

We next modify and extend the network flow problem. Let $G = (V, E)$ be a directed graph with capacities on the edges. Each vertex v has a demand d_v :

- $d_v > 0$: sink requiring d_v flow into this node.
- $d_v < 0$: source with $-d_v$ units of flow leaving it.
- $d_v = 0$: regular node.

Let S denote all the source vertices and T denote all the sink/target vertices.

For a concrete example of an instance of circulation with demands, see figure on the right.

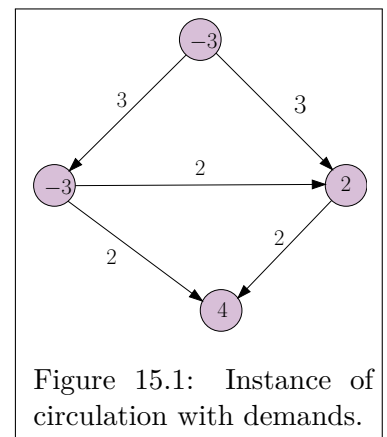


Figure 15.1: Instance of circulation with demands.

Definition 15.2.1. A **circulation** with demands $\{d_v\}$ is a function f that assigns nonnegative real values to the edges of G , such that:

- Capacity condition: $\forall e \in E$ we have $f(e) \leq c(e)$.
- Conservation condition: $\forall v \in V$ we have $f^{in}(v) - f^{out}(v) = d_v$.

Here, for a vertex v , let $f^{in}(v)$ denotes the flow into v and $f^{out}(v)$ denotes the flow out of v .

Problem 15.2.2. Is there a circulation that comply with the demand requirements?

See Figure 15.1 and Figure 15.2 for an example.

Lemma 15.2.3. *If there is a feasible circulation with demands $\{d_v\}$, then $\sum_v d_v = 0$.*

Proof: Since it is a circulation, we have that $d_v = f^{in}(v) - f^{out}(v)$. Summing over all vertices: $\sum_v d_v = \sum_v f^{in}(v) - \sum_v f^{out}(v)$. The flow on every edge is summed twice, one with positive sign, one with negative sign. As such,

$$\sum_v d_v = \sum_v f^{in}(v) - \sum_v f^{out}(v) = 0,$$

which implies the claim. ■

In particular, this implies that there is a feasible solution only if

$$D = \sum_{v, d_v > 0} d_v = \sum_{v, d_v < 0} -d_v.$$

15.2.1.1. The algorithm for computing a circulation

The algorithm performs the following steps:

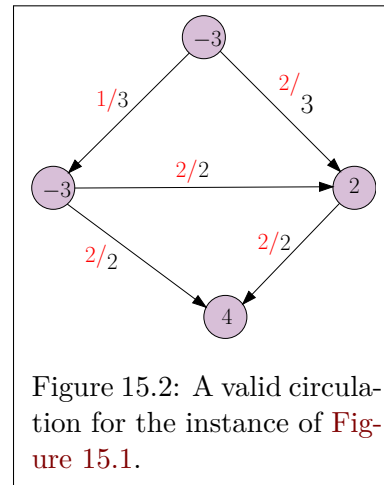
- $G = (V, E)$ - input flow network with demands on vertices.
- Check that $D = \sum_{v, d_v > 0} d_v = \sum_{v, d_v < 0} -d_v$.
- Create a new super source s , and connect it to all the vertices v with $d_v < 0$. Set the capacity of the edge (s, v) to be $-d_v$.
- Create a new super target t . Connect to it all the vertices u with $d_u > 0$. Set capacity on the new edge (u, t) to be d_u .
- On the resulting network flow network J (which is a standard instance of network flow). Compute maximum flow on J from s to t . If it is equal to D , then there is a valid circulation, and it is the flow restricted to the original graph. Otherwise, there is no valid circulation.

Theorem 15.2.4. *There is a feasible circulation with demands $\{d_v\}$ in G if and only if the maximum s - t flow in J has value D . If all capacities and demands in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer valued.*

15.3. Circulations with demands and lower bounds

Assume that in addition to specifying a circulation and demands on a network G , we also specify for each edge a lower bound on how much flow should be on each edge. Namely, for every edge $e \in E(G)$, we specify $\ell(e) \leq c(e)$, which is a lower bound to how much flow must be on this edge. As before we assume all numbers are integers.

We need now to compute a flow f that fill all the demands on the vertices, and that for any edge e , we have $\ell(e) \leq f(e) \leq c(e)$. The question is how to compute such a flow?



Let us start from the most naive flow, which transfer on every edge, exactly its lower bound. This is a valid flow as far as capacities and lower bounds, but of course, it might violate the demands. Formally, let $f_0(e) = \ell(e)$, for all $e \in E(G)$. Note that f_0 does not even satisfy the conservation rule:

$$L_v = f_0^{in}(v) - f_0^{out}(v) = \sum_{e \text{ into } v} \ell(e) - \sum_{e \text{ out of } v} \ell(e).$$

If $L_v = d_v$, then we are happy, since this flow satisfies the required demand. Otherwise, there is imbalance at v , and we need to fix it.

Formally, we set a new demand $d'_v = d_v - L_v$ for every node v , and the capacity of every edge e to be $c'(e) = c(e) - \ell(e)$. Let G' denote the new network with those capacities and demands (note, that the lower bounds had “disappeared”). If we can find a circulation f' on G' that satisfies the new demands, then clearly, the flow $f = f_0 + f'$, is a legal circulation, it satisfies the demands and the lower bounds.

But finding such a circulation, is something we already know how to do, using the algorithm of [Theorem 15.2.4](#). Thus, it follows that we can compute a circulation with lower bounds.

Lemma 15.3.1. *There is a feasible circulation in G if and only if there is a feasible circulation in G' .*

If all demands, capacities, and lower bounds in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer valued.

Proof: Let f' be a circulation in G' . Let $f(e) = f_0(e) + f'(e)$. Clearly, f satisfies the capacity condition in G , and the lower bounds. Furthermore,

$$f^{in}(v) - f^{out}(v) = \sum_{e \text{ into } v} (\ell(e) + f'(e)) - \sum_{e \text{ out of } v} (\ell(e) + f'(e)) = L_v + (d_v - L_v) = d_v.$$

As such f satisfies the demand conditions on G .

Similarly, let f be a valid circulation in G . Then it is easy to check that $f'(e) = f(e) - \ell(e)$ is a valid circulation for G' . ■

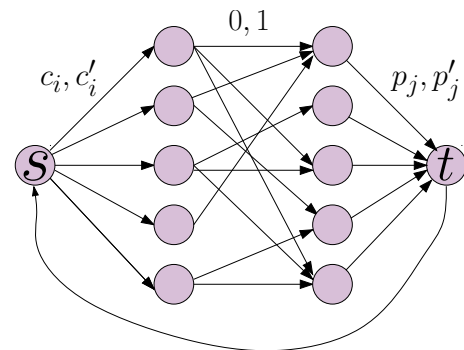
15.4. Applications

15.4.1. Survey design

We would like to design a survey of products used by consumers (i.e., “Consumer i : what did you think of product j ?”). The i th consumer agreed in advance to answer a certain number of questions in the range $[c_i, c'_i]$. Similarly, for each product j we would like to have at least p_j opinions about it, but not more than p'_j . Each consumer can be asked about a subset of the products which they consumed. In particular, we assume that we know in advance all the products each consumer used, and the above constraints. The question is how to assign questions to consumers, so that we get all the information we want to get, and every consumer is being asked a valid number of questions.

The idea of our solution is to reduce the design of the survey to the problem of computing a circulation in graph. First, we build a bipartite graph having consumers on one side, and products on the other side. Next, we insert the edge between consumer i and product j if the product was used by this consumer. The capacity of this edge is going to be 1. Intuitively, we are going to compute a flow in this network which is going to be an integer number. As such, every edge would be assigned either 0 or 1, where 1 is interpreted as asking the consumer about this product.

The next step, is to connect a source to all the consumers, where the edge (s, i) has lower bound c_i and upper bound c'_i . Similarly, we connect all the products to the destination t , where (j, t) has lower bound p_j and upper bound p'_j . We would like to compute a flow from s to t in this network that comply with the constraints. However, we only know how to compute a circulation on such a network. To overcome this, we create an edge with infinite capacity between t and s . Now, we are only looking for a valid circulation in the resulting graph G which complies with the aforementioned constraints. See figure on the right for an example of G .



Given a circulation f in G it is straightforward to interpret it as a survey design (i.e., all middle edges with flow 1 are questions to be asked in the survey). Similarly, one can verify that given a valid survey, it can be interpreted as a valid circulation in G . Thus, computing circulation in G indeed solves our problem.

We summarize:

Lemma 15.4.1. *Given n consumers and u products with their constraints $c_1, c'_1, c_2, c'_2, \dots, c_n, c'_n, p_1, p'_1, \dots, p_u, p'_u$ and a list of length m of which products were used by which consumers. An algorithm can compute a valid survey under these constraints, if such a survey exists, in time $O((n + u)m^2)$.*

Chapter 16

Network Flow IV - Applications II

16.1. Airline Scheduling

Problem 16.1.1. Given information about flights that an airline needs to provide, generate a profitable schedule.

The input is a detailed information about “legs” of flight that the airline need to serve. We denote this set of flights by \mathcal{F} . We would like to find the minimum number of airplanes needed to carry out this schedule. For an example of possible input, see [Figure 16.1](#) (i).

We can use the same airplane for two segments i and j if the destination of i is the origin of the segment j and there is enough time in between the two flights for required maintenance. Alternatively, the airplane can fly from $\text{dest}(i)$ to $\text{origin}(j)$ (assuming that the time constraints are satisfied).

Example 16.1.2. As a concrete example, consider the flights:

- (A) Boston (depart 6 A.M.) - Washington D.C. (arrive 7 A.M.),
- (B) Washington (depart 8 A.M.) - Los Angeles (arrive 11 A.M.)
- (C) Las Vegas (depart 5 P.M.) - Seattle (arrive 6 P.M.)

This schedule can be served by a single airplane by adding the leg “Los Angeles (depart 12 noon)- Las Vegas (1 P.M.)” to this schedule.

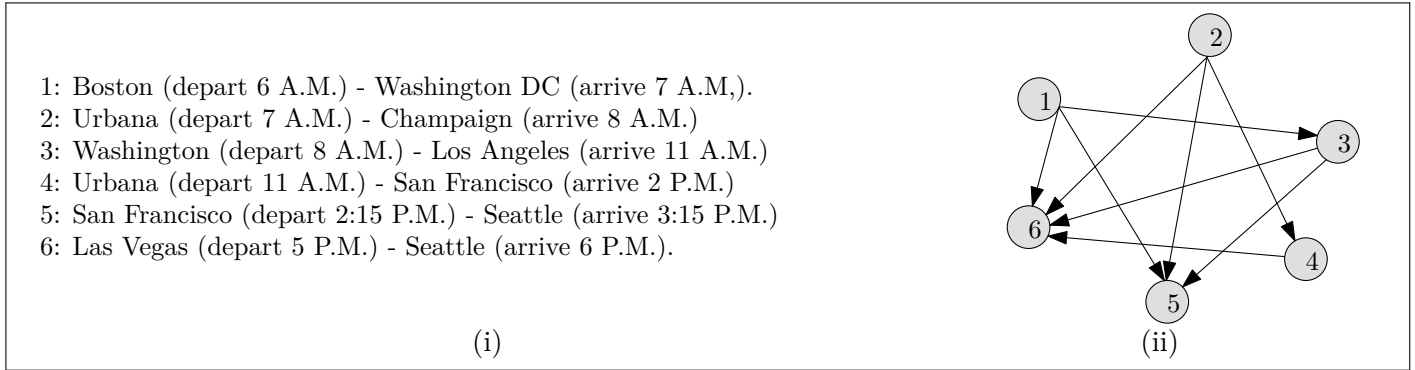


Figure 16.1: (i) a set \mathcal{F} of flights that have to be served, and (ii) the corresponding graph G representing these flights.

16.1.1. Modeling the problem

The idea is to model the feasibility constraints by a graph. Specifically, G is going to be a directed graph over the flight legs. For i and j , two given flight legs, the edge (i, j) will be present in the graph G if the same airplane can serve both i and j ; namely, the same airplane can perform leg i and afterwards serves the leg j .

Thus, the graph G is acyclic. Indeed, since we can have an edge (i, j) only if the flight j comes after the flight i (in time), it follows that we can not have cycles.

We need to decide if all the required legs can be served using only k airplanes?

16.1.2. Solution

The idea is to perform a reduction of this problem to the computation of circulation. Specifically, we construct a graph J , as follows:

- For every leg i , we introduce two vertices $u_i, v_i \in \mathcal{V}J$. We also add a source vertex s and a sink vertex t to J . We set the demand at t to be k , and the demand at s to be $-k$ (i.e., k units of flow are leaving s and need to arrive to t).
- Each flight on the list must be served. This is forced by introducing an edge $e_i = (u_i, v_i)$, for each leg i . We also set the lower bound on e_i to be 1, and the capacity on e_i to be 1 (i.e., $\ell(e_i) = 1$ and $c(e_i) = 1$).
- If the same plane can perform flight i and j (i.e., $(i, j) \in E(G)$) then add an edge (v_i, u_j) with capacity 1 to J (with no lower bound constraint).
- Since any airplane can start the day with flight i , we add an edge (s, u_i) with capacity 1 to J , for all flights i .
- Similarly, any airplane can end the day by serving the flight j . Thus, we add edge (v_j, t) with capacity 1 to G , for all flights j .
- If we have extra planes, we do not have to use them. As such, we introduce a “overflow” edge (s, t) with capacity k , that can carry over all the unneeded airplanes from s directly to t .

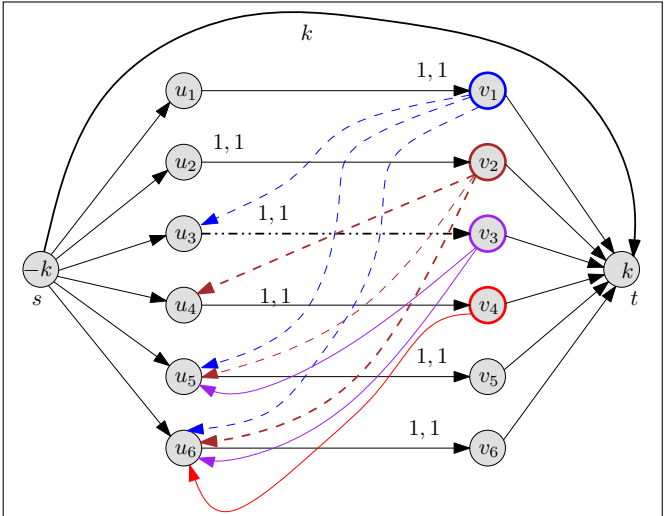


Figure 16.2: The resulting graph J for the instance of airline scheduling from Figure 16.1.

Let J denote the resulting graph. See Figure 16.2 for an example.

Lemma 16.1.3. *There is a way to perform all flights of \mathcal{F} using at most k planes if and only if there is a feasible circulation in the network J .*



(i)



(ii)

Figure 16.3: The (i) input image, and (ii) a possible segmentation of the image.

Proof: Assume there is a way to perform the flights using $k' \leq k$ flights. Consider such a feasible schedule. The schedule of an airplane in this schedule defines a path π in the network J that starts at s and ends at t , and we send one unit of flow on each such path. We also send $k - k'$ units of flow on the edge (s, t) . Note, that since the schedule is feasible, all legs are being served by some airplane. As such, all the “middle” edges with lower-bound 1 are being satisfied. Thus, this results in a valid circulation in J that satisfies all the given constraints.

As for the other direction, consider a feasible circulation in J . This is an integer valued circulation by the Integrality theorem. Suppose that k' units of flow are sent between s and t (ignoring the flow on the edge (s, t)). All the edges of J (except (s, t)) have capacity 1, and as such the circulation on all other edges is either zero or one (by the Integrality theorem). We convert this into k' paths by repeatedly traversing from the vertex s to the destination t , removing the edges we are using in each such path after extracting it (as we did for the k disjoint paths problem). Since we never use an edge twice, and J is acyclic, it follows that we would extract k' paths. Each of those paths correspond to one airplane, and the overall schedule for the airplanes is valid, since all required legs are being served (by the lower-bound constraint). ■

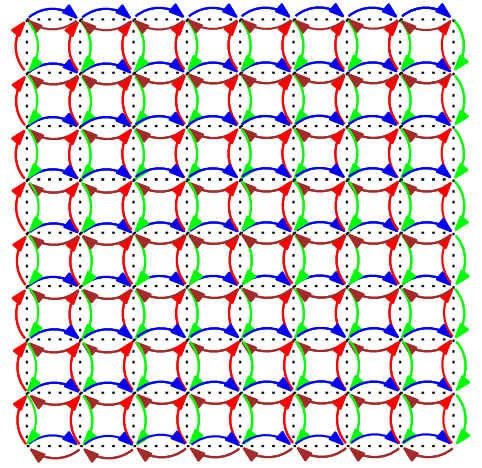
Extensions and limitations. There are a lot of other considerations that we ignored in the above problem: (i) airplanes have to undergo long term maintenance treatments every once in awhile, (ii) one needs to allocate crew to these flights, (iii) schedule differ between days, and (iv) ultimately we interested in maximizing revenue (a much more fluffy concept and much harder to explicitly describe).

In particular, while network flow is used in practice, real world problems are complicated, and network flow can capture only a few aspects. More than undermining the usefulness of network flow, this emphasize the complexity of real-world problems.

16.2. Image Segmentation

In the *image segmentation problem*, the input is an image, and we would like to partition it into background and foreground. For an example, see [Figure 16.3](#).

The input is a bitmap on a grid where every grid node represents a pixel. We convert this grid into a directed graph G , by interpreting every edge of the grid as two directed edges. See the figure on the right to see how the resulting graph looks like.



Specifically, the input for our problem is as follows: • A bitmap of size $N \times N$, with an associated directed graph $G = (V, E)$. • For every pixel i , we have a value $f_i \geq 0$, which is an estimate of the likelihood of this pixel to be in foreground (i.e., the larger f_i is the more probable that it is in the foreground) • For every pixel i , we have (similarly) an estimate b_i of the likelihood of pixel i to be in background.

• For every two adjacent pixels i and j we have a separation penalty p_{ij} , which is the “price” of separating i from j . This quantity is defined only for adjacent pixels in the bitmap. (For the sake of simplicity of exposition we assume that $p_{ij} = p_{ji}$. Note, however, that this assumption is not necessary for our discussion.)

Problem 16.2.1. Given input as above, partition V (the set of pixels) into two disjoint subsets F and B , such that

$$q(F, B) = \sum_{i \in F} f_i + \sum_{i \in B} b_i - \sum_{(i,j) \in E, |F \cap \{i,j\}|=1} p_{ij}.$$

is maximized.

We can rewrite $q(F, B)$ as:

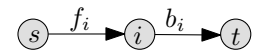
$$\begin{aligned} q(F, B) &= \sum_{i \in F} f_i + \sum_{j \in B} b_j - \sum_{(i,j) \in E, |F \cap \{i,j\}|=1} p_{ij} \\ &= \sum_{i \in V} (f_i + b_i) - \left(\sum_{i \in B} f_i + \sum_{j \in F} b_j + \sum_{(i,j) \in E, |F \cap \{i,j\}|=1} p_{ij} \right). \end{aligned}$$

Since the term $\sum_{i \in V} (f_i + b_i)$ is a constant, maximizing $q(F, B)$ is equivalent to minimizing $u(F, B)$, where

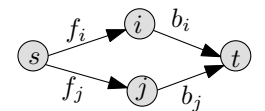
$$u(F, B) = \sum_{i \in B} f_i + \sum_{j \in F} b_j + \sum_{(i,j) \in E, |F \cap \{i,j\}|=1} p_{ij}. \quad (16.1)$$

How do we compute this partition. Well, the basic idea is to compute a minimum cut in a graph such that its price would correspond to $u(F, B)$. Before dwelling into the exact details, it is useful to play around with some toy examples to get some intuition. Note, that we are using the max-flow algorithm as an algorithm for computing minimum directed cut.

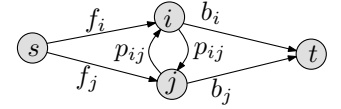
To begin with, consider a graph having a source s , a vertex i , and a sink t . We set the price of (s, i) to be f_i and the price of the edge (i, t) to be b_i . Clearly, there are two possible cuts in the graph, either $(\{s, i\}, \{t\})$ (with a price b_i) or $(\{s\}, \{i, t\})$ (with a price f_i). In particular, every path of length 2 in the graph between s and t forces the algorithm computing the minimum-cut (via network flow) to choose one of the edges, to the cut, where the algorithm “prefers” the edge with lower price.



Next, consider a bitmap with two vertices i and j that are adjacent. Clearly, minimizing the first two terms in Eq. (16.1) is easy, by generating length two parallel paths between s and t through i and j . See figure on the right. Clearly, the price of a cut in this graph is exactly the price of the partition of $\{i, j\}$ into background and foreground sets. However, this ignores the separation penalty p_{ij} .



To this end, we introduce two new edges (i, j) and (j, i) into the graph and set their price to be p_{ij} . Clearly, a price of a cut in the graph can be interpreted as the value of $u(F, B)$ of the corresponding sets F and B , since all the edges in the segmentation from nodes of F to nodes of B are contributing their separation price to the cut price. Thus, if we extend this idea to the directed graph G , the minimum-cut in the resulting graph would correspond to the required segmentation.



Let us recap: Given the directed grid graph $G = (V, E)$ we add two special source and sink vertices, denoted by s and t respectively. Next, for all the pixels $i \in V$, we add an edge $e_i = (s, i)$ to the graph, setting its capacity to be $c(e_i) = f_i$. Similarly, we add the edge $e'_i = (j, t)$ with capacity $c(e'_i) = b_i$. Similarly, for every pair of vertices i, j in that grid that are adjacent, we assign the cost p_{ij} to the edges (i, j) and (j, i) . Let J denote the resulting graph.

The following lemma, follows by the above discussion.

Lemma 16.2.2. *A minimum cut (F, B) in J minimizes $u(F, B)$.*

Using the minimum-cut max-flow theorem, we have:

Theorem 16.2.3. *One can solve the segmentation problem, in polynomial time, by computing the max flow in the graph J .*

16.3. Project Selection

You have a small company which can carry out some projects out of a set of projects P . Associated with each project $i \in P$ is a revenue p_i , where $p_i > 0$ is a profitable project and $p_i < 0$ is a losing project. To make things interesting, there is dependency between projects. Namely, one has to complete some “infrastructure” projects before one is able to do other projects. Namely, you are provided with a graph $G = (P, E)$ such that $(i, j) \in E$ if and only if j is a prerequisite for i .

Definition 16.3.1. A set $X \subset P$ is **feasible** if for all $i \in X$, all the prerequisites of i are also in X . Formally, for all $i \in X$, with an edge $(i, j) \in E$, we have $j \in X$.

The **profit** associated with a set of projects $X \subseteq P$ is $\text{profit}(X) = \sum_{i \in X} p_i$.

Problem 16.3.2 (Project Selection Problem). Select a feasible set of projects maximizing the overall profit.

The idea of the solution is to reduce the problem to a minimum-cut in a graph, in a similar fashion to what we did in the image segmentation problem.

16.3.1. The reduction

The reduction works by adding two vertices s and t to the graph G , we also perform the following modifications:

- For all projects $i \in P$ with positive revenue (i.e., $p_i > 0$) add the $e_i = (s, i)$ to G and set the capacity of the edge to be $c(e_i) = p_i$, where s is the added source vertex.
- Similarly, for all projects $j \in P$, with negative revenue (i.e., $p_j < 0$) add the edge $e'_j = (j, t)$ to G and set the edge capacity to $c(e'_j) = -p_j$, where t is the added sink vertex.
- Compute a bound on the max flow (and thus also profit) in this network: $C = \sum_{i \in P, p_i > 0} p_i$.
- Set capacity of all other edges in G to $4C$ (these are the dependency edges in the project, and intuitively they are too expensive to be “broken” by a cut).

Let J denote the resulting network.

Let $X \subseteq P$ be a set of feasible projects, and let $X' = X \cup \{s\}$ and $Y' = (P \setminus X) \cup \{t\}$. Consider the s - t cut (X', Y') in J . Note, that no edge of $E(G)$ is in (X', Y') since X is a feasible set (i.e., there is no $u \in X'$ and $v \in Y'$ such that $(u, v) \in E(G)$).

Lemma 16.3.3. *The capacity of the cut (X', Y') , as defined by a feasible project set X , is $c(X', Y') = C - \sum_{i \in X} p_i = C - \text{profit}(X)$.*

Proof: The edges of J are either:

- (i) original edges of G (conceptually, they have price $+\infty$),
- (ii) edges emanating from s , and
- (iii) edges entering t .

Since X is feasible, it follows that no edges of type (i) contribute to the cut. The edges entering t contribute to the cut the value

$$\beta = \sum_{i \in X \text{ and } p_i < 0} -p_i.$$

The edges leaving the source s contribute

$$\gamma = \sum_{i \notin X \text{ and } p_i > 0} p_i = \sum_{i \in P, p_i > 0} p_i - \sum_{i \in X \text{ and } p_i > 0} p_i = C - \sum_{i \in X \text{ and } p_i > 0} p_i,$$

by the definition of C .

The capacity of the cut (X', Y') is

$$\beta + \gamma = \sum_{i \in X \text{ and } p_i < 0} (-p_i) + \left(C - \sum_{i \in X \text{ and } p_i > 0} p_i \right) = C - \sum_{i \in X} p_i = C - \text{profit}(X),$$

as claimed. ■

Lemma 16.3.4. *If (X', Y') is a cut with capacity at most C in G , then the set $X = X' \setminus \{s\}$ is a feasible set of projects.*

Namely, cuts (X', Y') of capacity $\leq C$ in J corresponds one-to-one to feasible sets which are profitable.

Proof: Since $c(X', Y') \leq C$ it must not cut any of the edges of G , since the price of such an edge is $4C$. As such, X must be a feasible set. ■

Putting everything together, we are looking for a feasible set X that maximizes $\text{profit}(X) = \sum_{i \in X} p_i$. This corresponds to a set $X' = X \cup \{s\}$ of vertices in J that minimizes $C - \sum_{i \in X} p_i$, which is also the cut capacity (X', Y') . Thus, computing a minimum-cut in J corresponds to computing the most profitable feasible set of projects.

Theorem 16.3.5. *If (X', Y') is a minimum cut in J then $X = X' \setminus \{s\}$ is an optimum solution to the project selection problem. In particular, using network flow the optimal solution can be computed in polynomial time.*

Proof: Indeed, we use network flow to compute the minimum cut in the resulting graph J . Note, that it is quite possible that the most profitable project is still a net loss. ■

16.4. Baseball elimination

There is a baseball league taking place and it is nearing the end of the season. One would like to know which teams are still candidates to winning the season.

Example 16.4.1. There 4 teams that have the following number of wins:

NEW YORK: 92, BALTIMORE: 91, TORONTO: 91, BOSTON: 90,

and there are 5 games remaining (all pairs except New York and Boston).

We would like to decide if Boston can still win the season? Namely, can Boston finish the season with as many point as anybody else? (We are assuming here that at every game the winning team gets one point and the losing team gets nada.^①)

First analysis. Observe, that Boston can get at most 92 wins. In particular, if New York wins any game then it is over since New-York would have 93 points.

Thus, to Boston to have any hope it must be that both Baltimore wins against New York and Toronto wins against New York. At this point in time, both teams have 92 points. But now, they play against each other, and one of them would get 93 wins. So Boston is eliminated!

Second analysis. As before, Boston can get at most 92 wins. All three other teams gets $X = 92 + 91 + 91 + (5 - 2)$ points together by the end of the league. As such, one of these three teams will get $\geq \lceil X/3 \rceil = 93$ points, and as such Boston is eliminated.

While the analysis of the above example is very cute, it is too tedious to be done each time we want to solve this problem. Not to mention that it is unclear how to extend these analyses to other cases.

16.4.1. Problem definition

Problem 16.4.2. The input is a set S of teams, where for every team $x \in S$, the team has w_x points accumulated so far. For every pair of teams $x, y \in S$ we know that there are g_{xy} games remaining between x and y . Given a specific team z , we would like to decide if z is eliminated?

Alternatively, is there away such that z would get as many wins as anybody else by the end of the season?

16.4.2. Solution

First, we can assume that z wins all its remaining games, and let m be the number of points z has in this case. Our purpose now is to build a network flow so we can verify that no other team *must* get more than m points.

To this end, let s be the source (i.e., the source of wins). For every remaining game, a flow of one unit would go from s to one of the teams playing it. Every team can have at most $m - w_x$ flow from it to the target. If the max flow in this network has value

$$\alpha = \sum_{x,y \neq z, x < y} g_{xy}$$

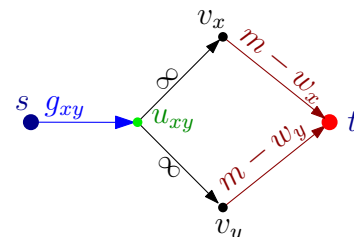
(which is the maximum flow possible) then there is a scenario such that all other teams gets at most m points and z can win the season. Negating this statement, we have that if the maximum flow is smaller than α then z is eliminated, since there must be a team that gets more than m points.

Construction. Let $S' = S \setminus \{z\}$ be the set of teams, and let

$$\alpha = \sum_{\{x,y\} \subseteq S'} g_{xy}. \tag{16.2}$$

We create a network flow G . For every team $x \in S'$ we add a vertex v_x to the network G . We also add the source and sink vertices, s and t , respectively, to G .

For every pair of teams $x, y \in S'$, such that $g_{xy} > 0$ we create a node u_{xy} , and add an edge (s, u_{xy}) with capacity g_{xy} to G . We also add the edge (u_{xy}, v_x) and (u_{xy}, v_y) with infinite capacity to G . Finally, for each team x we add the edge (v_x, t) with capacity $m - w_x$ to G . How the relevant edges look like for a pair of teams x and y is depicted on the right.



Analysis. If there is a flow of value α in G then there is a way that all teams get at most m wins. Similarly, if there exists a scenario such that z ties or gets first place then we can translate this into a flow in G of value α . This implies the following result.

Theorem 16.4.3. *Team z has been eliminated if and only if the maximum flow in G has value strictly smaller than α . Thus, we can test in polynomial time if z has been eliminated.*

16.4.3. A compact proof of a team being eliminated

Interestingly, once z is eliminated, we can generate a compact proof of this fact.

Theorem 16.4.4. *Suppose that team z has been eliminated. Then there exists a “proof” of this fact of the following form:*

(A) *The team z can finish with at most m wins.*

(B) *There is a set of teams $\widehat{S} \subset S$ so that $\sum_{s \in \widehat{S}} w_x + \sum_{\{x,y\} \subseteq \widehat{S}} g_{xy} > m|\widehat{S}|$.*

(And hence one of the teams in \widehat{S} must end with strictly more than m wins.)

Proof: If z is eliminated then the max flow in G has value γ , which is smaller than α , see Eq. (16.2). By the minimum-cut max-flow theorem, there exists a minimum cut (S, T) of capacity γ in G , and let $\widehat{S} = \{x \mid v_x \in S\}$

Claim 16.4.5. *For any two teams x and y for which the vertex u_{xy} exists, we have that $u_{xy} \in S$ if and only if both x and y are in \widehat{S} .*

Proof: $(x \notin \widehat{S} \text{ or } y \notin \widehat{S}) \implies u_{xy} \notin S$: If x is not in \widehat{S} then v_x is in T . But then, if u_{xy} is in S the edge (u_{xy}, v_x) is in the cut. However, this edge has infinite capacity, which implies this cut is not a minimum cut (in particular, (S, T) is a cut with capacity smaller than α). As such, in such a case u_{xy} must be in T . This implies that if either x or y are *not* in \widehat{S} then it must be that $u_{xy} \in T$. (And as such $u_{xy} \notin S$.)

$x \in \widehat{S}$ and $y \in \widehat{S} \implies u_{xy} \in S$: Assume that both x and y are in \widehat{S} , then v_x and v_y are in S . We need to prove that $u_{xy} \in S$. If $u_{xy} \in T$ then consider the new cut formed by moving u_{xy} to S . For the new cut (S', T') we have

$$c(S', T') = c(S, T) - c((s, u_{xy})).$$

Namely, the cut (S', T') has a lower capacity than the minimum cut (S, T) , which is a contradiction. See figure on the right for this *impossible* cut. We conclude that $u_{xy} \in S$. ■

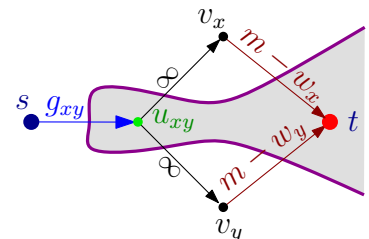
The above argumentation implies that edges of the type (u_{xy}, v_x) can not be in the cut (S, T) . As such, there are two type of edges in the cut (S, T) : (i) (v_x, t) , for $x \in \widehat{S}$, and (ii) (s, u_{xy}) where at least one of x or y is not in \widehat{S} . As such, the capacity of the cut (S, T) is

$$c(S, T) = \sum_{x \in \widehat{S}} (m - w_x) + \sum_{\{x,y\} \not\subseteq \widehat{S}} g_{xy} = m|\widehat{S}| - \sum_{x \in \widehat{S}} w_x + \left(\alpha - \sum_{\{x,y\} \subseteq \widehat{S}} g_{xy} \right).$$

However, $c(S, T) = \gamma < \alpha$, and it follows that

$$m|\widehat{S}| - \sum_{x \in \widehat{S}} w_x - \sum_{\{x,y\} \subseteq \widehat{S}} g_{xy} < \alpha - \alpha = 0.$$

Namely, $\sum_{x \in \widehat{S}} w_x + \sum_{\{x,y\} \subseteq \widehat{S}} g_{xy} > m|\widehat{S}|$, as claimed. ■



Chapter 17

Network Flow V - Min-cost flow

17.1. Minimum Average Cost Cycle

Let $G = (V, E)$ be a **digraph** (i.e., a directed graph) with n vertices and m edges, and $\omega : E \rightarrow \mathbb{R}$ be a weight function on the edges. A **directed cycle** is closed walk $C = (v_0, v_1, \dots, v_t)$, where $v_t = v_0$ and $(v_i, v_{i+1}) \in E$, for $i = 0, \dots, t-1$. The **average cost of a directed cycle** is $\text{AvgCost}(C) = \omega(C)/t = (\sum_{e \in C} \omega(e))/t$.

For each $k = 0, 1, \dots$, and $v \in V$, let $d_k(v)$ denote the minimum length of a walk with exactly k edges, ending at v (note, that the walk can start anywhere). So, for each v , we have

$$d_0(v) = 0 \quad \text{and} \quad d_{k+1}(v) = \min_{e=(u,v) \in E} (d_k(u) + \omega(e)).$$

Thus, we can compute $d_i(v)$, for $i = 0, \dots, n$ and $v \in V(G)$ in $O(nm)$ time using dynamic programming.

Let

$$\text{MinAvgCostCycle}(G) = \min_{C \text{ is a cycle in } G} \text{AvgCost}(C)$$

denote the average cost of the **minimum average cost cycle** in G .

The following theorem is somewhat surprising.

Theorem 17.1.1. *The minimum average cost of a directed cycle in G is equal to*

$$\alpha = \min_{v \in V} \max_{k=0}^{n-1} \frac{d_n(v) - d_k(v)}{n - k}.$$

Namely, $\alpha = \text{MinAvgCostCycle}(G)$.

Proof: Note, that adding a quantity r to the weight of every edge of G increases the average cost of a cycle $\text{AvgCost}(C)$ by r . Similarly, α would also increase by r . In particular, we can assume that the price of the minimum average cost cycle is zero. This implies that now all cycles have non-negative (average) cost.

Thus, from this point on we assume that $\text{MinAvgCostCycle}(G) = 0$, and we prove that $\alpha = 0$ in this case. This in turn would imply the theorem – indeed, given a graph where $\text{MinAvgCostCycle}(G) \neq 0$, then we will shift the costs the edges so that it is zero, use the proof below, and then shift it back.

$\text{MinAvgCostCycle}(G) = 0 \implies \alpha \geq 0$: We can rewrite α as $\alpha = \min_{u \in V} \beta(u)$, where

$$\beta(u) = \max_{k=0}^{n-1} \frac{d_n(u) - d_k(u)}{n - k}.$$

Assume, that α is realized by a vertex v ; that is $\alpha = \beta(v)$. Let P_n be a walk with n edges ending at v , of length $d_n(v)$. Since there are n vertices in G , it must be that P_n must contain a cycle. So, let us decompose P_n into a cycle π of length $n - k$ and a path σ of length k (k depends on the length of the cycle in P_n). We have that

$$d_n(v) = \omega(P_n) = \omega(\pi) + \omega(\sigma) \geq \omega(\sigma) \geq d_k(v),$$

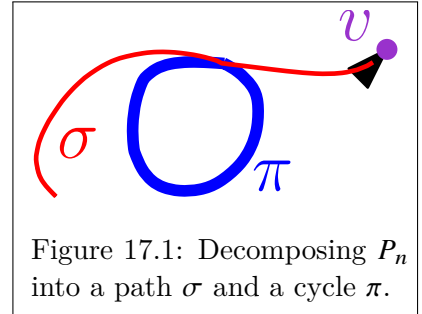


Figure 17.1: Decomposing P_n into a path σ and a cycle π .

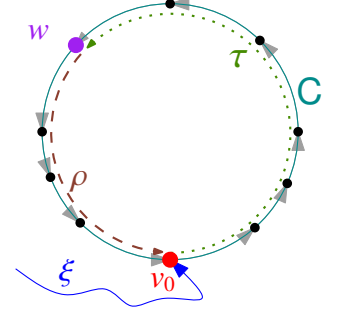
since $\omega(\pi) \geq 0$ as π is a cycle (and we assumed that all cycles have zero or positive cost). As such, we have $d_n(v) - d_k(v) \geq 0$. As such, $\frac{d_n(v) - d_k(v)}{n-k} \geq 0$. Let

$$\beta(v) = \max_{j=0}^{n-1} \frac{d_n(v) - d_j(v)}{n-j} \geq \frac{d_n(v) - d_k(v)}{n-k} \geq 0.$$

Now, $\alpha = \beta(v) \geq 0$, by the choice of v .

MinAvgCostCycle(G) = 0 $\implies \alpha \leq 0$: Let $C = (v_0, v_1, \dots, v_t)$ be the directed cycle of weight 0 in the graph. Observe, that $\min_{j=0}^{\infty} d_j(v_0)$ must be realized (for the first time) by an index $r < n$, since if it is longer, we can always shorten it by removing cycles and improve its price (since cycles have non-negative price). Let ξ denote this walk of length r ending at v_0 . Let w be a vertex on C reached by walking $n-r$ edges on C starting from v_0 , and let τ denote this walk (i.e., $|\tau| = n-r$). We have that

$$d_n(w) \leq \omega(\xi \parallel \tau) = d_r(v_0) + \omega(\tau), \quad (17.1)$$



where $\xi \parallel \tau$ denotes the path formed by concatenating the path τ to ξ .

Similarly, let ρ be the walk formed by walking on C from w all the way back to v_0 . Note that $\tau \parallel \rho$ goes around C several times, and as such, $\omega(\tau \parallel \rho) = 0$, as $\omega(C) = 0$. Next, for any k , since the shortest path with k edges arriving to w can be extended to a path that arrives to v_0 , by concatenating ρ to it, we have that

$$d_k(w) + \omega(\rho) \geq d_{k+|\rho|}(v_0) \geq d_r(v_0) \geq d_n(w) - \omega(\tau),$$

by Eq. (17.1). Rearranging, we have that $\omega(\rho) \geq d_n(w) - \omega(\tau) - d_k(w)$. Namely, we have

$$\begin{aligned} \forall k \quad 0 = \omega(\tau \parallel \rho) &= \omega(\rho) + \omega(\tau) \geq (d_n(w) - \omega(\tau) - d_k(w)) + \omega(\tau) = d_n(w) - d_k(w). \\ \implies \forall k \quad \frac{d_n(w) - d_k(w)}{n-k} &\leq 0 \\ \implies \beta(w) = \max_{k=0}^{n-1} \frac{d_n(w) - d_k(w)}{n-k} &\leq 0. \end{aligned}$$

As such, $\alpha = \min_{v \in V(G)} \beta(v) \leq \beta(w) \leq 0$, and we conclude that $\alpha = 0$. ■

Finding the minimum average cost cycle is now not too hard. We compute the vertex v that realizes α in **Theorem 17.1.1**. Next, we add $-\alpha$ to all the edges in the graph. We now know that we are looking for a cycle with price 0. We update the values $d_i(v)$ to agree with the new weights of the edges.

Now, v is the vertex realizing the quantity $0 = \alpha = \min_{u \in V} \max_{k=0}^{n-1} \frac{d_n(u) - d_k(u)}{n-k}$. Namely, we have that for the vertex v it holds

$$\begin{aligned} \max_{k=0}^{n-1} \frac{d_n(v) - d_k(v)}{n-k} = 0 &\implies \forall k \in \{0, \dots, n-1\} \quad \frac{d_n(v) - d_k(v)}{n-k} \leq 0 \\ \implies \forall k \in \{0, \dots, n-1\} \quad d_n(v) - d_k(v) &\leq 0. \end{aligned}$$

This implies that $d_n(v) \leq d_i(v)$, for all i . Now, we repeat the proof of **Theorem 17.1.1**. Let P_n be the path with n edges realizing $d_n(v)$. We decompose it into a path π of length k and a cycle τ . We know that $\omega(\tau) \geq 0$ (since all cycles have non-negative weights now). Now, $\omega(\pi) \geq d_k(v)$. As such, $\omega(\tau) = d_n(v) - \omega(\pi) \leq d_n(v) - d_k(v) \leq 0$, as π is a path of length k ending at v , and its cost is $\geq d_k(v)$. Namely, the cycle τ has $\omega(\tau) \leq 0$, and it the required cycle and computing it required $O(nm)$ time.

Note, that the above reweighting in fact was not necessary. All we have to do is to compute the node realizing α , extract P_n , and compute the cycle in P_n , and we are guaranteed by the above argumentation, that this is the cheapest average cycle.

Corollary 17.1.2. *Given a direct graph G with n vertices and m edges, and a weight function $\omega(\cdot)$ on the edges, one can compute the cycle with minimum average cost in $O(nm)$ time.*

17.2. Potentials

In general computing the shortest path in a graph that have negative weights is harder than just using the Dijkstra algorithm (that works only for graphs with non-negative weights on its edges). One can use Bellman-Ford algorithm^① in this case, but it considerably slower (i.e., it takes $O(mn)$ time). We next present a case where one can still use Dijkstra algorithm, with slight modifications.

The following is only required in the analysis of the minimum-cost flow algorithm we present later in this chapter. We describe it here in full detail since its simple and interesting.

For a directed graph $G = (V, E)$ with weight $w(\cdot)$ on the edges, let $\mathbf{d}_\omega(s, t)$ denote the length of the shortest path between s and t in G under the weight function w . Note, that w might assign negative weights to edges in G .

A **potential** $p(\cdot)$ is a function that assigns a real value to each vertex of G , such that if $e = (u, v) \in G$ then $w(e) \geq p(v) - p(u)$.

Lemma 17.2.1. (i) *There exists a potential $p(\cdot)$ for G if and only if G has no negative cycles (with respect to $w(\cdot)$).*

(ii) *Given a potential function $p(\cdot)$, for an edge $e = (u, v) \in E(G)$, let $\ell(e) = w(e) - p(v) + p(u)$. Then $\ell(\cdot)$ is non-negative for the edges in the graph and for any pair of vertices $s, t \in V(G)$, we have that the shortest path π realizing $\mathbf{d}_\ell(s, t)$ also realizes $\mathbf{d}_\omega(s, t)$.*

(iii) *Given G and a potential function $p(\cdot)$, one can compute the shortest path from s to all the vertices of G in $O(n \log n + m)$ time, where G has n vertices and m edges*

Proof: (i) Consider a cycle C , and assume there is a potential $p(\cdot)$ for G , and observe that

$$w(C) = \sum_{(u,v) \in E(C)} w(e) \geq \sum_{(u,v) \in E(C)} (p(v) - p(u)) = 0,$$

as required.

For a vertex $v \in V(G)$, let $p(v)$ denote the shortest walk that ends at v in G . We claim that $p(\cdot)$ is a potential. Since G does not have negative cycles, the quantity $p(v)$ is well defined. Observe that $p(v) \leq p(u) + w(u \rightarrow v)$ since we can always continue a walk to u into v by traversing (u, v) . Thus, $p(v) - p(u) \leq w(u \rightarrow v)$, as required.

(ii) Since $\ell(e) = w(e) - p(v) + p(u)$ we have that $w(e) \geq p(v) - p(u)$ since $p(\cdot)$ is a potential function. As such $w(e) - p(v) + p(u) \geq 0$, as required.

As for the other claim, observe that for any path π in G starting at s and ending at t we have that

$$\ell(\pi) = \sum_{e=(u,v) \in \pi} (w(e) - p(v) + p(u)) = w(\pi) + p(s) - p(t),$$

which implies that $\mathbf{d}_\ell(s, t) = \mathbf{d}_\omega(s, t) + p(s) - p(t)$. Implying the claim.

(iii) Just use the Dijkstra algorithm on the distances defined by $\ell(\cdot)$. The shortest paths are preserved under this distance by (ii), and this distance function is always positive. ■

17.3. Minimum cost flow

Given a network flow $G = (V, E)$ with source s and sink t , capacities $c(\cdot)$ on the edges, a real number ϕ , and a cost function $\kappa(\cdot)$ on the edges. The **cost** of a flow f is defined to be

$$\text{cost}(f) = \sum_{e \in E} \kappa(e) * f(e).$$

The **minimum-cost s - t flow problem** ask to find the flow f that minimizes the cost and has value ϕ .

^①http://en.wikipedia.org/wiki/Bellman-Ford_algorithm

It would be easier to look on the problem of *minimum-cost circulation problem*. Here, we are given instead of ϕ a lower-bound $\ell(\cdot)$ on the flow on every edge (and the regular upper bound $c(\cdot)$ on the capacities of the edges). All the flow coming into a node must leave this node. It is easy to verify that if we can solve the minimum-cost circulation problem, then we can solve the min-cost flow problem. Thus, we will concentrate on the min-cost circulation problem.

An important technicality is that all the circulations we discuss here have zero demands on the vertices. As such, a circulation can be conceptually considered to be a flow going around in cycles in the graph without ever stopping. In particular, for these circulations, the conservation of flow property should hold for all the vertices in the graph.

The *residual graph* of f is the graph $G_f = (V, E_f)$ where

$$E_f = \left\{ e = (u, v) \in V \times V \mid f(e) < c(e) \text{ or } f(e^{-1}) > \ell(e^{-1}) \right\}.$$

where $e^{-1} = (v, u)$ if $e = (u, v)$. Note, that the definition of the residual network takes into account the lower-bound on the capacity of the edges.

Assumption 17.3.1. *To simplify the exposition, we will assume that if $(u, v) \in E(G)$ then $(v, u) \notin E(G)$, for all $u, v \in V(G)$. This can be easily enforced by introducing a vertex in the middle of every edge of G . This is acceptable, since we are more concerned with solving the problem at hand in polynomial time, than the exact complexity. Note, that our discussion can be extended to handle the slightly more general case, with a bit of care.*

We extend the cost function to be anti-symmetric; namely,

$$\forall (u, v) \in E_f \quad \kappa((u, v)) = -\kappa((v, u)).$$

Consider a directed cycle C in G_f . For an edge $e = (u, v) \in E$, we define

$$\chi_C(e) = \begin{cases} 1 & e \in C \\ -1 & e^{-1} = (v, u) \in C \\ 0 & \text{otherwise;} \end{cases}$$

that is, we pay 1 if e is in C and -1 if we travel e in the “wrong” direction.

The *cost* of a directed cycle C in G_f is defined as

$$\kappa(C) = \sum_{e \in C} \kappa(e).$$

We will refer to a circulation that comply with the capacity and lower-bounds constraints as being *valid*. A function that just comply with the conservation property (i.e., all incoming flow into a vertex leaves it), is a *weak circulation*. In particular, a weak circulation might not comply with the capacity and lower bounds constraints of the given instance, and as such is not a valid circulation.

We need the following easy technical lemmas.

Lemma 17.3.2. *Let f and g be two valid circulations in $G = (V, E)$. Consider the function $h = g - f$. Then, h is a weak circulation, and if $h(u \rightarrow v) > 0$ then the edge $(u, v) \in G_f$.*

Proof: The fact that h is a circulation is trivial, as it is the difference between two circulations, and as such the same amount of flow that comes into a vertex leaves it, and thus it is a circulation. (Note, that h might not be a valid circulation, since it might not comply with the lower-bounds on the edges.)

Observe, that if $h(u \rightarrow v)$ is negative, then $h(v \rightarrow u) = -h(u \rightarrow v)$ by the anti-symmetry of f and g , which implies the same property holds for h .

Consider an arbitrary edge $e = (u, v)$ such that $h(u \rightarrow v) > 0$.

There are two possibilities. First, if $e = (u, v) \in E$, and $f(e) < c(e)$, then the claim trivially holds, since then $e \in G_f$. Thus, consider the case when $f(e) = c(e)$, but then $h(e) = g(e) - f(e) \leq 0$. Which contradicts our assumption that $h(u \rightarrow v) > 0$.

The second possibility, is that $e = (u, v) \notin E$. But then $e^{-1} = (v, u)$ must be in E , and it holds $0 > h(e^{-1}) = g(e^{-1}) - f(e^{-1})$. Implying that $f(e^{-1}) > g(e^{-1}) \geq \ell(e^{-1})$. Namely, there is a flow by f in G going in the direction of e^{-1} which larger than the lower bound. Since we can return this flow in the other direction, it must be that $e \in G_f$. ■

Lemma 17.3.3. *Let f be a circulation in a graph G . Then, f can be decomposed into at most m cycles, C_1, \dots, C_m , such that, for any $e \in E(G)$, we have*

$$f(e) = \sum_{i=1}^t \lambda_i \cdot \chi_{C_i}(e),$$

where $\lambda_1, \dots, \lambda_t > 0$ and $t \leq m$, where m is the number of edges in G .

Proof: Since f is a circulation, and the amount of flow into a node is equal to the amount of flow leaving the node, it follows that as long as f not zero, one can find a cycle in f . Indeed, start with a vertex which has non-zero amount of flow into it, and walk on an adjacent edge that has positive flow on it. Repeat this process, till you visit a vertex that was already visited. Now, extract the cycle contained in this walk.

Let C_1 be such a cycle, and observe that every edge of C_1 has positive flow on it, let λ_1 be the smallest amount of flow on any edge of C_1 , and let e_1 denote this edge. Consider the new flow $g = f - \lambda_1 \cdot \chi_{C_1}$. Clearly, g has zero flow on e_1 , and it is a circulation. Thus, we can remove e_1 from G , and let J denote the new graph. By induction, applied to g on J , the flow g can be decomposed into $m - 1$ cycles with positive coefficients. Putting these cycles together with λ_1 and C_1 implies the claim. ■

Theorem 17.3.4. *A flow f is a minimum cost feasible circulation if and only if each directed cycle of G_f has nonnegative cost.*

Proof: Let C be a negative cost cycle in G_f . Then, we can circulate more flow on C and get a flow with smaller price. In particular, let $\varepsilon > 0$ be a sufficiently small constant, such that $g = f + \varepsilon \cdot \chi_C$ is still a feasible circulation (observe, that since the edges of C are G_f , all of them have residual capacity that can be used to this end). Now, we have that

$$\text{cost}(g) = \text{cost}(f) + \sum_{e \in C} \kappa(e) * \varepsilon = \text{cost}(f) + \varepsilon * \sum_{e \in C} \kappa(e) = \text{cost}(f) + \varepsilon * \kappa(C) < \text{cost}(f),$$

since $\kappa(C) < 0$, which is a contradiction to the minimality of f .

As for the other direction, assume that all the cycles in G_f have non-negative cost. Then, let g be any feasible circulation. Consider the circulation $h = g - f$. By [Lemma 17.3.2](#), all the edges used by h are in G_f , and by [Lemma 17.3.3](#) we can find $t \leq |E(G_f)|$ cycles C_1, \dots, C_t in G_f , and coefficients $\lambda_1, \dots, \lambda_t$, such that

$$h(e) = \sum_{i=1}^t \lambda_i \chi_{C_i}(e).$$

We have that

$$\text{cost}(g) - \text{cost}(f) = \text{cost}(h) = \text{cost}\left(\sum_{i=1}^t \lambda_i \chi_{C_i}\right) = \sum_{i=1}^t \lambda_i \text{cost}(\chi_{C_i}) = \sum_{i=1}^t \lambda_i \kappa(C_i) \geq 0,$$

as $\kappa(C_i) \geq 0$, since there are no negative cycles in G_f . This implies that $\text{cost}(g) \geq \text{cost}(f)$. Namely, f is a minimum-cost circulation. ■

17.4. A Strongly Polynomial Time Algorithm for Min-Cost Flow

The algorithm would start from a feasible circulation f . We know how to compute such a flow f using the standard max-flow algorithm. At each iteration, it would find the cycle C of minimum average cost cycle in G_f (using the algorithm of Section 17.1). If the cost of C is non-negative, we are done since we had arrived to the minimum cost circulation, by Theorem 17.3.4.

Otherwise, we circulate as much flow as possible along C (without violating the lower-bound constraints and capacity constraints), and reduce the price of the flow f . By Corollary 17.1.2, we can compute such a cycle in $O(mn)$ time. Since the cost of the flow is monotonically decreasing the algorithm would terminate if all the number involved are integers. But we will show that this algorithm performs a polynomial number of iterations in n and m .

It is striking how simple is this algorithm, and the fact that it works in polynomial time. The analysis is somewhat more painful.

17.5. Analysis of the Algorithm

To analyze the above algorithm, let f_i be the flow in the beginning of the i th iteration. Let C_i be the cycle used in the i th iteration. For a flow f , let C_f the minimum average-length cycle of G_f , and let $\mu(f) = \kappa(C_f)/|C_f|$ denote the average “cost” per edge of C_f .

The following lemma, states that we are making “progress” in each iteration of the algorithm.

f, g, h, i	Flows or circulations
G_f	The residual graph for f
$c(e)$	The capacity of the flow on e
$\ell(e)$	The lower-bound (i.e., demand) on the flow on e
$\text{cost}(f)$	The overall cost of the flow f
$\kappa(e)$	The cost of sending one unit of flow on e
$\psi(e)$	The reduced cost of e

Figure 17.2: Notation used.

Lemma 17.5.1. *Let f be a flow, and let g the flow resulting from applying the cycle $C = C_f$ to it. Then, $\mu(g) \geq \mu(f)$.*

Proof: Assume for the sake of contradiction, that $\mu(g) < \mu(f)$. Namely, we have

$$\frac{\kappa(C_g)}{|C_g|} < \frac{\kappa(C_f)}{|C_f|}. \quad (17.2)$$

Now, the only difference between G_f and G_g are the edges of C_f . In particular, some edges of C_f might disappear from G_g , as they are being used in g to their full capacity. Also, all the edges in the opposite direction to C_f will be present in G_g .

Now, C_g must use at least one of the new edges in G_g , since otherwise this would contradict the minimality of C_f (i.e., we could use C_g in G_f and get a cheaper average cost cycle than C_f). Let U be the set of new edges of G_g that are being used by C_g and are not present in G_f . Let $U^{-1} = \{e^{-1} \mid e \in U\}$. Clearly, all the edges of U^{-1} appear in C_f .

Now, consider the cycle $\pi = C_f \cup C_g$. We have that the average of π is

$$\alpha = \frac{\kappa(C_f) + \kappa(C_g)}{|C_f| + |C_g|} < \max\left(\frac{\kappa(C_g)}{|C_g|}, \frac{\kappa(C_f)}{|C_f|}\right) = \mu(f),$$

by Eq. (17.2). We can write π is a union of k edge-disjoint cycles $\sigma_1, \dots, \sigma_k$ and some 2-cycles. A 2-cycle is formed by a pair of edges e and e^{-1} where $e \in U$ and $e^{-1} \in U^{-1}$. Clearly, the cost of these 2-cycles is zero. Thus,

since the cycles $\sigma_1, \dots, \sigma_k$ have no edges in U , it follows that they are all contained in G_f . We have

$$\kappa(C_f) + \kappa(C_g) = \sum_i \kappa(\sigma_i) + 0.$$

Thus, there is some non-negative integer constant c , such that

$$\alpha = \frac{\kappa(C_f) + \kappa(C_g)}{|C_f| + |C_g|} = \frac{\sum_i \kappa(\sigma_i)}{c + \sum_i |\sigma_i|} \geq \frac{\sum_i \kappa(\sigma_i)}{\sum_i |\sigma_i|},$$

since α is negative (since $\alpha < \mu(f) < 0$ as otherwise the algorithm would have already terminated). Namely, $\mu(f) > (\sum_i \kappa(\sigma_i)) / (\sum_i |\sigma_i|)$. Which implies that there is a cycle σ_r , such that $\mu(f) > \kappa(\sigma_r) / |\sigma_r|$ and this cycle is contained in G_f . But this is a contradiction to the minimality of $\mu(f)$. ■

17.5.1. Reduced cost induced by a circulation

Conceptually, consider the function $\mu(f)$ to be a potential function that increases as the algorithm progresses. To make further progress in our analysis, it would be convenient to consider a reweighting of the edges of G , in such a way that preserves the weights of cycles.

Given a circulation f , we are going to define a different cost function on the edges which is induced by f . To begin with, let $\beta(u \rightarrow v) = \kappa(u \rightarrow v) - \mu(f)$. Note, that under the cost function α , the cheapest cycle has price 0 in G (since the average cost of an edge in the cheapest average cycle has price zero). Namely, G has no negative cycles under β . Thus, for every vertex $v \in V(G)$, let $d(v)$ denote the length of the shortest walk that ends at v . The function $d(v)$ is a potential in G , by [Lemma 17.2.1](#), and as such

$$d(v) - d(u) \leq \beta(u \rightarrow v) = \kappa(u \rightarrow v) - \mu(f). \quad (17.3)$$

Next, let the *reduced cost* of (u, v) (in relation to f) be

$$\psi(u \rightarrow v) = \kappa(u \rightarrow v) + d(u) - d(v).$$

In particular, [Eq. \(17.3\)](#) implies that

$$\forall (u, v) \in E(G_f) \quad \psi(u \rightarrow v) = \kappa(u \rightarrow v) + d(u) - d(v) \geq \mu(f). \quad (17.4)$$

Namely, the reduced cost of any edge (u, v) is at least $\mu(f)$.

Note that $\psi(v \rightarrow u) = \kappa(v \rightarrow u) + d(v) - d(u) = -\kappa(u \rightarrow v) + d(v) - d(u) = -\psi(u \rightarrow v)$ (i.e., it is anti-symmetric). Also, for any cycle C in G , we have that $\kappa(C) = \psi(C)$, since the contribution of the potential $d(\cdot)$ cancels out.

The idea is that now we think about the algorithm as running with the reduced cost instead of the regular costs. Since the costs of cycles under the original cost and the reduced costs are the same, negative cycles are negative in both costs. The advantage is that the reduced cost is more useful for our purposes.

17.5.2. Bounding the number of iterations

Lemma 17.5.2. *Let f be a flow used in the i th iteration of the algorithm, let g be the flow used in the $(i + m)$ th iteration, where m is the number of edges in G . Furthermore, assume that the algorithm performed at least one more iteration on g . Then, $\mu(g) \geq (1 - 1/n)\mu(f)$.*

Proof: Let C_0, \dots, C_{m-1} be the m cycles used in computing g from f . Let $\psi(\cdot)$ be the reduced cost function induced by f .

If a cycle has only negative *reduced cost* edges, then after it is applied to the flow, one of these edges disappear from the residual graph, and the reverse edge (with positive reduced cost) appears in the residual graph. As such, if all the edges of these cycles have negative reduced costs, then G_g has no negative reduced

cost edge, and as such $\mu(\mathbf{g}) \geq 0$. But the algorithm stops as soon as the average cost cycle becomes positive. A contradiction to our assumption that the algorithm performs at least another iteration.

Let C_h be the first cycle in this sequence, such that it contains an edge e' , such that its reduced cost is positive; that is $\psi(e') \geq 0$. Note, that C_h has most n edges. We have that

$$\kappa(C_h) = \psi(C_h) = \sum_{e \in C_h} \psi(e) = \psi(e') + \sum_{e \in C_h, e \neq e'} \psi(e) \geq 0 + (|C_h| - 1)\mu(\mathbf{f}),$$

by Eq. (17.4). Namely, the average cost of C_h is

$$0 > \mu(\mathbf{f}_h) = \frac{\kappa(C_h)}{|C_h|} \geq \frac{|C_h| - 1}{|C_h|} \mu(\mathbf{f}) \geq \left(1 - \frac{1}{n}\right) \mu(\mathbf{f}).$$

The claim now easily follows from Lemma 17.5.1. ■

To bound the running time of the algorithm, we will argue that after sufficient number of iterations edges start disappearing from the residual network and never show up again in the residual network. Since there are only $2m$ possible edges, this would imply the termination of the algorithm.

Observation 17.5.3. *We have that $(1 - 1/n)^n \leq (\exp(-1/n))^n \leq 1/e$, since $1 - x \leq e^{-x}$, for all $x \geq 0$, as can be easily verified.*

Lemma 17.5.4. *Let \mathbf{f} be the circulation maintained by the algorithm at iteration ρ . Then there exists an edge e in the residual network $G_{\mathbf{f}}$ such that it never appears in the residual networks of circulations maintained by the algorithm, for iterations larger than $\rho + t$, where $t = 2nm \lceil \ln n \rceil$.*

Proof: Let \mathbf{g} be the flow used by the algorithm at iteration $\rho + t$. We define the reduced cost over the edges of G , as induced by the flow \mathbf{g} . Namely,

$$\psi(u \rightarrow v) = \kappa(u \rightarrow v) + d(u) - d(v),$$

where $d(u)$ is the length of the shortest walk ending at u where the weight of edge (u, w) is $\kappa(u \rightarrow w) - \mu(\mathbf{g})$.

Now, conceptually, we are running the algorithm using this reduced cost function over the edges, and consider the minimum average cost cycle at iteration ρ with cost $\alpha = \mu(\mathbf{f})$. There must be an edge $e \in E(G_{\mathbf{f}})$, such that $\psi(e) \leq \alpha$. (Note, that α is a negative quantity, as otherwise the algorithm would have terminated at iteration ρ .)

We have that, at iteration $\rho + t$, it holds

$$\mu(\mathbf{g}) \geq \alpha * \left(1 - \frac{1}{n}\right)^t \geq \alpha * \exp(-2m \lceil \ln n \rceil) \geq \frac{\alpha}{2n}, \quad (17.5)$$

by Lemma 17.5.2 and Observation 17.5.3 and since $\alpha < 0$. On the other hand, by Eq. (17.4), we know that for all the edges \mathbf{f} in $E(G_{\mathbf{g}})$, it holds $\psi(\mathbf{f}) \geq \mu(\mathbf{g}) \geq \alpha/2n$. As such, e can not be an edge of $G_{\mathbf{g}}$ since $\psi(e) \leq \alpha$. Namely, it must be that $\mathbf{g}(e) = \mathbf{c}(e)$.

So, assume that at a later iteration, say $\rho + t + \tau$, the edge e reappeared in the residual graph. Let \mathbf{h} be the flow at the $(\rho + t + \tau)$ th iteration, and let $G_{\mathbf{h}}$ be the residual graph. It must be that $\mathbf{h}(e) < \mathbf{c}(e) = \mathbf{g}(e)$.

Now, consider the circulation $\mathbf{i} = \mathbf{g} - \mathbf{h}$. It has a positive flow on the edge e , since $\mathbf{i}(e) = \mathbf{g}(e) - \mathbf{h}(e) > 0$. In particular, there is a directed cycle C of positive flow of \mathbf{i} in $G_{\mathbf{i}}$ that includes e , as implied by Lemma 17.3.3. Note, that Lemma 17.3.2 implies that C is also a cycle of $G_{\mathbf{h}}$.

Now, the edges of C^{-1} are present in $G_{\mathbf{g}}$. To see that, observe that for every edge $\mathbf{g} \in C$, we have that $0 < \mathbf{i}(\mathbf{g}) = \mathbf{g}(\mathbf{g}) - \mathbf{h}(\mathbf{g}) \leq \mathbf{g}(\mathbf{g}) - \ell(\mathbf{g})$. Namely, $\mathbf{g}(\mathbf{g}) > \ell(\mathbf{g})$ and as such $\mathbf{g}^{-1} \in E(G_{\mathbf{g}})$. As such, by Eq. (17.4), we have $\psi(\mathbf{g}^{-1}) \geq \mu(\mathbf{g})$. This implies

$$\forall \mathbf{g} \in C \quad \psi(\mathbf{g}) = -\psi(\mathbf{g}^{-1}) \leq -\mu(\mathbf{g}) \leq -\frac{\alpha}{2n},$$

by Eq. (17.5). Since C is a cycle of G_h , we have

$$\kappa(C) = \psi(C) = \psi(e) + \psi(C \setminus \{e\}) \leq \alpha + (|C| - 1) \cdot \left(-\frac{\alpha}{2n}\right) < \frac{\alpha}{2}.$$

Namely, the average cost of the cycle C , which is present in G_h , is $\kappa(C)/|C| < \alpha/(2n)$.

On the other hand, the minimum average cost cycle in G_h has average price $\mu(h) \geq \mu(g) \geq \frac{\alpha}{2n}$, by Lemma 17.5.1. A contradiction, since we found a cycle C in G_h which is cheaper. ■

We are now ready for the “kill” – since one edge disappears forever every $O(mn \log n)$ iterations, it follows that after $O(m^2 n \log n)$ iterations the algorithm terminates. Every iteration takes $O(mn)$ time, by Corollary 17.1.2. Putting everything together, we get the following.

Theorem 17.5.5. *Given a digraph G with n vertices and m edges, lower bound and upper bound on the flow of each edge, and a cost associated with each edge, then one can compute a valid circulation of minimum-cost in $O(m^3 n^2 \log n)$ time.*

17.6. Bibliographical Notes

The minimum average cost cycle algorithm, of Section 17.1, is due to Karp [Kar78].

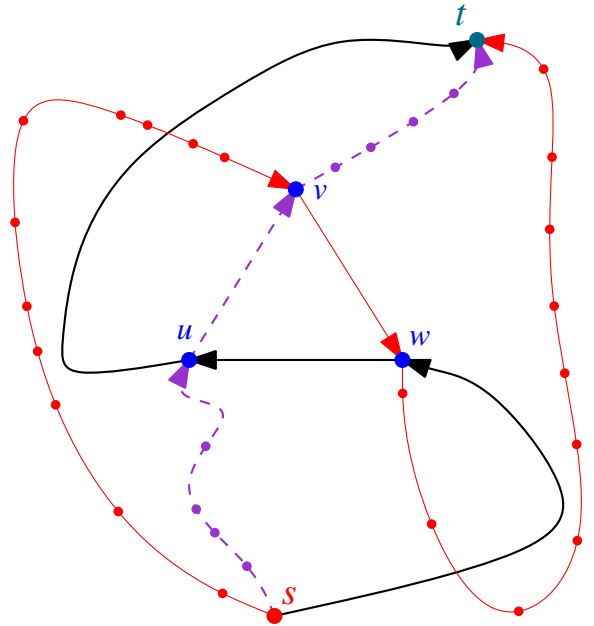
The description here follows very roughly the description of [Sch04]. The first strongly polynomial time algorithm for minimum-cost circulation is due to Éva Tardos [Tar85]. The algorithm we show is an improved version due to Andrew Goldberg and Robert Tarjan [GT89]. Initial research on this problem can be traced back to the 1940s, so it took almost fifty years to find a satisfactory solution to this problem.

Chapter 18

Network Flow VI - Min-Cost Flow Applications

18.1. Efficient Flow

A flow f would be considered to be *efficient* if it contains no cycles in it. Surprisingly, even the **Ford-Fulkerson** algorithm might generate flows with cycles in them. As a concrete example consider the picture on the right. A disc in the middle of edges indicate that we split the edge into multiple edges by introducing a vertex at this point. All edges have capacity one. For this graph, **Ford-Fulkerson** would first augment along $s \rightarrow w \rightarrow u \rightarrow t$. Next, it would augment along $s \rightarrow u \rightarrow v \rightarrow t$, and finally it would augment along $s \rightarrow v \rightarrow w \rightarrow t$. But now, there is a cycle in the flow; namely, $u \rightarrow v \rightarrow w \rightarrow u$.



One easy way to avoid such cycles is to first compute the max flow in G . Let α be the value of this flow. Next, we compute the min-cost flow in this network from s to t with flow α , where every edge has cost one. Clearly, the flow computed by the min-cost flow would not contain any such cycles. If it did contain cycles, then we can remove them by pushing flow against the cycle (i.e., reducing the flow along the cycle), resulting in a cheaper flow with the same value, which would be a contradiction. We got the following result.

Theorem 18.1.1. *Computing an efficient (i.e., acyclic) max-flow can be done in polynomial time.*

(BTW, this can also be achieved directly by removing cycles directly in the flow. Naturally, this flow might be less efficient than the min-cost flow computed.)

18.2. Efficient Flow with Lower Bounds

Consider the problem **AFWLB** (acyclic flow with lower-bounds) of computing efficient flow, where we have lower bounds on the edges. Here, we require that the returned flow would be integral, if all the numbers involved are integers. Surprisingly, this problem which looks like very similar to the problems we know how to solve efficiently is **NP-COMplete**. Indeed, consider the following problem.

Hamiltonian Path

Instance: A directed graph G and two vertices s and t .

Question: Is there a Hamiltonian path (i.e., a path visiting every vertex exactly once) in G starting at s and ending at t ?

It is easy to verify that **Hamiltonian Path** is **NP-COMplete**^①. We reduce this problem to **AFWLB** by replacing each vertex of G with two vertices and a direct edge in between them (except for the source vertex s and the sink vertex t). We set the lower-bound and capacity of each such edge to 1. Let J denote the resulting graph.

Consider now acyclic flow in J of capacity 1 from s to t which is integral. Its 0/1-flow, and as such it defines a path that visits all the special edges we created. In particular, it corresponds to a path in the original graph that starts at s , visits all the vertices of G and ends up at t . Namely, if we can compute an integral acyclic flow with lower-bounds in J in polynomial time, then we can solve **Hamiltonian path** in polynomial time. Thus, **AFWLB** is **NP-HARD**.

^①Verify that you know to do this — its a natural question for the exam.

Theorem 18.2.1. *Computing an efficient (i.e., acyclic) max-flow with lower-bounds is NP-HARD (where the flow must be integral). The related decision problem (of whether such a flow exist) is NP-COMPLETE.*

By this point you might be as confused as I am. We can model an acyclic max-flow problem with lower bounds as min-cost flow, and solve it, no? Well, not quite. The solution returned from the min-cost flow might have cycles and we can not remove them by canceling the cycles. That was only possible when there was no lower bounds on the edge capacities. Namely, the min-cost flow algorithm would return us a solution with cycles in it if there are lower bounds on the edges.

18.3. Shortest Edge-Disjoint Paths

Let G be a directed graph. We would like to compute k -edge disjoint paths between vertices s and t in the graph. We know how to do it using network flow. Interestingly, we can find the shortest k -edge disjoint paths using min-cost flow. Here, we assign cost 1 for every edge, and capacity 1 for every edge. Clearly, the min-cost flow in this graph with value k , corresponds to a set of k edge disjoint paths, such that their total length is minimized.

18.4. Covering by Cycles

Given a direct graph G , we would like to cover all its vertices by a set of cycles which are vertex disjoint. This can be done again using min-cost flow. Indeed, replace every vertex u in G by an edge (u', u'') . Where all the incoming edges to u are connected to u' and all the outgoing edges from u are now starting from u'' . Let J denote the resulting graph. All the new edges in the graph have a lower bound and capacity 1, and all the other edges have no lower bound, but their capacity is 1. We compute the minimum cost circulation in J . Clearly, this corresponds to a collection of cycles in G covering all the vertices of minimum cost.

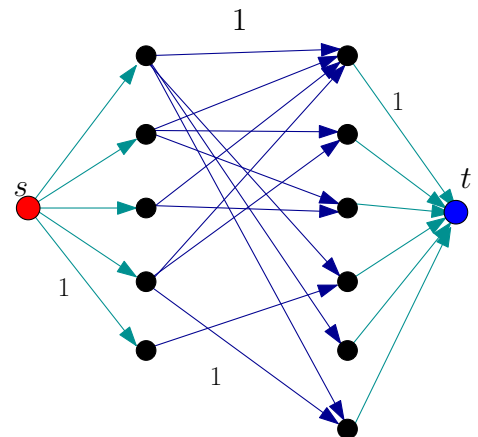
Theorem 18.4.1. *Given a directed graph G and costs on the edges, one can compute a cover of G by a collection of vertex disjoint cycles, such that the total cost of the cycles is minimized.*

18.5. Minimum weight bipartite matching

Given an undirected bipartite graph G , we would like to find the maximum cardinality matching in G that has minimum cost. The idea is to reduce this to network flow as we did in the unweighted case, and compute the maximum flow – the graph constructed is depicted on the right. Here, any edge has capacity 1. This gives us the size ϕ of the maximum matching in G . Next, we compute the min-cost flow in G with this value ϕ , where the edges connected to the source or the sink has cost zero, and the other edges are assigned their original cost in G . Clearly, the min-cost flow in this graph corresponds to a maximum cardinality min-cost flow in the original graph.

Here, we are using the fact that the flow computed is integral, and as such, it is a 0/1-flow.

Theorem 18.5.1. *Given a bipartite graph G and costs on the edges, one can compute the maximum cardinality minimum cost matching in polynomial time.*



18.6. The transportation problem

In the *transportation problem*, we are given m facilities f_1, \dots, f_m . The facility f_i contains x_i units of some commodity, for $i = 1, \dots, m$. Similarly, there are u_1, \dots, u_n customers that would like to buy this commodity. In particular, u_i would like to buy d_i units, for $i = 1, \dots, n$. To make things interesting, it costs c_{ij} to send one unit of commodity from facility i to customer j . The natural question is how to supply the demands while minimizing the total cost.

To this end, we create a bipartite graph with f_1, \dots, f_m on one side, and u_1, \dots, u_n on the other side. There is an edge from (f_i, u_j) with costs c_{ij} , for $i = 1, \dots, m$ and $j = 1, \dots, n$. Next, we create a source vertex that is connected to f_i with capacity x_i , for $i = 1, \dots, m$. Similarly, we create an edge from u_j to the sink t , with capacity d_j , for $j = 1, \dots, n$. We compute the min-cost flow in this network that pushes $\phi = \sum_j d_j$ units from the source to the sink. Clearly, the solution encodes the required optimal solution to the transportation problem.

Theorem 18.6.1. *The transportation problem can be solved in polynomial time.*

Part VI

Linear Programming

Chapter 19

Linear Programming in Low Dimensions

At the sight of the still intact city, he remembered his great international precursors and set the whole place on fire with his artillery in order that those who came after him might work off their excess energies in rebuilding.

The tin drum, Gunter Grass

In this chapter, we shortly describe (and analyze) a simple randomized algorithm for linear programming in low dimensions. Next, we show how to extend this algorithm to solve linear programming with violations. Finally, we will show how one can efficiently approximate the number constraints that one needs to violate to make a linear program feasible. This serves as a fruitful ground to demonstrate some of the techniques we visited already. Our discussion is going to be somewhat intuitive – it can be made more formal with more work.

19.1. Some geometry first

We first prove Radon's and Helly's theorems.

Definition 19.1.1. The *convex hull* of a set $P \subseteq \mathbb{R}^d$ is the set of all convex combinations of points of P ; that is,

$$CH(P) = \left\{ \sum_{i=0}^m \alpha_i s_i \mid \forall i \ s_i \in P, \alpha_i \geq 0, \text{ and } \sum_{j=1}^m \alpha_j = 1 \right\}.$$

Claim 19.1.2. Let $P = \{p_1, \dots, p_{d+2}\}$ be a set of $d + 2$ points in \mathbb{R}^d . There are real numbers $\beta_1, \dots, \beta_{d+2}$, not all of them zero, such that $\sum_i \beta_i p_i = 0$ and $\sum_i \beta_i = 0$.

Proof: Indeed, set $q_i = (p_i, 1)$, for $i = 1, \dots, d + 2$. Now, the points $q_1, \dots, q_{d+2} \in \mathbb{R}^{d+1}$ are linearly dependent, and there are coefficients $\beta_1, \dots, \beta_{d+2}$, not all of them zero, such that $\sum_{i=1}^{d+2} \beta_i q_i = 0$. Considering only the first d coordinates of these points implies that $\sum_{i=1}^{d+2} \beta_i p_i = 0$. Similarly, by considering only the $(d + 1)$ st coordinate of these points, we have that $\sum_{i=1}^{d+2} \beta_i = 0$. ■

Theorem 19.1.3 (Radon's theorem). Let $P = \{p_1, \dots, p_{d+2}\}$ be a set of $d + 2$ points in \mathbb{R}^d . Then, there exist two disjoint subsets C and D of P , such that $CH(C) \cap CH(D) \neq \emptyset$ and $C \cup D = P$.

Proof: By **Claim 19.1.2** there are real numbers $\beta_1, \dots, \beta_{d+2}$, not all of them zero, such that $\sum_i \beta_i \mathbf{p}_i = 0$ and $\sum_i \beta_i = 0$.

Assume, for the sake of simplicity of exposition, that $\beta_1, \dots, \beta_k \geq 0$ and $\beta_{k+1}, \dots, \beta_{d+2} < 0$. Furthermore, let $\mu = \sum_{i=1}^k \beta_i = -\sum_{i=k+1}^{d+2} \beta_i$. We have that

$$\sum_{i=1}^k \beta_i \mathbf{p}_i = -\sum_{i=k+1}^{d+2} \beta_i \mathbf{p}_i.$$

In particular, $v = \sum_{i=1}^k (\beta_i/\mu) \mathbf{p}_i$ is a point in $\mathcal{CH}(\{\mathbf{p}_1, \dots, \mathbf{p}_k\})$. Furthermore, for the same point v we have $v = \sum_{i=k+1}^{d+2} -(\beta_i/\mu) \mathbf{p}_i \in \mathcal{CH}(\{\mathbf{p}_{k+1}, \dots, \mathbf{p}_{d+2}\})$. We conclude that v is in the intersection of the two convex hulls, as required. \blacksquare

Theorem 19.1.4 (Helly's theorem). *Let \mathcal{F} be a set of n convex sets in \mathbb{R}^d . The intersection of all the sets of \mathcal{F} is non-empty if and only if any $d + 1$ of them has non-empty intersection.*

Proof: This theorem is the “dual” to Radon's theorem.

If the intersection of all sets in \mathcal{F} is non-empty, then any intersection of $d + 1$ of them is non-empty. As for the other direction, assume for the sake of contradiction that \mathcal{F} is the minimal set of convex sets for which the claim fails. Namely, for $m = |\mathcal{F}| > d + 1$, any subset of $m - 1$ sets of \mathcal{F} has non-empty intersection, and yet the intersection of all the sets of \mathcal{F} is empty.

As such, for $X \in \mathcal{F}$, let \mathbf{p}_X be a point in the intersection of all sets of \mathcal{F} excluding X . Let $P = \{\mathbf{p}_X \mid X \in \mathcal{F}\}$. Here $|P| = |\mathcal{F}| > d + 1$. By Radon's theorem, there is a partition of P into two disjoint sets R and Q such that $\mathcal{CH}(R) \cap \mathcal{CH}(Q) = \emptyset$. Let s be any point inside this non-empty intersection.

Let $U(R) = \{X \mid \mathbf{p}_X \in R\}$ and $U(Q) = \{X \mid \mathbf{p}_X \in Q\}$ be the two subsets of \mathcal{F} corresponding to R and Q , respectively. By definition, for $X \in U(R)$, we have that

$$\mathbf{p}_X \in \bigcap_{Y \in \mathcal{F}, Y \neq X} Y \subseteq \bigcap_{Y \in \mathcal{F} \setminus U(R)} Y = \bigcap_{Y \in U(Q)} Y,$$

since $U(Q) \cup U(R) = \mathcal{F}$ and $U(Q) \cap U(R) = \emptyset$. As such, $R \subseteq \bigcap_{Y \in U(Q)} Y$ and $Q \subseteq \bigcap_{Y \in U(R)} Y$. Now, by the convexity of the sets of \mathcal{F} , we have $\mathcal{CH}(R) \subseteq \bigcap_{Y \in U(Q)} Y$ and $\mathcal{CH}(Q) \subseteq \bigcap_{Y \in U(R)} Y$. Namely, we have

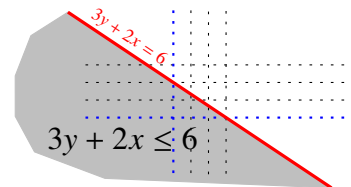
$$s \in \mathcal{CH}(R) \cap \mathcal{CH}(Q) \subseteq \left(\bigcap_{Y \in U(Q)} Y \right) \cap \left(\bigcap_{Y \in U(R)} Y \right) = \bigcap_{Y \in \mathcal{F}} Y.$$

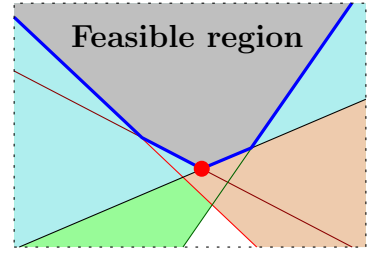
Namely, the intersection of all the sets of \mathcal{F} is not empty, a contradiction. \blacksquare

19.2. Linear programming

Assume we are given a set of n linear inequalities of the form $a_1 x_1 + \dots + a_d x_d \leq b$, where a_1, \dots, a_d, b are constants and x_1, \dots, x_d are the variables. In the **linear programming** (LP) problem, one has to find a **feasible solution**, that is, a point (x_1, \dots, x_d) for which all the linear inequalities hold. In the following, we use the shorthand **LPI** to stand for **linear programming instance**. Usually we would like to find a feasible point that maximizes a linear expression (referred to as the **target function** of the given LPI) of the form $c_1 x_1 + \dots + c_d x_d$, where c_1, \dots, c_d are prespecified constants.

The set of points complying with a linear inequality $a_1 x_1 + \dots + a_d x_d \leq b$ is a halfspace of \mathbb{R}^d having the hyperplane $a_1 x_1 + \dots + a_d x_d = b$ as a boundary; see the figure on the right. As such, the feasible region of a LPI is the intersection of n halfspaces; that is, it is a **polyhedron**. If the polyhedron is bounded, then it is a **polytope**. The linear target function is no more than specifying a direction, such that we need to find the point inside the polyhedron which is extreme in this direction. If the polyhedron is unbounded in this direction, the optimal solution is **unbounded**.





For the sake of simplicity of exposition, it will be easier to think of the direction for which one has to optimize as the negative x_d -axis direction. This can be easily realized by rotating the space such that the required direction is pointing downward. Since the feasible region is the intersection of convex sets (i.e., halfspaces), it is convex. As such, one can imagine the boundary of the feasible region as a vessel (with a convex interior). Next, we release a ball at the top of the vessel, and the ball rolls down (by “gravity” in the direction of the negative x_d -axis) till it reaches the lowest point in the vessel and gets “stuck”. This point is the optimal solution to the LPI that we are interested in computing.

In the following, we will assume that the given LPI is in general position. Namely, if we intersect k hyperplanes, induced by k inequalities in the given LPI (the hyperplanes are the result of taking each of this inequalities as an equality), then their intersection is a $(d - k)$ -dimensional affine subspace. In particular, the intersection of d of them is a point (referred to as a *vertex*). Similarly, the intersection of any $d + 1$ of them is empty.

A polyhedron defined by an LPI with n constraints might have $O(n^{\lfloor d/2 \rfloor})$ vertices on its boundary (this is known as the upper-bound theorem [Grü03]). As we argue below, the optimal solution is a vertex. As such, a naive algorithm would enumerate all relevant vertices (this is a non-trivial undertaking) and return the best possible vertex. Surprisingly, in low dimension, one can do much better and get an algorithm with linear running time.

We are interested in the best vertex of the feasible region, while this polyhedron is defined implicitly as the intersection of halfspaces, and this hints to the quandary that we are in: We are looking for an optimal vertex in a large graph that is defined implicitly. Intuitively, this is why proving the correctness of the algorithms we present here is a non-trivial undertaking (as already mentioned, we will prove correctness in the next chapter).

19.2.1. A solution and how to verify it

Observe that an optimal solution of an LPI is either a vertex or unbounded. Indeed, if the optimal solution \mathbf{p} lies in the middle of a segment s , such that s is feasible, then either one of its endpoints provides a better solution (i.e., one of them is lower in the x_d -direction than \mathbf{p}) or both endpoints of s have the same target value. But then, we can move the solution to one of the endpoints of s . In particular, if the solution lies on a k -dimensional facet F of the boundary of the feasible polyhedron (i.e., formally F is a set with affine dimension k formed by the intersection of the boundary of the polyhedron with a hyperplane), we can move it so that it lies on a $(k - 1)$ -dimensional facet F' of the feasible polyhedron, using the preceding argumentation. Using it repeatedly, one ends up in a vertex of the polyhedron or in an unbounded solution.

Thus, given an instance of LPI, the LP solver should output one of the following answers.

- (A) **Finite.** The optimal solution is finite, and the solver would provide a vertex which realizes the optimal solution.
- (B) **Unbounded.** The given LPI has an unbounded solution. In this case, the LP solver would output a ray ζ , such that the ζ lies inside the feasible region and it points down the negative x_d -axis direction.
- (C) **Infeasible.** The given LPI does not have any point which complies with all the given inequalities. In this case the solver would output $d + 1$ constraints which are infeasible on their own.

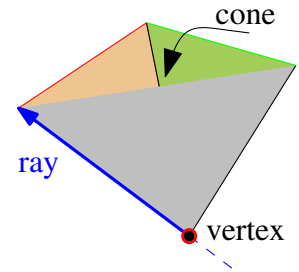
Lemma 19.2.1. *Given a set of d linear inequalities in \mathbb{R}^d , one can compute the vertex induced by the intersection of their boundaries in $O(d^3)$ time.*

Proof: Write down the system of equalities that the vertex must fulfill. It is a system of d equalities in d variables and it can be solved in $O(d^3)$ time using Gaussian elimination. ■

A **cone** is the intersection of d constraints, where its apex is the vertex associated with this set of constraints. A set of such d constraints is a **basis**. An intersection of $d - 1$ of the hyperplanes of a basis forms a line and intersecting this line with the cone of the basis forms a ray. Clipping the same line to the feasible region would yield either a segment, referred to as an **edge** of the polyhedron, or a ray (if the feasible region is an unbounded polyhedron). An edge of the polyhedron connects two vertices of the polyhedron.

As such, one can think about the boundary of the feasible region as inducing a graph – its vertices and edges are the vertices and edges of the polyhedron, respectively. Since every vertex has d hyperplanes defining it (its basis) and an adjacent edge is defined by $d - 1$ of these hyperplanes, it follows that each vertex has $\binom{d}{d-1} = d$ edges adjacent to it.

The following lemma tells us when we have an optimal vertex. While it is intuitively clear, its proof requires a systematic understanding of what the feasible region of a linear program looks like, and we delegate it to the next chapter.



Lemma 19.2.2. *Let L be a given LPI, and let \mathcal{P} denote its feasible region. Let v be a vertex of \mathcal{P} , such that all the d rays emanating from v are in the upward x_d -axis direction (i.e., the direction vectors of all these d rays have positive x_d -coordinate). Then v is the lowest (in the x_d -axis direction) point in \mathcal{P} and it is thus the optimal solution to L .*

Interestingly, when we are at a vertex v of the feasible region, it is easy to find the adjacent vertices. Indeed, compute the d rays emanating from v . For such a ray, intersect it with all the constraints of the LPI. The closest intersection point along this ray is the vertex u of the feasible region adjacent to v . Doing this naively takes $O(dn + d^{O(1)})$ time.

Lemma 19.2.2 offers a simple algorithm for computing the optimal solution for an LPI. Start from a feasible vertex of the LPI. As long as this vertex has at least one ray that points downward, follow this ray to an adjacent vertex on the feasible polytope that is lower than the current vertex (i.e., compute the d rays emanating from the current vertex, and follow one of the rays that points downward, till you hit a new vertex). Repeat this till the current vertex has all rays pointing upward, by **Lemma 19.2.2** this is the optimal solution. Up to tedious (and non-trivial) details this is the **simplex** algorithm.

We need the following lemma, whose proof is also delegated to the next chapter.

Lemma 19.2.3. *If L is an LPI in d dimensions which is not feasible, then there exist $d + 1$ inequalities in L which are infeasible on their own.*

Note that given a set of $d + 1$ inequalities, it is easy to verify (in polynomial time in d) if they are feasible or not. Indeed, compute the $\binom{d+1}{d}$ vertices formed by this set of constraints, and check whether any of these vertices are feasible (for these $d + 1$ constraints). If all of them are infeasible, then this set of constraints is infeasible.

19.3. Low-dimensional linear programming

19.3.1. An algorithm for a restricted case

There are a lot of tedious details that one has to take care of to make things work with linear programming. As such, we will first describe the algorithm for a special case and then provide the envelope required so that one can use it to solve the general case.

A vertex v is **acceptable** if all the d rays associated with it point upward (note that the vertex might not be feasible). The optimal solution (if it is finite) must be located at an acceptable vertex.

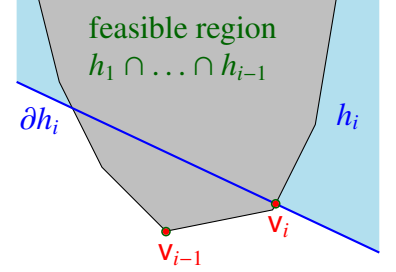
Input for the restricted case. The input for the restricted case is an LPI L , which is defined by a set of n linear inequalities in \mathbb{R}^d , and a basis $B = \{h_1, \dots, h_d\}$ of an acceptable vertex.

Let h_{d+1}, \dots, h_m be a random permutation of the remaining constraints of the LPI L .

We are looking for the lowest point in \mathbb{R}^d which is feasible for L . Our algorithm is randomized incremental. At the i th step, for $i > d$, it will maintain the optimal solution for the first i constraints. As such, in the i th step, the algorithm checks whether the optimal solution v_{i-1} of the previous iteration is still feasible with the new constraint h_i (namely, the algorithm checks if v_{i-1} is inside the halfspace defined by h_i). If v_{i-1} is still feasible, then it is still the optimal solution, and we set $v_i \leftarrow v_{i-1}$.

The more interesting case is when $v_{i-1} \notin h_i$. First, we check if the basis of v_{i-1} together with h_i forms a set of constraints which is infeasible. If so, the given LPI is infeasible, and we output $B(v_{i-1}) \cup \{h_i\}$ as the proof of infeasibility.

Otherwise, the new optimal solution must lie on the hyperplane associated with h_i . As such, we recursively compute the lowest vertex in the $(d-1)$ -dimensional polyhedron $(\partial h_i) \cap \bigcap_{j=1}^{i-1} h_j$, where ∂h_i denotes the hyperplane which is the boundary of the halfspace h_i . This is a linear program involving $i-1$ constraints, and it involves $d-1$ variables since the LPI lies on the $(d-1)$ -dimensional hyperplane ∂h_i . The solution found, v_i , is defined by a basis of $d-1$ constraints in the $(d-1)$ -dimensional subspace ∂h_i , and adding h_i to it results in an acceptable vertex that is feasible in the original d -dimensional space. We continue to the next iteration.



Clearly, the vertex v_n is the required optimal solution.

19.3.1.1. Running time analysis

Every set of d constraints is feasible and computing the vertex formed by this constraint takes $O(d^3)$ time, by [Lemma 19.2.1](#).

Let X_i be an indicator variable that is 1 if and only if the vertex v_i is recomputed in the i th iteration (by performing a recursive call). This happens only if h_i is one of the d constraints in the basis of v_i . Since there are most d constraints that define the basis and there are at least $i-d$ constraints that are being randomly ordered (as the first d slots are fixed), we have that the probability that $v_i \neq v_{i-1}$ is

$$\alpha_i = \mathbb{P}[X_i = 1] \leq \min\left(\frac{d}{i-d}, 1\right) \leq \frac{2d}{i},$$

for $i \geq d+1$, as can be easily verified.^① So, let $T(m, d)$ be the expected time to solve an LPI with m constraints in d dimensions. We have that $T(d, d) = O(d^3)$ by the above. Now, in every iteration, we need to check if the current solution lies inside the new constraint, which takes $O(d)$ time per iteration and $O(dm)$ time overall.

Now, if $X_i = 1$, then we need to update each of the $i-1$ constraints to lie on the hyperplane h_i . The hyperplane h_i defines a linear equality, which we can use to eliminate one of the variables. This takes $O(di)$ time, and we have to do the recursive call. The probability that this happens is α_i . As such, we have

$$\begin{aligned} T(m, d) &= \mathbb{E}\left[O(md) + \sum_{i=d+1}^m X_i(di + T(i-1, d-1))\right] \\ &= O(md) + \sum_{i=d+1}^m \alpha_i(di + T(i-1, d-1)) \\ &= O(md) + \sum_{i=d+1}^m \frac{2d}{i}(di + T(i-1, d-1)) \\ &= O(md^2) + \sum_{i=d+1}^m \frac{2d}{i}T(i-1, d-1). \end{aligned}$$

^①Indeed, $\frac{(d)+d}{(i-d)+d}$ lies between $\frac{d}{i-d}$ and $\frac{d}{d} = 1$.

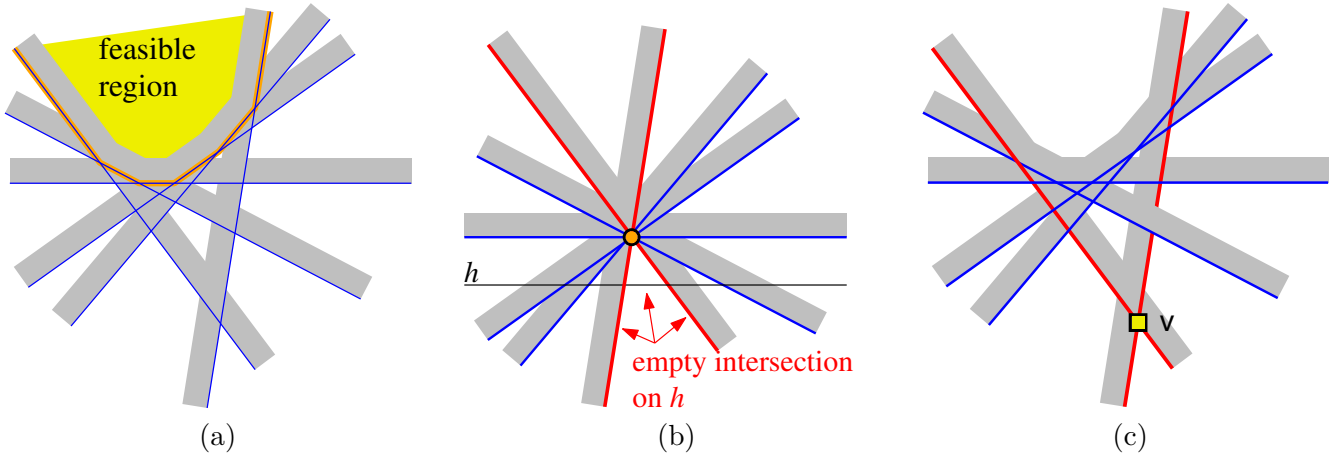


Figure 19.1: Demonstrating the algorithm for the general case: (a) given constraints and feasible region, (b) constraints moved to pass through the origin, and (c) the resulting acceptable vertex v .

Guessing that $T(m, d) \leq c_d m$, we have that

$$T(m, d) \leq \widehat{c}_1 m d^2 + \sum_{i=d+1}^m \frac{2d}{i} c_{d-1} (i-1) \leq \widehat{c}_1 m d^2 + \sum_{i=d+1}^m 2d c_{d-1} = (\widehat{c}_1 d^2 + 2d c_{d-1}) m,$$

where \widehat{c}_1 is some absolute constant. We need that $\widehat{c}_1 d^2 + 2c_{d-1} d \leq c_d$, which holds for $c_d = O((3d)^d)$ and $T(m, d) = O((3d)^d m)$.

Lemma 19.3.1. *Given an LPI with n constraints in d dimensions and an acceptable vertex for this LPI, then can compute the optimal solution in expected $O((3d)^d n)$ time.*

19.3.2. The algorithm for the general case

Let L be the given LPI, and let L' be the instance formed by translating all the constraints so that they pass through the origin. Next, let h be the hyperplane $x_d = -1$. Consider a solution to the LP L' when restricted to h . This is a $(d-1)$ -dimensional instance of linear programming, and it can be solved recursively.

If the recursive call on $L' \cap h$ returned no solution, then the d constraints that prove that the LP L' is infeasible on h corresponds to a basis in L of a vertex v which is acceptable in the original LPI. Indeed, as we move these d constraints to the origin, their intersection on h is empty (i.e., the “quadrant” that their intersection forms is unbounded only in the upward direction). As such, we can now apply the algorithm of [Lemma 19.3.1](#) to solve the given LPI. See [Figure 19.1](#).

If there is a solution to $L' \cap h$, then it is a vertex v on h which is feasible. Thus, consider the original set of $d-1$ constraints in L that corresponds to the basis B of v . Let ℓ be the line formed by the intersection of the hyperplanes of B . It is now easy to verify that the intersection of the feasible region with this line is an unbounded ray, and the algorithm returns this unbounded (downward oriented) ray, as a proof that the LPI is unbounded.

Theorem 19.3.2. *Given an LP instance with n constraints defined over d variables, it can be solved in expected $O((3d)^d n)$ time.*

Proof: The expected running time is

$$S(n, d) = O(nd) + S(n, d-1) + T(m, d),$$

where $T(m, d)$ is the time to solve an LP in the restricted case of [Section 19.3.1](#). Indeed, we first solve the problem on the $(d - 1)$ -dimensional subspace $h \equiv x_d = -1$. This takes $O(dn) + S(n, d - 1)$ time (we need to rewrite the constraints for the lower-dimensional instance, and that takes $O(dn)$ time). If the solution on h is feasible, then the original LPI has an unbounded solution, and we return it. Otherwise, we obtained an acceptable vertex, and we can use the special case algorithm on the original LPI. Now, the solution to this recurrence is $O((3d)^d n)$; see [Lemma 19.3.1](#). ■

Chapter 20

Linear Programming

20.1. Introduction and Motivation

In the VCR/guns/nuclear-bombs/napkins/star-wars/professors/butter/mice problem, the benevolent dictator, Biga Piguinus, of Penguina (a country in south Antarctica having 24 million penguins under its control) has to decide how to allocate her empire resources to the maximal benefit of her penguins. In particular, she has to decide how to allocate the money for the next year budget. For example, buying a nuclear bomb has a tremendous positive effect on security (the ability to destruct yourself completely together with your enemy induces a peaceful serenity feeling in most people). Guns, on the other hand, have a weaker effect. Penguina (the state) has to supply a certain level of security. Thus, the allocation should be such that:

$$x_{gun} + 1000 * x_{nuclear-bomb} \geq 1000,$$

where x_{guns} is the number of guns constructed, and $x_{nuclear-bomb}$ is the number of nuclear-bombs constructed. On the other hand,

$$100 * x_{gun} + 1000000 * x_{nuclear-bomb} \leq x_{security}$$

where $x_{security}$ is the total Penguina is willing to spend on security, and 100 is the price of producing a single gun, and 1,000,000 is the price of manufacturing one nuclear bomb. There are a lot of other constrains of this type, and Biga Piguinus would like to solve them, while minimizing the total money allocated for such spending (the less spent on budget, the larger the tax cut).

More formally, we have a (potentially large) number of variables: x_1, \dots, x_n and a (potentially large) system of linear inequalities. We will refer to such an inequality as a *constraint*. We would like to decide if there is an assignment of values to x_1, \dots, x_n where all these inequalities are satisfied. Since there might be infinite number of such solutions, we want the solution that maximizes some linear quantity. See the instance on the right.

$a_{11}x_1 + \dots + a_{1n}x_n \leq b_1$ $a_{21}x_1 + \dots + a_{2n}x_n \leq b_2$ \vdots $a_{m1}x_1 + \dots + a_{mn}x_n \leq b_m$ $\max \quad c_1x_1 + \dots + c_nx_n.$

The linear target function we are trying to maximize is known as the *objective function* of the linear program. Such a problem is an instance of *linear programming*. We refer to linear programming as LP.

20.1.1. History

Linear programming can be traced back to the early 19th century. It started in earnest in 1939 when L. V. Kantorovich noticed the importance of certain type of Linear Programming problems. Unfortunately, for several

$$\begin{array}{ll}
\forall (u, v) \in E & 0 \leq x_{u \rightarrow v} \\
& x_{u \rightarrow v} \leq c(u \rightarrow v) \\
\forall v \in V \setminus \{s, t\} & \sum_{(u, v) \in E} x_{u \rightarrow v} - \sum_{(v, w) \in E} x_{v \rightarrow w} \leq 0 \\
& \sum_{(u, v) \in E} x_{u \rightarrow v} - \sum_{(v, w) \in E} x_{v \rightarrow w} \geq 0 \\
\text{maximizing} & \sum_{(s, u) \in E} x_{s \rightarrow u}
\end{array}$$

Figure 20.1

$$\begin{array}{ll}
\max & \sum_{j=1}^n c_j x_j \\
\text{subject to} & \sum_{j=1}^n a_{ij} x_j \leq b_i \\
& \text{for } i = 1, 2, \dots, m.
\end{array}$$

Figure 20.2

years, Kantorovich work was unknown in the west and unnoticed in the east.

Dantzig, in 1947, invented the simplex method for solving LP problems for the US Air force planning problems.

T. C. Koopmans, in 1947, showed that LP provide the right model for the analysis of classical economic theories.

In 1975, both Koopmans and Kantorovich got the Nobel prize of economics. Dantzig probably did not get it because his work was too mathematical. That is how it goes. Kantorovich was the only the Russian economist that got the Nobel prize^①.

20.1.2. Network flow via linear programming

To see the impressive expressive power of linear programming, we next show that network flow can be solved using linear programming. Thus, we are given an instance of max flow; namely, a network flow $G = (V, E)$ with source s and sink t , and capacities $c(\cdot)$ on the edges. We would like to compute the maximum flow in G .

To this end, for an edge $(u, v) \in E$, let $x_{u \rightarrow v}$ be a variable which is the amount of flow assign to (u, v) in the maximum flow. We demand that $0 \leq x_{u \rightarrow v}$ and $x_{u \rightarrow v} \leq c(u \rightarrow v)$ (flow is non negative on edges, and it comply with the capacity constraints). Next, for any vertex v which is not the source or the sink, we require that $\sum_{(u, v) \in E} x_{u \rightarrow v} = \sum_{(v, w) \in E} x_{v \rightarrow w}$ (this is conservation of flow). Note, that an equality constraint $a = b$ can be rewritten as two inequality constraints $a \leq b$ and $b \leq a$. Finally, under all these constraints, we are interest in the maximum flow. Namely, we would like to maximize the quantity $\sum_{(s, u) \in E} x_{s \rightarrow u}$. Clearly, putting all these constraints together, we get the linear program depicted in **Figure 20.1**.

It is not too hard to write down min-cost network flow using linear programming.

20.2. The Simplex Algorithm

20.2.1. Linear program where all the variables are positive

We are given a LP, depicted in [Figure 20.2](#), where a variable can have any real value. As a first step to solving it, we would like to rewrite it, such that every variable is non-negative. This is easy to do, by replacing a variable x_i by two new variables x'_i and x''_i , where $x_i = x'_i - x''_i$, $x'_i \geq 0$ and $x''_i \geq 0$. For example, the (trivial) linear program containing the single constraint $2x + y \geq 5$ would be replaced by the following LP: $2x' - 2x'' + y' - y'' \geq 5$, $x' \geq 0$, $y' \geq 0$, $x'' \geq 0$ and $y'' \geq 0$.

Lemma 20.2.1. *Given an instance I of LP, one can rewrite it into an equivalent LP, such that all the variables must be non-negative. This takes linear time in the size of I .*

20.2.2. Standard form

Using [Lemma 20.2.1](#), we can now require a LP to be specified using only positive variables. This is known as *standard form*.

A linear program in standard form.

$$\begin{aligned} \max \quad & \sum_{j=1}^n c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^n a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \dots, m \\ & x_j \geq 0 \quad \text{for } j = 1, \dots, n. \end{aligned}$$

A linear program in standard form.

(Matrix notation.)

$$\begin{aligned} \max \quad & c^T x \\ \text{subject to} \quad & Ax \leq b. \\ & x \geq 0. \end{aligned}$$

Here the matrix notation rises, by setting

$$c = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}, b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}, A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2(n-1)} & a_{2n} \\ \vdots & \dots & \dots & \dots & \vdots \\ a_{(m-1)1} & a_{(m-1)2} & \dots & a_{(m-1)(n-1)} & a_{(m-1)n} \\ a_{m1} & a_{m2} & \dots & a_{m(n-1)} & a_{mn} \end{pmatrix}, \text{ and } x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}.$$

Note, that c, b and A are prespecified, and x is the vector of unknowns that we have to solve the LP for.

In the following in order to solve the LP, we are going to do a long sequence of rewritings till we reach the optimal solution.

20.2.3. Slack Form

We next rewrite the LP into *slack form*. It is a more convenient^② form for describing the [Simplex](#) algorithm for solving LP.

Specifically, one can rewrite a LP, so that every inequality becomes equality, and all variables must be positive; namely, the new LP will have a form depicted on the right (using matrix notation). To this end, we introduce new variables (*slack variables*) rewriting the inequality

$$\begin{aligned} \max \quad & c^T x \\ \text{subject to} \quad & Ax = b. \\ & x \geq 0. \end{aligned}$$

$$\sum_{i=1}^n a_i x_i \leq b$$

^①There were other economists that were born in Russia, but lived in the west that got the Nobel prize – Leonid Hurwicz for example.

^②The word *convenience* is used here in the most liberal interpretation possible.

as

$$x_{n+1} = b - \sum_{i=1}^n a_i x_i$$

$$x_{n+1} \geq 0.$$

Intuitively, the value of the slack variable x_{n+1} encodes how far is the original inequality for holding with equality.

Now, we have a special variable for each inequality in the LP (this is x_{n+1} in the above example). These variables are special, and would be called **basic variables**. All the other variables on the right side are **nonbasic variables** (original isn't it?). A LP in this form is in **slack form**.

The slack form is defined by a tuple (N, B, A, b, c, v) .

Linear program in slack form.

$$\begin{aligned} \max \quad & z = v + \sum_{j \in N} c_j x_j, \\ \text{s.t.} \quad & x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B, \\ & x_i \geq 0, \quad \forall i = 1, \dots, n + m. \end{aligned}$$

B - Set of indices of basic variables
 N - Set of indices of nonbasic variables
 $n = |N|$ - number of original variables
 b, c - two vectors of constants
 $m = |B|$ - number of basic variables
 (i.e., number of inequalities)
 $A = \{a_{ij}\}$ - The matrix of coefficients
 $N \cup B = \{1, \dots, n + m\}$
 v - objective function constant.

Exercise 20.2.2. Show that any linear program can be transformed into equivalent slack form.

Example 20.2.3. Consider the following LP which is in slack form, and its translation into the tuple (N, B, A, b, c, v) .

$$\begin{aligned} \max \quad & z = 29 - \frac{1}{9}x_3 - \frac{1}{9}x_5 - \frac{2}{9}x_6 \\ & x_1 = 8 + \frac{1}{6}x_3 + \frac{1}{6}x_5 - \frac{1}{3}x_6 \\ & x_2 = 4 - \frac{8}{3}x_3 - \frac{2}{3}x_5 + \frac{1}{3}x_6 \\ & x_4 = 18 - \frac{1}{2}x_3 + \frac{1}{2}x_5 \end{aligned}$$

$$\begin{aligned} B &= \{1, 2, 4\}, N = \{3, 5, 6\} \\ A &= \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix} \\ b &= \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix} \quad c = \begin{pmatrix} c_3 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} -1/9 \\ -1/9 \\ -2/9 \end{pmatrix} \\ v &= 29. \end{aligned}$$

Note that indices depend on the sets N and B , and also that the entries in A are negation of what they appear in the slack form.

20.2.4. The Simplex algorithm by example

Before describing the **Simplex** algorithm in detail, it would be beneficial to derive it on an example. So, consider the following LP.

$$\begin{aligned} \max \quad & 5x_1 + 4x_2 + 3x_3 \\ \text{s.t.} \quad & 2x_1 + 3x_2 + x_3 \leq 5 \\ & 4x_1 + x_2 + 2x_3 \leq 11 \\ & 3x_1 + 4x_2 + 2x_3 \leq 8 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Next, we introduce slack variables, for example, rewriting $2x_1 + 3x_2 + x_3 \leq 5$ as the constraints: $w_1 \geq 0$ and $w_1 = 5 - 2x_1 - 3x_2 - x_3$. The resulting LP in slack form is

$$\begin{aligned} \max \quad & z = 5x_1 + 4x_2 + 3x_3 \\ \text{s.t.} \quad & w_1 = 5 - 2x_1 - 3x_2 - x_3 \\ & w_2 = 11 - 4x_1 - x_2 - 2x_3 \\ & w_3 = 8 - 3x_1 - 4x_2 - 2x_3 \\ & x_1, x_2, x_3, w_1, w_2, w_3 \geq 0 \end{aligned}$$

Here w_1, w_2, w_3 are the slack variables. Note also that they are currently also the basic variables. Consider the slack representation trivial solution, where all the non-basic variables are assigned zero; namely, $x_1 = x_2 = x_3 = 0$. We then have that $w_1 = 5$, $w_2 = 11$ and $w_3 = 8$. Fortunately for us, this is a feasible solution, and the associated objective value is $z = 0$.

We are interested in further improving the value of the objective function (i.e., z), while still having a feasible solution. Inspecting carefully the above LP, we realize that all the basic variables $w_1 = 5$, $w_2 = 11$ and $w_3 = 8$ have values which are strictly larger than zero. Clearly, if we change the value of one non-basic variable a bit, all the basic variables would remain positive (we are thinking about the above system as being function of the nonbasic variables x_1, x_2 and x_3). So, consider the objective function $z = 5x_1 + 4x_2 + 3x_3$. Clearly, if we increase the value of x_1 , from its current zero value, then the value of the objective function would go up, since the coefficient of x_1 for z is a positive number (5 in our example).

Deciding how much to increase the value of x_1 is non-trivial. Indeed, as we increase the value of x_1 , the the solution might stop being feasible (although the objective function values goes up, which is a good thing). So, let us increase x_1 as much as possible without violating any constraint. In particular, for $x_2 = x_3 = 0$ we have that

$$\begin{aligned} w_1 &= 5 - 2x_1 - 3x_2 - x_3 = 5 - 2x_1 \\ w_2 &= 11 - 4x_1 - x_2 - 2x_3 = 11 - 4x_1 \\ w_3 &= 8 - 3x_1 - 4x_2 - 2x_3 = 8 - 3x_1. \end{aligned}$$

We want to increase x_1 as much as possible, as long as w_1, w_2, w_3 are non-negative. Formally, the constraints are that

$$\begin{aligned} w_1 &= 5 - 2x_1 \geq 0, \\ w_2 &= 11 - 4x_1 \geq 0, \\ \text{and } w_3 &= 8 - 3x_1 \geq 0. \end{aligned}$$

This implies that whatever value we pick for x_1 it must comply with the inequalities $x_1 \leq 2.5$, $x_1 \leq 11/4 = 2.75$ and $x_1 \leq 8/3 = 2.66$. We select as the value of x_1 the largest value that still comply with all these conditions. Namely, $x_1 = 2.5$. Putting it into the system, we now have a solution which is

$$x_1 = 2.5, x_2 = 0, x_3 = 0, w_1 = 0, w_2 = 1, w_3 = 0.5 \quad \Rightarrow \quad z = 5x_1 + 4x_2 + 3x_3 = 12.5.$$

As such, all the variables are non-negative and this solution is feasible. Furthermore, this is a better solution than the previous one, since the old solution had (the objective function) value $z = 0$.

What really happened? One zero nonbasic variable (i.e., x_1) became non-zero, and one basic variable became zero (i.e., w_1). It is natural now to want to exchange between the nonbasic variable x_1 (since it is no longer zero) and the basic variable w_1 . This way, we will preserve the invariant, that the current solution we maintain is the one where all the nonbasic variables are assigned zero.

So, consider the equality in the LP that involves w_1 , that is $w_1 = 5 - 2x_1 - 3x_2 - x_3$. We can rewrite this equation, so that x_1 is on the left side:

$$x_1 = 2.5 - 0.5w_1 - 1.5x_2 - 0.5x_3. \tag{20.1}$$

The problem is that x_1 still appears in the right side of the equations for w_2 and w_3 in the LP. We observe, however, that any appearance of x_1 can be replaced by substituting it by the expression on the right side of Eq. (20.1). Collecting similar terms, we get the following equivalent LP:

$$\begin{aligned} \max \quad z &= 12.5 - 2.5w_1 - 3.5x_2 + 0.5x_3 \\ x_1 &= 2.5 - 0.5w_1 - 1.5x_2 - 0.5x_3 \\ w_2 &= 1 + 2w_1 + 5x_2 \\ w_3 &= 0.5 + 1.5w_1 + 0.5x_2 - 0.5x_3. \end{aligned}$$

Note, that the nonbasic variables are now $\{w_1, x_2, x_3\}$ and the basic variables are $\{x_1, w_2, w_3\}$. In particular, the trivial solution, of assigning zero to all the nonbasic variables is still feasible; namely we set $w_1 = x_2 = x_3 = 0$. Furthermore, the value of this solution is 12.5.

This rewriting step, we just did, is called *pivoting*. And the variable we pivoted on is x_1 , as x_1 was transferred from being a nonbasic variable into a basic variable.

We would like to continue pivoting till we reach an optimal solution. We observe, that we can not pivot on w_1 , since if we increase the value of w_1 then the objective function value goes down, since the coefficient of w_1 is -2.5 . Similarly, we can not pivot on x_2 since its coefficient in the objective function is -3.5 . Thus, we can only pivot on x_3 since its coefficient in the objective function is 0.5 , which is a positive number.

Checking carefully, it follows that the maximum we can increase x_3 is to 1, since then w_3 becomes zero. Thus, rewriting the equality for w_3 in the LP; that is,

$$w_3 = 0.5 + 1.5w_1 + 0.5x_2 - 0.5x_3,$$

for x_3 , we have

$$x_3 = 1 + 3w_1 + x_2 - 2w_3,$$

Substituting this into the LP, we get the following LP.

$$\begin{aligned} \max \quad z &= 13 - w_1 - 3x_2 - w_3 \\ \text{s.t.} \quad x_1 &= 2 - 2w_1 - 2x_2 + w_3 \\ w_2 &= 1 + 2w_1 + 5x_2 \\ x_3 &= 1 + 3w_1 + x_2 - 2w_3 \end{aligned}$$

Can we further improve the current (trivial) solution that assigns zero to all the nonbasic variables? (Here the nonbasic variables are $\{w_1, x_2, w_3\}$.)

The resounding answer is no. We had reached the optimal solution. Indeed, all the coefficients in the objective function are negative (or zero). As such, the trivial solution (all nonbasic variables get zero) is maximal, as they must all be non-negative, and increasing their value decreases the value of the objective function. So we better stop.

Intuition. The crucial observation underlining our reasoning is that at each stage we had to replace the LP by a completely equivalent LP. In particular, any feasible solution to the original LP would be feasible for the final LP (and vice versa). Furthermore, they would have exactly the same objective function value. However, in the final LP, we get an objective function that can not be improved for any feasible point, and we stopped. Thus, we found the optimal solution to the linear program.

This gives a somewhat informal description of the simplex algorithm. At each step we pivot on a nonbasic variable that improves our objective function till we reach the optimal solution. There is a problem with our description, as we assumed that the starting (trivial) solution of assigning zero to the nonbasic variables is feasible. This is of course might be false. Before providing a formal (and somewhat tedious) description of the above algorithm, we show how to resolve this problem.

20.2.4.1. Starting somewhere

$$\begin{aligned} \max \quad & z = v + \sum_{j \in N} c_j x_j, \\ \text{s.t.} \quad & x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B, \\ & x_i \geq 0, \quad \forall i = 1, \dots, n + m. \end{aligned}$$

We had transformed a linear programming problem into slack form. Intuitively, what the **Simplex** algorithm is going to do, is to start from a feasible solution and start walking around in the feasible region till it reaches the best possible point as far as the objective function is concerned. But *maybe* the linear program L is not feasible at all (i.e., no

solution exists.). Let L be a linear program (in slack form depicted on the left. Clearly, if we set all $x_i = 0$ if $i \in N$ then this determines the values of the basic variables. If they are all positive, we are done, as we found a feasible solution. The problem is that they might be negative.

We generate a new LP problem L' from L . This LP $L' = \text{Feasible}(L)$ is depicted on the right. Clearly, if we pick $x_j = 0$ for all $j \in N$ (all the nonbasic variables), and a value large enough for x_0 then all the basic variables would be non-negatives, and as such, we have found a feasible solution for L' . Let $\text{LPStartSolution}(L')$ denote this easily computable feasible solution.

$$\begin{aligned} \min \quad & x_0 \\ \text{s.t.} \quad & x_i = x_0 + b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B, \\ & x_i \geq 0, \quad \forall i = 1, \dots, n + m. \end{aligned}$$

We can now use the **Simplex** algorithm we described to find this optimal solution to L' (because we have a feasible solution to start from!).

Lemma 20.2.4. *The LP L is feasible if and only if the optimal objective value of LP L' is zero.*

Proof: A feasible solution to L is immediately an optimal solution to L' with $x_0 = 0$, and vice versa. Namely, given a solution to L' with $x_0 = 0$ we can transform it to a feasible solution to L by removing x_0 . ■

One technicality that is ignored above, is that the starting solution we have for L' , generated by $\text{LPStartSolution}(L)$ is not legal as far as the slack form is concerned, because the non-basic variable x_0 is assigned a non-zero value. However, this can be easily resolved by immediately pivoting on x_0 when we run the **Simplex** algorithm. Namely, we first try to decrease x_0 as much as possible.

Chapter 21

Linear Programming II

21.1. The **Simplex** Algorithm in Detail

B - Set of indices of basic variables
 N - Set of indices of nonbasic variables
 $n = |N|$ - number of original variables
 b, c - two vectors of constants
 $m = |B|$ - number of basic variables (i.e., number of inequalities)
 $A = \{a_{ij}\}$ - The matrix of coefficients
 $N \cup B = \{1, \dots, n + m\}$
 v - objective function constant.

(i)

$$\begin{aligned}
 \max \quad & z = v + \sum_{j \in N} c_j x_j, \\
 \text{s.t.} \quad & x_i = b_i - \sum_{j \in N} a_{ij} x_j \text{ for } i \in B, \\
 & x_i \geq 0, \quad \forall i = 1, \dots, n + m.
 \end{aligned}$$

(ii)

Figure 21.2: A linear program in slack form is specified by a tuple (N, B, A, b, c, v) .

The **Simplex** algorithm is presented on the right. We assume that we are given **SimplexInner**, a black box that solves a LP if the trivial solution of assigning zero to all the nonbasic variables is feasible. We remind the reader that $L' = \text{Feasible}(L)$ returns a new LP for which we have an easy feasible solution. This is done by introducing a new variable x_0 into the LP, where the original LP \widehat{L} is feasible if and only if the new LP L has a feasible solution with $x_0 = 0$. As such, we set the target function in L to be minimizing x_0 .

We now apply **SimplexInner** to L' and the easy solution computed for L' by **LPStartSolution**(L'). If $x_0 > 0$ in the optimal solution for L' then there is no feasible solution for L , and we exit. Otherwise, we found a feasible solution to L , and we use it as the starting point for **SimplexInner** when it is applied to L .

Thus, in the following, we have to describe **SimplexInner** - a procedure to solve an LP in slack form, when we start from a feasible solution defined by the nonbasic variables assigned value zero.

One technicality that is ignored above, is that the starting solution we have for L' , generated by **LPStartSolution**(L) is not legal as far as the slack form is concerned, because the non-basic variable x_0 is assigned a non-zero value. However, this can be easily resolve by immediately pivot on x_0 when we execute (*) in **Figure 21.1**. Namely, we first try to decrease x_0 as much as possible.

21.2. The **SimplexInner** Algorithm

We next describe the **SimplexInner** algorithm.

We remind the reader that the LP is given to us in slack form, see **Figure 21.2**. Furthermore, we assume that the trivial solution $x = \tau$, which is assigning all nonbasic variables zero, is feasible. In particular, we immediately get the objective value for this solution from the notation which is v .

Assume, that we have a nonbasic variable x_e that appears in the objective function, and furthermore its coefficient c_e is positive in (the objective function), which is $z = v + \sum_{j \in N} c_j x_j$. Formally, we pick e to be one of the indices of

$$\{j \mid c_j > 0, j \in N\}.$$

The variable x_e is the **entering variable** variable (since it is going to join the set of basic variables).

Clearly, if we increase the value of x_e (from the current value of 0 in τ) then one of the basic variables is going to vanish (i.e., become zero). Let x_l be this basic variable. We increase the value of x_e (the **entering**

```

Simplex(  $\widehat{L}$  a LP )
  Transform  $\widehat{L}$  into slack form.
  Let  $L$  be the resulting slack form.
   $L' \leftarrow \text{Feasible}(L)$ 
   $x \leftarrow \text{LPStartSolution}(L')$ 
   $x' \leftarrow \text{SimplexInner}(L', x)$  (*)
   $z \leftarrow$  objective function value of  $x'$ 
  if  $z > 0$  then
    return "No solution"
   $x'' \leftarrow \text{SimplexInner}(L, x')$ 
  return  $x''$ 
  
```

Figure 21.1: The **Simplex** algorithm.

variable) till x_l (the *leaving* variable) becomes zero.

Setting all nonbasic variables to zero, and letting x_e grow, implies that $x_i = b_i - a_{ie}x_e$, for all $i \in B$.

All those variables must be non-negative, and thus we require that $\forall i \in B$ it holds $x_i = b_i - a_{ie}x_e \geq 0$. Namely, $x_e \leq (b_i/a_{ie})$ or alternatively, $\frac{1}{x_e} \geq \frac{a_{ie}}{b_i}$. Namely, $\frac{1}{x_e} \geq \max_{i \in B} \frac{a_{ie}}{b_i}$ and, the largest value of x_e which is still feasible is

$$U = \left(\max_{i \in B} \frac{a_{ie}}{b_i} \right)^{-1}.$$

We pick l (the index of the leaving variable) from the set all basic variables that vanish to zero when $x_e = U$. Namely, l is from the set

$$\left\{ j \mid \frac{a_{je}}{b_j} = U \text{ where } j \in B \right\}.$$

Now, we know x_e and x_l . We rewrite the equation for x_l in the LP so that it has x_e on the left size. Formally, we do

$$x_l = b_l - \sum_{j \in N} a_{lj}x_j \quad \Rightarrow \quad x_e = \frac{b_l}{a_{le}} - \sum_{j \in N \cup \{l\}} \frac{a_{lj}}{a_{le}}x_j, \quad \text{where } a_{ll} = 1.$$

We need to remove all the appearances on the right side of the LP of x_e . This can be done by substituting x_e into the other equalities, using the above equality. Alternatively, we do beforehand Gaussian elimination, to remove any appearance of x_e on the right side of the equalities in the LP (and also from the objective function) replaced by appearances of x_l on the left side, which we then transfer to the right side.

In the end of this process, we have a new *equivalent* LP where the basic variables are $B' = (B \setminus \{l\}) \cup \{e\}$ and the non-basic variables are $N' = (N \setminus \{e\}) \cup \{l\}$.

In end of this *pivoting* stage the LP objective function value had increased, and as such, we made progress. Note, that the linear system is completely defined by which variables are basic, and which are non-basic. Furthermore, pivoting never returns to a combination (of basic/non-basic variable) that was already visited. Indeed, we improve the value of the objective function in each pivoting stage. Thus, we can do at most

$$\binom{n+m}{n} \leq \left(\frac{n+m}{n} \cdot e \right)^n$$

pivoting steps. And this is close to tight in the worst case (there are examples where 2^n pivoting steps are needed).

Each pivoting step takes polynomial time in n and m . Thus, the overall running time of **Simplex** is exponential in the worst case. However, in practice, **Simplex** is extremely fast.

21.2.1. Degeneracies

If you inspect carefully the **Simplex** algorithm, you would notice that it might get stuck if one of the b_i s is zero. This corresponds to a case where $> m$ hyperplanes passes through the same point. This might cause the effect that you might not be able to make any progress at all in pivoting.

There are several solutions, the simplest one is to add tiny random noise to each coefficient. You can even do this symbolically. Intuitively, the degeneracy, being a local phenomena on the polytope disappears with high probability.

The larger danger, is that you would get into cycling; namely, a sequence of pivoting operations that do not improve the objective function, and the bases you get are cyclic (i.e., infinite loop).

There is a simple scheme based on using the symbolic perturbation, that avoids cycling, by carefully choosing what is the leaving variable. This is described in detail in **Section 21.6**.

There is an alternative approach, called *Bland's rule*, which always choose the lowest index variable for entering and leaving out of the possible candidates. We will not prove the correctness of this approach here.

21.2.2. Correctness of linear programming

Definition 21.2.1. A solution to an LP is a *basic solution* if it the result of setting all the nonbasic variables to zero.

Note that the *Simplex* algorithm deals only with basic solutions. In particular we get the following.

Theorem 21.2.2 (Fundamental theorem of Linear Programming.). *For an arbitrary linear program, the following statements are true:*

- (A) *If there is no optimal solution, the problem is either infeasible or unbounded.*
- (B) *If a feasible solution exists, then a basic feasible solution exists.*
- (C) *If an optimal solution exists, then a basic optimal solution exists.*

Proof: Proof is constructive by running the simplex algorithm. ■

21.2.3. On the ellipsoid method and interior point methods

The *Simplex* algorithm has exponential running time in the worst case.

The ellipsoid method is *weakly* polynomial (namely, it is polynomial in the number of bits of the input). Khachian in 1979 came up with it. It turned out to be completely useless in practice.

In 1984, Karmakar came up with a different method, called the *interior-point method* which is also weakly polynomial. However, it turned out to be quite useful in practice, resulting in an arm race between the interior-point method and the simplex method.

The question of whether there is a *strongly* polynomial time algorithm for linear programming, is one of the major open questions in computer science.

21.3. Duality and Linear Programming

Every linear program L has a *dual linear program* L' . Solving the dual problem is essentially equivalent to solving the *primal linear program* (i.e., the original) LP.

21.3.1. Duality by Example

Consider the linear program L depicted on the right (Figure 21.3). Note, that any feasible solution, gives us a lower bound on the maximal value of the target function, denoted by η . In particular, the solution $x_1 = 1, x_2 = x_3 = 0$ is feasible, and implies $z = 4$ and thus $\eta \geq 4$.

Similarly, $x_1 = x_2 = 0, x_3 = 3$ is feasible and implies that $\eta \geq z = 9$.

We might be wondering how close is this solution to the optimal solution? In particular, if this solution is very close to the optimal solution, we might be willing to stop and be satisfied with it.

Let us add the first inequality (multiplied by 2) to the second inequality (multiplied by 3). Namely, we add the two inequalities:

max	$z = 4x_1 + x_2 + 3x_3$
s.t.	$x_1 + 4x_2 \leq 1$
	$3x_1 - x_2 + x_3 \leq 3$
	$x_1, x_2, x_3 \geq 0$

Figure 21.3: The linear program L .

$$\begin{aligned} 2(x_1 + 4x_2) &\leq 2(1) \\ +3(3x_1 - x_2 + x_3) &\leq 3(3). \end{aligned}$$

The resulting inequality is

$$11x_1 + 5x_2 + 3x_3 \leq 11. \tag{21.1}$$

Note, that this inequality must hold for any feasible solution of L . Now, the objective function is $z = 4x_1 + x_2 + 3x_3$ and x_1, x_2 and x_3 are all non-negative, and the inequality of Eq. (21.1) has larger coefficients than all the coefficients of the target function, for the corresponding variables. It thus follows, that for any feasible solution, we have

$$z = 4x_1 + x_2 + 3x_3 \leq 11x_1 + 5x_2 + 3x_3 \leq 11,$$

since all the variables are non-negative. As such, the optimal value of the LP L is somewhere between 9 and 11.

We can extend this argument. Let us multiply the first inequality by y_1 and second inequality by y_2 and add them up. We get:

$y_1(x_1$	+	$4x_2$)	\leq	$y_1(1)$	(21.2)
$+ y_2(3x_1$	-	x_2	+	x_3	$) \leq y_2(3)$	
<hr style="border: none; border-top: 1px solid black;"/>					$y_1 + 3y_2.$	

Compare this to the target function $z = 4x_1 + x_2 + 3x_3$. If this expression is bigger than the target function in each variable, namely

$$\begin{aligned} 4 &\leq y_1 + 3y_2 \\ 1 &\leq 4y_1 - y_2 \\ 3 &\leq y_2, \end{aligned}$$

\min	$y_1 + 3y_2$
s.t.	$y_1 + 3y_2 \geq 4$
	$4y_1 - y_2 \geq 1$
	$y_2 \geq 3$
	$y_1, y_2 \geq 0.$

then, $z = 4x_1 + x_2 + 3x_3 \leq (y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \leq y_1 + 3y_2$, the last step follows by Eq. (21.2).

Thus, if we want the best upper bound on η (the maximal value of z) then we want to solve the LP \hat{L} depicted in Figure 21.4. This is the dual program to L and its optimal solution is an upper bound to the optimal solution for L .

Figure 21.4: The dual LP \hat{L} . The primal LP is depicted in Figure 21.3.

21.3.2. The Dual Problem

Given a linear programming problem (i.e., **primal problem**, seen in Figure 21.5 (a)), its associated **dual linear program** is in Figure 21.5 (b). The standard form of the dual LP is depicted in Figure 21.5 (c). Interestingly, you can just compute the dual LP to the given dual LP. What you get back is the original LP. This is demonstrated in Figure 21.6.

We just proved the following result.

Lemma 21.3.1. *Let L be an LP, and let L' be its dual. Let L'' be the dual to L' . Then L and L'' are the same LP.*

21.3.3. The Weak Duality Theorem

Theorem 21.3.2. *If (x_1, x_2, \dots, x_n) is feasible for the primal LP and (y_1, y_2, \dots, y_m) is feasible for the dual LP, then*

$$\sum_j c_j x_j \leq \sum_i b_i y_i.$$

Namely, all the feasible solutions of the dual bound all the feasible solutions of the primal.

Proof: By substitution from the dual form, and since the two solutions are feasible, we know that

$$\sum_j c_j x_j \leq \sum_j \left(\sum_{i=1}^m y_i a_{ij} \right) x_j \leq \sum_i \left(\sum_j a_{ij} x_j \right) y_i \leq \sum_i b_i y_i. \quad \blacksquare$$

$$\begin{aligned} \max \quad & \sum_{j=1}^n c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^n a_{ij} x_j \leq b_i, \\ & \text{for } i = 1, \dots, m, \\ & x_j \geq 0, \\ & \text{for } j = 1, \dots, n. \end{aligned}$$

(a) *primal program*

$$\begin{aligned} \min \quad & \sum_{i=1}^m b_i y_i \\ \text{s.t.} \quad & \sum_{i=1}^m a_{ij} y_i \geq c_j, \\ & \text{for } j = 1, \dots, n, \\ & y_i \geq 0, \\ & \text{for } i = 1, \dots, m. \end{aligned}$$

(b) *dual program*

$$\begin{aligned} \max \quad & \sum_{i=1}^m (-b_i) y_i \\ \text{s.t.} \quad & \sum_{i=1}^m (-a_{ij}) y_i \leq -c_j, \\ & \text{for } j = 1, \dots, n, \\ & y_i \geq 0, \\ & \text{for } i = 1, \dots, m. \end{aligned}$$

(c) dual program in standard form

Figure 21.5: Dual linear programs.

$$\begin{aligned} \max \quad & \sum_{i=1}^m (-b_i) y_i \\ \text{s.t.} \quad & \sum_{i=1}^m (-a_{ij}) y_i \leq -c_j, \\ & \text{for } j = 1, \dots, n, \\ & y_i \geq 0, \\ & \text{for } i = 1, \dots, m. \end{aligned}$$

(a) dual program

$$\begin{aligned} \min \quad & \sum_{j=1}^n -c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^n (-a_{ij}) x_j \geq -b_i, \\ & \text{for } i = 1, \dots, m, \\ & x_j \geq 0, \\ & \text{for } j = 1, \dots, n. \end{aligned}$$

(b) the dual program to the dual program

$$\begin{aligned} \max \quad & \sum_{j=1}^n c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^n a_{ij} x_j \leq b_i, \\ & \text{for } i = 1, \dots, m, \\ & x_j \geq 0, \\ & \text{for } j = 1, \dots, n. \end{aligned}$$

(c) ... which is the original LP.

Figure 21.6: The dual to the dual linear program. Computing the dual of (a) can be done mechanically by following Figure 21.5 (a) and (b). Note, that (c) is just a rewriting of (b).

Interestingly, if we apply the weak duality theorem on the dual program (namely, Figure 21.6 (a) and (b)), we get the inequality $\sum_{i=1}^m (-b_i) y_i \leq \sum_{j=1}^n -c_j x_j$, which is the original inequality in the weak duality theorem. Thus, the weak duality theorem does not imply the strong duality theorem which will be discussed next.

21.4. The strong duality theorem

The *strong duality theorem* states the following.

Theorem 21.4.1. *If the primal LP problem has an optimal solution $x^* = (x_1^*, \dots, x_n^*)$ then the dual also has an optimal solution, $y^* = (y_1^*, \dots, y_m^*)$, such that*

$$\sum_j c_j x_j^* = \sum_i b_i y_i^*.$$

Its proof is somewhat tedious and not very insightful, the basic idea to prove this theorem is to run the simplex algorithm simultaneously on both the primal and the dual LP making steps in sync. When the two stop, they must be equal if they are feasible. We omit the tedious proof.

21.5. Some duality examples

21.5.1. Shortest path

You are given a graph $G = (V, E)$, with source s and target t . We have weights $\omega(u, v)$ on each edge $(u, v) \in E$, and we are interested in the shortest path in this graph from s to t . To simplify the exposition assume that there are no incoming edges in s and no edges leave t . To this end, let d_x be a variable that is the distance between s and x , for any $x \in V$. Clearly, we must have for any edge $(u, v) \in E$, that $d_u + \omega(u, v) \geq d_v$. We also know that $d_s = 0$. Clearly, a trivial solution to this constraints is to set all the variables to zero. So, we are trying to find the assignment that maximizes d_t , such that all the constraints are filled. As such, the LP for computing the shortest path from s to t is the following LP.

$$\begin{array}{ll} \max & d_t \\ \text{s.t.} & d_s \leq 0 \\ & d_u + \omega(u, v) \geq d_v & \forall (u, v) \in E, \\ & d_x \geq 0 & \forall x \in V. \end{array}$$

Equivalently, we get

$$\begin{array}{ll} \max & d_t \\ \text{s.t.} & d_s \leq 0 \\ & d_v - d_u \leq \omega(u, v) & \forall (u, v) \in E, \\ & d_x \geq 0 & \forall x \in V. \end{array}$$

Let us compute the dual. To this end, let y_{uv} be the dual variable for the edge (u, v) , and let y_s be the dual variable for the $d_s \leq 0$ inequality. We get the following dual LP.

$$\begin{array}{ll} \min & \sum_{(u,v) \in E} y_{uv} \omega(u, v) \\ \text{s.t.} & y_s - \sum_{(s,u) \in E} y_{su} \geq 0 & (*) \\ & \sum_{(u,x) \in E} y_{ux} - \sum_{(x,v) \in E} y_{xv} \geq 0 & \forall x \in V \setminus \{s, t\} & (**) \\ & \sum_{(u,t) \in E} y_{ut} \geq 1 & (***) \\ & y_{uv} \geq 0 & \forall (u, v) \in E, \\ & y_s \geq 0. \end{array}$$

Look carefully at this LP. The trick is to think about the y_{uv} as a flow on the edge y_{uv} . (Also, we assume here that the weights are positive.) Then, this LP is the min cost flow of sending one unit of flow from the source s to t . Indeed, if the weights are positive, then $(**)$ can be assumed to hold with equality in the optimal solution, and this is conservation of flow. Equation $(***)$ implies that one unit of flow arrives to the sink t . Finally, $(*)$ implies that at least y_s units of flow leaves the source. The remaining of the LP implies that $y_s \geq 1$. Of course, this min-cost flow version, is without capacities on the edges.

21.5.2. Set Cover and Packing

Consider an instance of **Set Cover** with (S, \mathcal{F}) , where $S = \{u_1, \dots, u_n\}$ and $\mathcal{F} = \{F_1, \dots, F_m\}$, where $F_i \subseteq S$. The natural LP to solve this problem is

$$\begin{aligned} \min \quad & \sum_{F_j \in \mathcal{F}} x_j \\ \text{s.t.} \quad & \sum_{\substack{F_j \in \mathcal{F}, \\ u_i \in F_j}} x_j \geq 1 & \forall u_i \in S, \\ & x_j \geq 0 & \forall F_j \in \mathcal{F}. \end{aligned}$$

The dual LP is

$$\begin{aligned} \max \quad & \sum_{u_i \in S} y_i \\ \text{s.t.} \quad & \sum_{u_i \in F_j} y_i \leq 1 & \forall F_j \in \mathcal{F}, \\ & y_i \geq 0 & \forall u_i \in S. \end{aligned}$$

This is a *packing* LP. We are trying to pick as many vertices as possible, such that no set has more than one vertex we pick. If the sets in \mathcal{F} are pairs (i.e., the set system is a graph), then the problem is known as *edge cover*, and the dual problem is the familiar *independent set* problem. Of course, these are all the fractional versions – getting an integral solution for these problems is completely non-trivial, and in all these cases is impossible in polynomial time since the problems are **NP-COMPLETE**.

As an exercise, write the LP for **Set Cover** for the case where every set has a price associated with it, and you are trying to minimize the total cost of the cover.

21.5.3. Network flow

(We do the following in excruciating details – hopefully its make the presentation clearer.)

Let assume we are given an instance of network flow G , with source s , and sink t . As usual, let us assume there are no incoming edges into the source, no outgoing edges from the sink, and the two are not connected by an edge. The LP for this network flow is the following.

$$\begin{aligned} \max \quad & \sum_{(s,v) \in E} x_{s \rightarrow v} \\ & x_{u \rightarrow v} \leq c(u \rightarrow v) & \forall (u,v) \in E \\ & \sum_{(u,v) \in E} x_{u \rightarrow v} - \sum_{(v,w) \in E} x_{v \rightarrow w} \leq 0 & \forall v \in V \setminus \{s,t\} \\ & - \sum_{(u,v) \in E} x_{u \rightarrow v} + \sum_{(v,w) \in E} x_{v \rightarrow w} \leq 0 & \forall v \in V \setminus \{s,t\} \\ & 0 \leq x_{u \rightarrow v} & \forall (u,v) \in E. \end{aligned}$$

To perform the duality transform, we define a dual variable for each inequality. We get the following dual LP:

$$\begin{array}{ll}
\max & \sum_{(s,v) \in E} x_{s \rightarrow v} \\
& x_{u \rightarrow v} \leq c(u \rightarrow v) & * \quad y_{u \rightarrow v} \quad \forall (u,v) \in E \\
& \sum_{(u,v) \in E} x_{u \rightarrow v} - \sum_{(v,w) \in E} x_{v \rightarrow w} \leq 0 & * \quad y_v \quad \forall v \in V \setminus \{s,t\} \\
& - \sum_{(u,v) \in E} x_{u \rightarrow v} + \sum_{(v,w) \in E} x_{v \rightarrow w} \leq 0 & * \quad y'_v \quad \forall v \in V \setminus \{s,t\} \\
& 0 \leq x_{u \rightarrow v} & \forall (u,v) \in E.
\end{array}$$

Now, we generate the inequalities on the coefficients of the variables of the target functions. We need to carefully account for the edges, and we observe that there are three kinds of edges: source edges, regular edges, and sink edges. Doing the duality transformation carefully, we get the following:

$$\begin{array}{ll}
\min & \sum_{(u,v) \in E} c(u \rightarrow v) y_{u \rightarrow v} \\
& 1 \leq y_{s \rightarrow v} + y_v - y'_v & \forall (s,v) \in E \\
& 0 \leq y_{u \rightarrow v} + y_v - y'_v - y_u + y'_u & \forall (u,v) \in E(G \setminus \{s,t\}) \\
& 0 \leq y_{v \rightarrow t} - y_v + y'_v & \forall (v,t) \in E \\
& y_{u \rightarrow v} \geq 0 & \forall (u,v) \in E \\
& y_v \geq 0 & \forall v \in V \\
& y'_v \geq 0 & \forall v \in V
\end{array}$$

To understand what is going on, let us rewrite the LP, introducing the variable $d_v = y_v - y'_v$, for each $v \in V$ ^①. We get the following modified LP:

$$\begin{array}{ll}
\min & \sum_{(u,v) \in E} c(u \rightarrow v) y_{u \rightarrow v} \\
& 1 \leq y_{s \rightarrow v} + d_v & \forall (s,v) \in E \\
& 0 \leq y_{u \rightarrow v} + d_v - d_u & \forall (u,v) \in E(G \setminus \{s,t\}) \\
& 0 \leq y_{v \rightarrow t} - d_v & \forall (v,t) \in E \\
& y_{u \rightarrow v} \geq 0 & \forall (u,v) \in E
\end{array}$$

Adding the two variables for t and s, and setting their values as follows $d_t = 0$ and $d_s = 1$, we get the following LP:

$$\begin{array}{ll}
\min & \sum_{(u,v) \in E} c(u \rightarrow v) y_{u \rightarrow v} \\
& 0 \leq y_{s \rightarrow v} + d_v - d_s & \forall (s,v) \in E \\
& 0 \leq y_{u \rightarrow v} + d_v - d_u & \forall (u,v) \in E(G \setminus \{s,t\}) \\
& 0 \leq y_{v \rightarrow t} + d_t - d_v & \forall (v,t) \in E \\
& y_{u \rightarrow v} \geq 0 & \forall (u,v) \in E \\
& d_s = 1, \quad d_t = 0
\end{array}$$

^①We could have done this directly, treating the two inequalities as equality, and multiplying it by a single variable that can be both positive and negative – however, it is useful to see why this is correct at least once.

Which simplifies to the following LP:

$$\begin{aligned}
\min \quad & \sum_{(u,v) \in E} c(u \rightarrow v) y_{u \rightarrow v} \\
& d_u - d_v \leq y_{u \rightarrow v} && \forall (u,v) \in E \\
& y_{u \rightarrow v} \geq 0 && \forall (u,v) \in E \\
& d_s = 1, \quad d_t = 0.
\end{aligned}$$

The above LP can be interpreted as follows: We are assigning weights to the edges (i.e., $y_{(u,v)}$). Given such an assignment, it is easy to verify that setting d_u (for all u) to be the shortest path distance under this weighting to the sink t , complies with all inequalities, the assignment $d_s = 1$ implies that we require that the shortest path distance from the source to the sink has length exactly one.

We are next going to argue that the optimal solution to this LP is a min-cut. Lets us first start with the other direction, given a cut (S, T) with $s \in S$ and $t \in T$, observe that setting

$$\begin{aligned}
d_u &= 1 && \forall u \in S \\
d_u &= 0 && \forall u \in T \\
y_{u \rightarrow v} &= 1 && \forall (u,v) \in (S, T) \\
y_{u \rightarrow v} &= 0 && \forall (u,v) \in E \setminus (S, T)
\end{aligned}$$

is a valid solution for the LP.

As for the other direction, consider the optimal solution for the LP, and let its target function value be

$$\alpha^* = \sum_{(u,v) \in E} c(u \rightarrow v) y_{u \rightarrow v}^*$$

(we use $(*)$ notation to denote the values of the variables in the optimal LP solution). Consider generating a cut as follows, we pick a random value uniformly in $z \in [0, 1]$, and we set $S = \{u \mid d_u^* \geq z\}$ and $T = \{u \mid d_u^* < z\}$. This is a valid cut, as $s \in S$ (as $d_s^* = 1$) and $t \in T$ (as $d_t^* = 0$). Furthermore, an edge (u, v) is in the cut, only if $d_u^* > d_v^*$ (otherwise, it is not possible to cut this edge using this approach).

In particular, the probability of $u \in S$ and $v \in T$, is exactly $d_u^* - d_v^*$! Indeed, it is the probability that z falls inside the interval $[d_v^*, d_u^*]$. As such, (u, v) is in the cut with probability $d_u^* - d_v^*$ (again, only if $d_u^* > d_v^*$), which is bounded by $y_{(u,v)}^*$ (by the inequality $d_u - d_v \leq y_{u \rightarrow v}$ in the LP).

So, let $X_{u \rightarrow v}$ be an indicator variable which is one if the edge is in the generated cu. We just argued that $\mathbb{E}[X_{u \rightarrow v}] = \mathbb{P}[X_{u \rightarrow v} = 1] \leq y_{(u,v)}^*$. We thus have that the expected cost of this random cut is

$$\mathbb{E} \left[\sum_{(u,v) \in E} X_{u \rightarrow v} c(u \rightarrow v) \right] = \sum_{(u,v) \in E} c(u \rightarrow v) \mathbb{E}[X_{u \rightarrow v}] \leq \sum_{(u,v) \in E} c(u \rightarrow v) y_{u \rightarrow v}^* = \alpha^*.$$

That is, the expected cost of a random cut here is at most the value of the LP optimal solution. In particular, there must be a cut that has cost at most α^* , see [Remark 21.5.2](#) below. However, we argued that α^* is no larger than the cost of any cut. We conclude that α^* is the cost of the min cut.

We are now ready for the kill, the optimal value of the original max-flow LP; that is, the max-flow (which is a finite number because all the capacities are bounded numbers), is equal by the strong duality theorem, to the optimal value of the dual LP (i.e., α^*). We just argued that α^* is the cost of the min cut in the given network. As such, we proved the following.

Lemma 21.5.1. *The Min-Cut Max-Flow Theorem follows from the strong duality Theorem for Linear Programming.*

Remark 21.5.2. In the above, we used the following “trivial” but powerful argument. Assume you have a random variable Z , and consider its expectation $\mu = \mathbb{E}[Z]$. The expectation μ is the weighted average value of the values the random variable Z might have, and in particular, there must be a value z that might be assigned to Z (with non-zero probability), such that $z \leq \mu$. Putting it differently, the weighted average of a set of numbers is bigger (formally, no smaller) than some number in this set.

This argument is one of the standard tools in the *probabilistic method* – a technique to prove the existence of entities by considering expectations and probabilities.

21.6. Solving LPs without ever getting into a loop - symbolic perturbations

21.6.1. The problem and the basic idea

Consider the following LP:

$$\begin{aligned} \max \quad & z = v + \sum_{j \in N} c_j x_j, \\ \text{s.t.} \quad & x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i = 1, \dots, n, \\ & x_i \geq 0, \quad \forall i = 1, \dots, n + m. \end{aligned}$$

(Here $B = \{1, \dots, n\}$ and $N = \{n + 1, \dots, n + m\}$.) The **Simplex** algorithm might get stuck in a loop of pivoting steps, if one of the constants b_i becomes zero during the algorithm execution. To avoid this, we are going to add tiny infinitesimals to all the equations. Specifically, let $\varepsilon > 0$ be an arbitrarily small constant, and let $\varepsilon_i = \varepsilon^i$. The quantities $\varepsilon_1, \dots, \varepsilon_n$ are infinitesimals of different scales. We slightly perturb the above LP by adding them to each equation. We get the following modified LP:

$$\begin{aligned} \max \quad & z = v + \sum_{j \in N} c_j x_j, \\ \text{s.t.} \quad & x_i = \varepsilon_i + b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i = 1, \dots, n, \\ & x_i \geq 0, \quad \forall i = 1, \dots, n + m. \end{aligned}$$

Importantly, any feasible solution to the original LP translates into a valid solution of this LP (we made things better by adding these symbolic constants).

The rule of the game is now that we treat $\varepsilon_1, \dots, \varepsilon_n$ as symbolic constants. Of course, when we do pivoting, we need to be able to compare two numbers and decide which one is bigger. Formally, given two numbers

$$\alpha = \alpha_0 + \alpha_1 \varepsilon_1 + \dots + \alpha_n \varepsilon_n \quad \text{and} \quad \beta = \beta_0 + \beta_1 \varepsilon_1 + \dots + \beta_n \varepsilon_n, \quad (21.3)$$

then $\alpha > \beta$ if and only if there is an index i such that $\alpha_0 = \beta_0, \alpha_1 = \beta_1, \dots, \alpha_{i-1} = \beta_{i-1}$ and $\alpha_i > \beta_i$. That is, $\alpha > \beta$ if the vector $(\alpha_0, \alpha_1, \dots, \alpha_n)$ is *lexicographically* larger than $(\beta_0, \beta_1, \dots, \beta_n)$.

Significantly, but not obviously at this stage, the simplex algorithm would never divide an ε_i by an ε_j , so we are good to go – we can perform all the needed arithmetic operations of the **Simplex** using these symbolic constants, and we claim that now the constant term (which is a number of the form of Eq. (21.3)) is now never zero. This implies immediately that the **Simplex** algorithm always makes progress, and it does terminate. We still need to address the two issues:

- (A) How are the symbolic perturbations updated at each iteration?
- (B) Why the constants can never be zero?

21.6.2. Pivoting as a Gauss elimination step

Consider the LP equations

$$x_i + \sum_{j \in N} a_{ij}x_j = b_i, \quad \text{for } i \in B,$$

where $B = \{1, \dots, n\}$ and $N = \{n+1, \dots, n+m\}$. We can write these equations down in matrix form

x_1	x_2	\dots	x_n	x_{n+1}	x_{n+2}	\dots	x_j	\dots	x_{n+m}	const		
1	0	\dots	0	$a_{1,n+1}$	$a_{1,n+2}$	\dots	$a_{1,j}$	\dots	$a_{1,n+m}$	b_1		
0	1	\dots	0	$a_{2,n+1}$	$a_{2,n+2}$	\dots	$a_{2,j}$	\dots	$a_{2,n+m}$	b_2		
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots		
0	0	\dots 0	1	0	\dots 0	$a_{k,n+1}$	$a_{k,n+2}$	\dots	$a_{k,j}$	\vdots	$a_{k,n+m}$	b_k
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
0	\dots	0	1	$a_{n,n+1}$	$a_{n,n+2}$	\dots	$a_{n,j}$	\dots	$a_{n,n+m}$	b_n		

Assume that now we do a pivoting step with x_j entering the basic variables, and x_k leaving. To this end, let us multiply the k th row (i.e., the k th equation) by $1/a_{k,j}$, this result in the k th row having 1 instead of $a_{k,j}$. Let this resulting row be denoted by \mathbf{r} . Now, add $a_{i,j}\mathbf{r}$ to the i th row of the matrix, for all i . Clearly, in the resulting row/equation, the coefficient of x_j is going to be zero, in all rows except the k th one, where it is 1. Note, that on the matrix on the left side, all the columns are the same, except for the k th column, which might now have various numbers in this column. The final step is to exchange the k th column on the left, with the j th column on the right. And that is one pivoting step, when working on the LP using a matrix. It is very similar to one step of the Gauss elimination in matrices, if you are familiar with that.

21.6.2.1. Back to the perturbation scheme

We now add a new matrix to the above representations on the right side, that keeps track of the ε s. This looks initially as follows.

x_1	x_2	\dots	x_n	x_{n+1}	\dots	x_j	\dots	x_{n+m}	const	ε_1	ε_2	\dots	ε_n
1	0	\dots	0	$a_{1,n+1}$	\dots	$a_{1,j}$	\dots	$a_{1,n+m}$	b_1	1	0	\dots	0
0	1	\dots	0	$a_{2,n+1}$	\dots	$a_{2,j}$	\dots	$a_{2,n+m}$	b_2	0	1	\dots	0
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
0	0	\dots 0	1	0	\dots 0	$a_{k,n+1}$	\dots	$a_{k,j}$	\vdots	$a_{k,n+m}$	\vdots	\vdots	\vdots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
0	\dots	0	1	$a_{n,n+1}$	\dots	$a_{n,j}$	\dots	$a_{n,n+m}$	b_n	0	0	\dots	1

Now, we run the algorithm as described above, using the ε s to resolve which variables are entering and leaving. The critical observation is that throughout the algorithm execution we are adding rows, and multiplying them by non-zero constants. The matrix on the right has initially full rank, and throughout the execution of the algorithm its rank remains the same (because the linear operation we do on the rows can not change the rank of the matrix). In particular, it is impossible that a row on the right side of the matrix is all zero, or equal to another row, or equal to another row if multiplied by a constant. Namely, the symbolic constant encoded by the ε s as we run the **Simplex** algorithm can never be zero. And furthermore, these constants are never equal for two different equations. We conclude that the **Simplex** algorithm now always make progress in each pivoting step.

21.6.2.2. The overall algorithm

We run the **Simplex** algorithm with the above described symbolic perturbation. The final stroke is that each basic variable x_i in the computed solution now equal to a number of the form $x_i = \alpha_0 + \sum_i \alpha_i \varepsilon_i$. We interpret this as $x_i = \alpha_0$, by setting all the ε s to be zero.

Chapter 22

Approximation Algorithms using Linear Programming

22.1. Weighted vertex cover

Consider the **Weighted Vertex Cover** problem. Here, we have a graph $G = (V, E)$, and each vertex $v \in V$ has an associated cost c_v . We would like to compute a vertex cover of minimum cost – a subset of the vertices of G with minimum total cost so that each edge has at least one of its endpoints in the cover. This problem is (of course) **NP-HARD**, since the decision problem where all the weights are 1, is the **Vertex Cover** problem, which we had shown to be **NPC**.

Let us first state this optimization problem is an integer programming. Indeed, for any $v \in V$, let define a variable x_v which is 1 if we decide to pick v to the vertex cover, and zero otherwise. The restriction that x_v is either 0 or 1, is written formally as $x_v \in \{0, 1\}$. Next, its required that every edge $vu \in E$ is covered. Namely, we require that $x_v \vee x_u$ to be **TRUE**. For reasons that would be come clearer shortly, we prefer to write this condition as a linear inequality; namely, we require that $x_v + x_u \geq 1$. Finally, minimize the total cost of the vertices picked for the cover – namely, minimize $\sum_{v \in V} x_v c_v$. Putting it together, we get the following integer programming instance:

$$\begin{array}{ll} \min & \sum_{v \in V} c_v x_v \\ \text{such that} & x_v \in \{0, 1\} \quad \forall v \in V \\ & x_v + x_u \geq 1 \quad \forall vu \in E. \end{array} \quad (22.1)$$

Naturally, solving this integer programming efficiently is **NP-HARD**, so instead let us try to relax this optimization problem to be a **LP** (which we can solve efficiently, at least in practice^①). To do this, we need to relax the integer program. We will do it by allowing the variables x_v to get real values between 0 and 1. This is done by replacing the condition that $x_v \in \{0, 1\}$ by the constraint $0 \leq x_v \leq 1$. The resulting **LP** is

$$\begin{array}{ll} \min & \sum_{v \in V} c_v x_v \\ \text{such that} & 0 \leq x_v \quad \forall v \in V, \\ & x_v \leq 1 \quad \forall v \in V, \\ & x_v + x_u \geq 1 \quad \forall vu \in E. \end{array} \quad (22.2)$$

^①And also in theory if the costs are integers, using more advanced algorithms than the **Simplex** algorithm.

So, consider the optimal solution to this LP, assigning value \hat{x}_v to the variable X_v , for all $v \in V$. As such, the optimal value of the LP solution is

$$\hat{\alpha} = \sum_{v \in V} c_v \hat{x}_v.$$

Similarly, let the optimal integer solution to integer program (IP) Eq. (22.1) denoted by x_v^I , for all $v \in V$ and α^I , respectively. Note, that any feasible solution for the IP of Eq. (22.1), is a feasible solution for the LP of Eq. (22.2). As such, we must have that

$$\hat{\alpha} \leq \alpha^I,$$

where α^I is the value of the optimal solution.

So, what happened? We solved the relaxed optimization problem, and got a fractional solution (i.e., values of \hat{x}_v can be fractions). On the other hand, the cost of this fractional solution is better than the optimal cost. So, the natural question is how to turn this fractional solution into a (valid!) integer solution. This process is known as *rounding*.

To this end, it is beneficial to consider a vertex v and its fractional value \hat{x}_v . If $\hat{x}_v = 1$ then we definitely want to put it into our solution. If $\hat{x}_v = 0$ then the LP consider this vertex to be useless, and we really do not want to use it. Similarly, if $\hat{x}_v = 0.9$, then the LP considers this vertex to be very useful (0.9 useful to be precise, whatever this “means”). Intuitively, since the LP puts its money where its belief is (i.e., $\hat{\alpha}$ value is a function of this “belief” generated by the LP), we should trust the LP values as a guidance to which vertices are useful and which are not. Which brings to forefront the following idea: Lets pick all the vertices that are above a certain threshold of usefulness according to the LP solution. Formally, let

$$S = \{v \mid \hat{x}_v \geq 1/2\}.$$

We claim that S is a valid vertex cover, and its cost is low.

Indeed, let us verify that the solution is valid. We know that for any edge vu , it holds

$$\hat{x}_v + \hat{x}_u \geq 1.$$

Since $0 \leq \hat{x}_v \leq 1$ and $0 \leq \hat{x}_u \leq 1$, it must be either $\hat{x}_v \geq 1/2$ or $\hat{x}_u \geq 1/2$. Namely, either $v \in S$ or $u \in S$, or both of them are in S , implying that indeed S covers all the edges of G .

As for the cost of S , we have

$$c_S = \sum_{v \in S} c_v = \sum_{v \in S} 1 \cdot c_v \leq \sum_{v \in S} 2\hat{x}_v \cdot c_v \leq 2 \sum_{v \in V} \hat{x}_v c_v = 2\hat{\alpha} \leq 2\alpha^I,$$

since $\hat{x}_v \geq 1/2$ as $v \in S$.

Since α^I is the cost of the optimal solution, we got the following result.

Theorem 22.1.1. *The **Weighted Vertex Cover** problem can be 2-approximated by solving a single LP. Assuming computing the LP takes polynomial time, the resulting approximation algorithm takes polynomial time.*

What lessons can we take from this example? First, this example might be simple, but the resulting approximation algorithm is non-trivial. In particular, I am not aware of any other 2-approximation algorithm for the weighted problem that does not use LP. Secondly, the *relaxation* of an optimization problem into a LP provides us with a way to get some insight into the problem in hand. It also hints that in interpreting the values returned by the LP, and how to use them to do the rounding, we have to be creative.

22.2. Revisiting **Set Cover**

In this section, we are going to revisit the **Set Cover** problem, and provide an approximation algorithm for this problem. This approximation algorithm would not be better than the greedy algorithm we already saw, but it would expose us to a new technique that we would use shortly for a different problem.

Set Cover

Instance: (S, \mathcal{F})

S - a set of n elements

\mathcal{F} - a family of subsets of S , s.t. $\bigcup_{X \in \mathcal{F}} X = S$.

Question: The set $\mathcal{X} \subseteq \mathcal{F}$ such that \mathcal{X} contains as few sets as possible, and \mathcal{X} covers S .

As before, we will first define an IP for this problem. In the following IP, the second condition just states that any $s \in S$, must be covered by some set.

$$\begin{aligned} \min \quad & \alpha = \sum_{U \in \mathcal{F}} x_U, \\ \text{s.t.} \quad & x_U \in \{0, 1\} && \forall U \in \mathcal{F}, \\ & \sum_{U \in \mathcal{F}, s \in U} x_U \geq 1 && \forall s \in S. \end{aligned}$$

Next, we relax this IP into the following LP.

$$\begin{aligned} \min \quad & \alpha = \sum_{U \in \mathcal{F}} x_U, \\ & 0 \leq x_U \leq 1 && \forall U \in \mathcal{F}, \\ & \sum_{U \in \mathcal{F}, s \in U} x_U \geq 1 && \forall s \in S. \end{aligned}$$

As before, consider the optimal solution to the LP: $\forall U \in \mathcal{F}$, \widehat{x}_U , and $\widehat{\alpha}$. Similarly, let the optimal solution to the IP (and thus for the problem) be: $\forall U \in \mathcal{F}$, x_U^I , and α^I . As before, we would try to use the LP solution to guide us in the rounding process. As before, if \widehat{x}_U is close to 1 then we should pick U to the cover and if \widehat{x}_U is close to 0 we should not. As such, its natural to pick $U \in \mathcal{F}$ into the cover by randomly choosing it into the cover with **probability** \widehat{x}_U . Consider the resulting family of sets \mathcal{G} . Let Z_S be an indicator variable which is one if $S \in \mathcal{G}$. We have that the cost of \mathcal{G} is $\sum_{S \in \mathcal{F}} Z_S$, and the expected cost is

$$\mathbb{E}[\text{cost of } \mathcal{G}] = \mathbb{E}\left[\sum_{S \in \mathcal{F}} Z_S\right] = \sum_{S \in \mathcal{F}} \mathbb{E}[Z_S] = \sum_{S \in \mathcal{F}} \mathbb{P}[S \in \mathcal{G}] = \sum_{S \in \mathcal{F}} \widehat{x}_S = \widehat{\alpha} \leq \alpha^I. \quad (22.3)$$

As such, in expectation, \mathcal{G} is not too expensive. The problem, of course, is that \mathcal{G} might fail to cover some element $s \in S$. To this end, we repeat this algorithm

$$m = 10 \lceil \lg n \rceil = O(\log n)$$

times, where $n = |S|$. Let \mathcal{G}_i be the random cover computed in the i th iteration, and let $\mathcal{H} = \cup_i \mathcal{G}_i$. We return \mathcal{H} as the required cover.

The solution \mathcal{H} covers S . For an element $s \in S$, we have that

$$\sum_{U \in \mathcal{F}, s \in U} \widehat{x}_U \geq 1, \quad (22.4)$$

and consider the probability that s is not covered by \mathcal{G}_i , where \mathcal{G}_i is the family computed in the i th iteration of the algorithm. Since deciding if the include each set U into \mathcal{G}_i is done independently for each set, we have that the probability that s is not covered is

$$\begin{aligned} \mathbb{P}[s \text{ not covered by } \mathcal{G}_i] &= \mathbb{P}[\text{none of } U \in \mathcal{F}, \text{ such that } s \in U \text{ were picked into } \mathcal{G}_i] \\ &= \prod_{U \in \mathcal{F}, s \in U} \mathbb{P}[U \text{ was not picked into } \mathcal{G}_i] \\ &= \prod_{U \in \mathcal{F}, s \in U} (1 - \widehat{x}_U) \\ &\leq \prod_{U \in \mathcal{F}, s \in U} \exp(-\widehat{x}_U) = \exp\left(-\sum_{U \in \mathcal{F}, s \in U} \widehat{x}_U\right) \\ &\leq \exp(-1) \leq \frac{1}{2}, \end{aligned}$$

by Eq. (22.4). As such, the probability that s is not covered in all m iterations is at most

$$\left(\frac{1}{2}\right)^m < \frac{1}{n^{10}},$$

since $m = O(\log n)$. In particular, the probability that one of the n elements of S is not covered by \mathcal{H} is at most $n(1/n^{10}) = 1/n^9$.

Cost. By Eq. (22.3), in each iteration the expected cost of the cover computed is at most the cost of the optimal solution (i.e., α^I). As such the expected cost of the solution computed is

$$c_{\mathcal{H}} \leq \sum_i c_{B_i} \leq m\alpha^I = O(\alpha^I \log n).$$

. Putting everything together, we get the following result.

Theorem 22.2.1. *By solving an LP one can get an $O(\log n)$ -approximation to set cover by a randomized algorithm. The algorithm succeeds with high probability.*

22.3. Minimizing congestion

Let G be a graph with n vertices, and let π_i and σ_i be two paths with the same endpoints $v_i, u_i \in V(G)$, for $i = 1, \dots, t$. Imagine that we need to send one unit of flow from v_i to u_i , and we need to choose whether to use the path π_i or σ_i . We would like to do it in such a way that no edge in the graph is being used too much.

Definition 22.3.1. Given a set X of paths in a graph G , the **congestion** of X is the maximum number of paths in X that use the same edge.

Consider the following linear program:

$$\begin{aligned} \min \quad & w \\ \text{s.t.} \quad & x_i \geq 0 && i = 1, \dots, t, \\ & x_i \leq 1 && i = 1, \dots, t, \\ & \sum_{e \in \pi_i} x_i + \sum_{e \in \sigma_i} (1 - x_i) \leq w && \forall e \in E. \end{aligned}$$

Let \hat{x}_i be the value of x_i in the optimal solution of this LP, and let \hat{w} be the value of w in this solution. Clearly, the optimal congestion must be bigger than \hat{w} .

Let X_i be a random variable which is one with probability \hat{x}_i , and zero otherwise. If $X_i = 1$ then we use π to route from v_i to u_i , otherwise we use σ_i . Clearly, the congestion of e is

$$Y_e = \sum_{e \in \pi_i} X_i + \sum_{e \in \sigma_i} (1 - X_i).$$

And in expectation

$$\begin{aligned} \alpha_e &= \mathbb{E}[Y_e] = \mathbb{E} \left[\sum_{e \in \pi_i} X_i + \sum_{e \in \sigma_i} (1 - X_i) \right] = \sum_{e \in \pi_i} \mathbb{E}[X_i] + \sum_{e \in \sigma_i} \mathbb{E}[(1 - X_i)] \\ &= \sum_{e \in \pi_i} \hat{x}_i + \sum_{e \in \sigma_i} (1 - \hat{x}_i) \leq \hat{w}. \end{aligned}$$

Using the Chernoff inequality, we have that

$$\mathbb{P}[Y_e \geq (1 + \delta)\alpha_e] \leq \exp\left(-\frac{\alpha_e \delta^2}{4}\right) \leq \exp\left(-\frac{\hat{w} \delta^2}{4}\right).$$

(Note, that this works only if $\delta < 2e - 1$, see [Theorem 10.2.7](#)). Let $\delta = \sqrt{\frac{400}{\hat{w}} \ln t}$. We have that

$$\mathbb{P}[Y_e \geq (1 + \delta)\alpha_e] \leq \exp\left(-\frac{\delta^2 \hat{w}}{4}\right) \leq \frac{1}{t^{100}},$$

which is very small. In particular, if $t \geq n^{1/50}$ then all the edges in the graph do not have congestion larger than $(1 + \delta)\hat{w}$.

To see what this result means, let us play with the numbers. Let assume that $t = n$, and $\hat{w} \geq \sqrt{n}$. Then, the solution has congestion larger than the optimal solution by a factor of

$$1 + \delta = 1 + \sqrt{\frac{20}{\hat{w}} \ln t} \leq 1 + \frac{\sqrt{20 \ln n}}{n^{1/4}},$$

which is of course extremely close to 1, if n is sufficiently large.

Theorem 22.3.2. *Given a graph with n vertices, and t pairs of vertices, such that for every pair (s_i, t_i) there are two possible paths to connect s_i to t_i . Then one can choose for each pair which path to use, such that the most congested edge, would have at most $(1 + \delta)\text{opt}$, where opt is the congestion of the optimal solution, and $\delta = \sqrt{\frac{20}{\hat{w}} \ln t}$.*

When the congestion is low. Assume that \hat{w} is a constant. In this case, we can get a better bound by using the Chernoff inequality in its more general form, see [Theorem 10.2.7](#). Indeed, set $\delta = c \ln t / \ln \ln t$, where c is a constant. For $\mu = \alpha_e$, we have that

$$\begin{aligned} \mathbb{P}[Y_e \geq (1 + \delta)\mu] &\leq \left(\frac{e^\delta}{(1 + \delta)^{1 + \delta}} \right)^\mu = \exp\left(\mu(\delta - (1 + \delta) \ln(1 + \delta))\right) = \exp\left(-\mu c' \ln t\right) \\ &\leq \frac{1}{t^{O(1)}}, \end{aligned}$$

where c' is a constant that depends on c and grows if c grows. We thus proved that if the optimal congestion is $O(1)$, then the algorithm outputs a solution with congestion $O(\log t / \log \log t)$, and this holds with high probability.

Part VII

Miscellaneous topics

Chapter 23

Fast Fourier Transform

“But now, reflecting further, there begins to creep into his breast a touch of fellow-feeling for his imitators. For it seems to him now that there are but a handful of stories in the world; and if the young are to be forbidden to prey upon the old then they must sit for ever in silence.”

– J.M. Coetzee,

23.1. Introduction

In this chapter, we will address the problem of multiplying two polynomials quickly.

Definition 23.1.1. A *polynomial* $p(x)$ of degree n is a function of the form $p(x) = \sum_{j=0}^n a_j x^j = a_0 + x(a_1 + x(a_2 + \dots + x a_n))$.

Note, that given x_0 , the polynomial can be evaluated at x_0 in $O(n)$ time.

There is a “dual” (and equivalent) representation of a polynomial. We sample its value in enough points, and store the values of the polynomial at those points. The following theorem states this formally. We omit the proof as you should have seen it already at some earlier math class.

Theorem 23.1.2. For any set $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$ of n **point-value pairs** such that all the x_k values are distinct, there is a unique polynomial $p(x)$ of degree $n - 1$, such that $y_k = p(x_k)$, for $k = 0, \dots, n - 1$.

An explicit formula for $p(x)$ as a function of those point-value pairs is

$$p(x) = \sum_{i=0}^{n-1} y_i \frac{\prod_{j \neq i} (x - x_j)}{\prod_{j \neq i} (x_i - x_j)}.$$

Note, that the i th term in this summation is zero for $X = x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_{n-1}$, and is equal to y_i for $x = x_i$.

It is easy to verify that given n point-value pairs, we can compute $p(x)$ in $O(n^2)$ time (using the above formula).

The point-value pairs representation has the advantage that we can multiply two polynomials quickly. Indeed, if we have two polynomials p and q of degree $n - 1$, both represented by $2n$ (we are using more points

than we need) point-value pairs

$$\begin{aligned} & \{(x_0, y_0), (x_1, y_1), \dots, (x_{2n-1}, y_{2n-1})\} \text{ for } p(x), \\ & \text{and } \{(x_0, y'_0), (x_1, y'_1), \dots, (x_{2n-1}, y'_{2n-1})\} \text{ for } q(x). \end{aligned}$$

Let $r(x) = p(x)q(x)$ be the product of these two polynomials. Computing $r(x)$ directly requires $O(n^2)$ using the naive algorithm. However, in the point-value representation we have, that the representation of $r(x)$ is

$$\begin{aligned} \{(x_0, r(x_0)), \dots, (x_{2n-1}, r(x_{2n-1}))\} &= \{(x_0, p(x_0)q(x_0)), \dots, (x_{2n-1}, p(x_{2n-1})q(x_{2n-1}))\} \\ &= \{(x_0, y_0 y'_0), \dots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\}. \end{aligned}$$

Namely, once we computed the representation of $p(x)$ and $q(x)$ using point-value pairs, we can multiply the two polynomials in linear time. Furthermore, we can compute the standard representation of $r(x)$ from this representation.

Thus, if could translate quickly (i.e., $O(n \log n)$ time) from the standard representation of a polynomial to point-value pairs representation, and back (to the regular representation) then we could compute the product of two polynomials in $O(n \log n)$ time. The **Fast Fourier Transform** is a method for doing exactly this. It is based on the idea of choosing the x_i values carefully and using divide and conquer.

23.2. Computing a polynomial quickly on n values

In the following, we are going to assume that the polynomial we work on has degree $n-1$, where $n = 2^k$. If this is not true, we can pad the polynomial with terms having zero coefficients.

Assume that we magically were able to find a set of numbers $\Psi = \{x_1, \dots, x_n\}$, so that it has the following property: $|\text{SQ}(\Psi)| = n/2$, where $\text{SQ}(\Psi) = \{x^2 \mid x \in \Psi\}$. Namely, when we square the numbers of Ψ , we remain with only $n/2$ distinct values, although we started with n values. It is quite easy to find such a set.

What is much harder is to find a set that have this property repeatedly. Namely, $\text{SQ}(\text{SQ}(\Psi))$ would have $n/4$ distinct values, $\text{SQ}(\text{SQ}(\text{SQ}(\Psi)))$ would have $n/8$ values, and $\text{SQ}^i(\Psi)$ would have $n/2^i$ distinct values.

Predictably, maybe, it is easy to show that there is no such set of real numbers (verify...). But let us for the time being ignore this technicality, and fly, for a moment, into the land of fantasy, and assume that we do have such a set of numbers, so that $|\text{SQ}^i(\Psi)| = n/2^i$ numbers, for $i = 0, \dots, k$. Let us call such a set of numbers **collapsible**.

Given a set of numbers $\mathcal{X} = \{x_0, \dots, x_n\}$ and a polynomial $p(x)$, let

$$p(\mathcal{X}) = \langle (x_0, p(x_0)), \dots, (x_n, p(x_n)) \rangle.$$

Furthermore, let us rewrite $p(x) = \sum_{i=0}^{n-1} a_i x^i$ as $p(x) = u(x^2) + x \cdot v(x^2)$, where

$$u(y) = \sum_{i=0}^{n/2-1} a_{2i} y^i \quad \text{and} \quad v(y) = \sum_{i=0}^{n/2-1} a_{1+2i} y^i.$$

Namely, we put all the even degree terms of $p(x)$ into $u(\cdot)$, and all the odd degree terms into $v(\cdot)$. The maximum degree of the two polynomials $u(y)$ and $v(y)$ is $n/2$.

We are now ready for the kill: To compute $p(\Psi)$ for Ψ , which is a collapsible set, we have to compute $u(\text{SQ}(\Psi)), v(\text{SQ}(\Psi))$. Namely, once we have the value-point pairs of $u(\text{SQ}(\Psi)), v(\text{SQ}(\Psi))$ we can, in *linear* time, compute $p(\Psi)$. But, $\text{SQ}(\Psi)$ have $n/2$ values because we assumed that Ψ is collapsible. Namely, to compute n point-value pairs of $p(\cdot)$, we have to compute $n/2$ point-value pairs of two polynomials of degree $n/2$ over a set of $n/2$ numbers.

Namely, we reduce a problem of size n into two problems of size $n/2$. The resulting algorithm is depicted in **Figure 23.1**.

```

FFTAlg( $p, X$ )
  input:  $p(x)$ : A polynomial of degree  $n$ :  $p(x) = \sum_{i=0}^{n-1} a_i x^i$ 
            $X$ : A collapsible set of  $n$  elements.
  output:  $p(X)$ 
  begin
     $u(y) = \sum_{i=0}^{n/2-1} a_{2i} y^i$ 
     $v(y) = \sum_{i=0}^{n/2-1} a_{1+2i} y^i$ .
     $Y = \text{SQ}(X) = \{x^2 \mid x \in X\}$ .
     $U = \text{FFTA}lg(u, Y)$            /*  $U = u(Y)$  */
     $V = \text{FFTA}lg(v, Y)$            /*  $V = v(Y)$  */

     $Out \leftarrow \emptyset$ 
    for  $x \in A$  do
      /*  $p(x) = u(x^2) + x \cdot v(x^2)$  */
      /*  $U[x^2]$  is the value  $u(x^2)$  */
       $(x, p(x)) \leftarrow (x, U[x^2] + x \cdot V[x^2])$ 
       $Out \leftarrow Out \cup \{(x, p(x))\}$ 

    return  $Out$ 
  end

```

Figure 23.1: The FFT algorithm.

What is the running time of **FFTA**lg? Well, clearly, all the operations except the recursive calls takes $O(n)$ time (assume, for the time being, that we can fetch $U[x^2]$ in $O(1)$ time). As for the recursion, we call recursively on a polynomial of degree $n/2$ with $n/2$ values (Ψ is collapsible!). Thus, the running time is $T(n) = 2T(n/2) + O(n)$, which is $O(n \log n)$ – exactly what we wanted.

23.2.1. Generating Collapsible Sets

Nice! But how do we resolve this “technicality” of not having collapsible set? It turns out that if we work over the complex numbers (instead of over the real numbers), then generating collapsible sets is quite easy. Describing complex numbers is outside the scope of this writeup, and we assume that you already have encountered them before. Nevertheless a quick reminder is provided in [Section 23.4.1](#). Everything you can do over the real numbers you can do over the complex numbers, and much more (complex numbers are your friend).

In particular, let γ denote a n th root of unity. There are n such roots, and let $\gamma_j(n)$ denote the j th root, see [Figure 23.2_{p161}](#). In particular, let

$$\gamma_j(n) = \cos((2\pi j)/n) + \mathbf{i} \sin((2\pi j)/n) = \gamma^j.$$

Let $\mathcal{A}(n) = \{\gamma_0(n), \dots, \gamma_{n-1}(n)\}$. It is easy to verify that $|\text{SQ}(\mathcal{A}(n))|$ has exactly $n/2$ elements. In fact, $\text{SQ}(\mathcal{A}(n)) = \mathcal{A}(n/2)$, as can be easily verified. Namely, if we pick n to be a power of 2, then $\mathcal{A}(n)$ is the *required* collapsible set.

Theorem 23.2.1. *Given polynomial $p(x)$ of degree n , where n is a power of two, then we can compute $p(X)$ in $O(n \log n)$ time, where $X = \mathcal{A}(n)$ is the set of n different powers of the n th root of unity over the complex numbers.*

We can now multiply two polynomials quickly by transforming them to the point-value pairs representation over the n th root of unity, but we still have to transform this representation back to the regular representation.

23.3. Recovering the polynomial

This part of the writeup is somewhat more technical. Putting it shortly, we are going to apply the **FFTA** algorithm once again to recover the original polynomial. The details follow.

It turns out that we can interpret the FFT as a matrix multiplication operator. Indeed, if we have $p(x) = \sum_{i=0}^{n-1} a_i x^i$ then evaluating $p(\cdot)$ on $\mathcal{A}(n)$ is equivalent to:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & \gamma_0 & \gamma_0^2 & \gamma_0^3 & \cdots & \gamma_0^{n-1} \\ 1 & \gamma_1 & \gamma_1^2 & \gamma_1^3 & \cdots & \gamma_1^{n-1} \\ 1 & \gamma_2 & \gamma_2^2 & \gamma_2^3 & \cdots & \gamma_2^{n-1} \\ 1 & \gamma_3 & \gamma_3^2 & \gamma_3^3 & \cdots & \gamma_3^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \gamma_{n-1} & \gamma_{n-1}^2 & \gamma_{n-1}^3 & \cdots & \gamma_{n-1}^{n-1} \end{pmatrix}}_{\text{the matrix } V} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix},$$

where $\gamma_j = \gamma_j(n) = (\gamma_1(n))^j$ is the j th power of the n th root of unity, and $y_j = p(\gamma_j)$.

This matrix V is very interesting, and is called the **Vandermonde** matrix. Let V^{-1} be the inverse matrix of this Vandermonde matrix. And let multiply the above formula from the left. We get:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = V^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

Namely, we can recover the polynomial $p(x)$ from the point-value pairs

$$\{(\gamma_0, p(\gamma_0)), (\gamma_1, p(\gamma_1)), \dots, (\gamma_{n-1}, p(\gamma_{n-1}))\}$$

by doing a single matrix multiplication of V^{-1} by the vector $[y_0, y_1, \dots, y_{n-1}]$. However, multiplying a vector with n entries with a matrix of size $n \times n$ takes $O(n^2)$ time. Thus, we had not benefited anything so far.

However, since the Vandermonde matrix is so well behaved^①, it is not too hard to figure out the inverse matrix.

Claim 23.3.1.

$$V^{-1} = \frac{1}{n} \begin{pmatrix} 1 & \beta_0 & \beta_0^2 & \beta_0^3 & \cdots & \beta_0^{n-1} \\ 1 & \beta_1 & \beta_1^2 & \beta_1^3 & \cdots & \beta_1^{n-1} \\ 1 & \beta_2 & \beta_2^2 & \beta_2^3 & \cdots & \beta_2^{n-1} \\ 1 & \beta_3 & \beta_3^2 & \beta_3^3 & \cdots & \beta_3^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \beta_{n-1} & \beta_{n-1}^2 & \beta_{n-1}^3 & \cdots & \beta_{n-1}^{n-1} \end{pmatrix},$$

where $\beta_j = (\gamma_j(n))^{-1}$.

Proof: Consider the (u, v) entry in the matrix $C = V^{-1}V$. We have

$$C_{u,v} = \sum_{j=0}^{n-1} \frac{(\beta_u)^j (\gamma_j)^v}{n}.$$

^①Not to mention famous, beautiful and well known – in short a celebrity matrix.

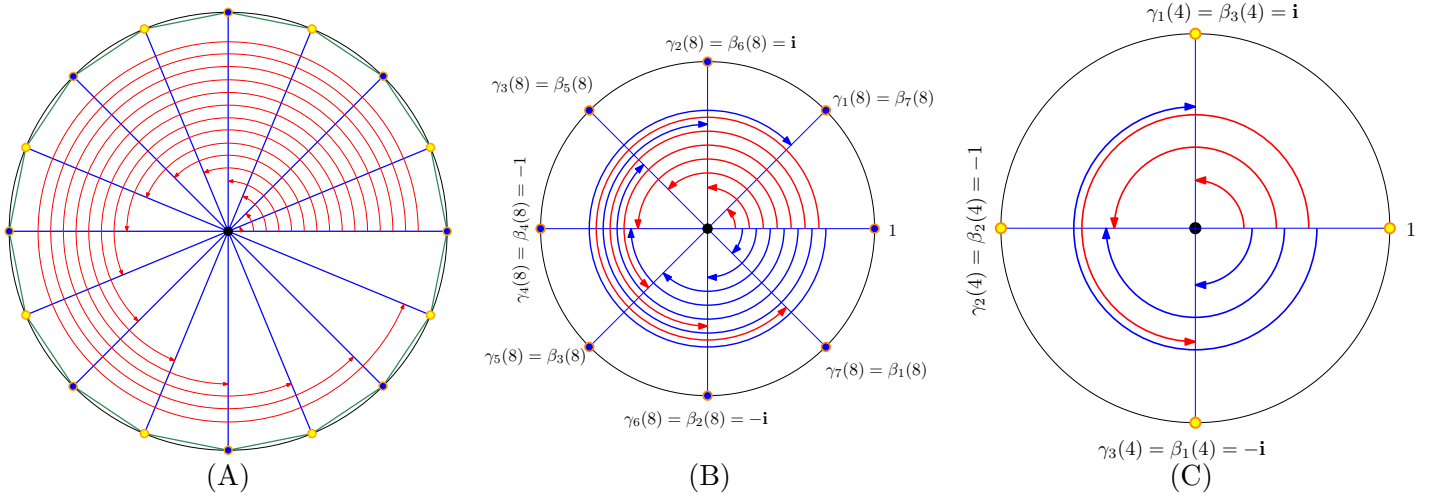


Figure 23.2: (A) The 16 roots of unity. (B) The 8 roots of unity. (C) The 4 roots of unity.

We need to use the fact here that $\gamma_j = (\gamma_1)^j$ as can be easily verified. Thus,

$$C_{u,v} = \sum_{j=0}^{n-1} \frac{(\beta_u)^j ((\gamma_1)^j)^v}{n} = \sum_{j=0}^{n-1} \frac{(\beta_u)^j ((\gamma_1)^v)^j}{n} = \sum_{j=0}^{n-1} \frac{(\beta_u \gamma_v)^j}{n}.$$

Clearly, if $u = v$ then

$$C_{u,u} = \frac{1}{n} \sum_{j=0}^{n-1} (\beta_u \gamma_u)^j = \frac{1}{n} \sum_{j=0}^{n-1} (1)^j = \frac{n}{n} = 1.$$

If $u \neq v$ then,

$$\beta_u \gamma_v = (\gamma_u)^{-1} \gamma_v = (\gamma_1)^{-u} \gamma_1^v = (\gamma_1)^{v-u} = \gamma_{v-u}.$$

And

$$C_{u,v} = \frac{1}{n} \sum_{j=0}^{n-1} (\gamma_{v-u})^j = \frac{1}{n} \cdot \frac{\gamma_{v-u}^n - 1}{\gamma_{v-u} - 1} = \frac{1}{n} \cdot \frac{1 - 1}{\gamma_{v-u} - 1} = 0,$$

this follows by the formula for the sum of a geometric series, and as γ_{v-u} is an n th root of unity, and as such if we raise it to power n we get 1.

We just proved that the matrix C have ones on the diagonal and zero everywhere else. Namely, it is the identity matrix, establishing our claim that the given matrix is indeed the inverse matrix to the Vandermonde matrix. \blacksquare

Let us recap, given n point-value pairs $\{(\gamma_0, y_0), \dots, (\gamma_{n-1}, y_{n-1})\}$ of a polynomial $p(x) = \sum_{i=0}^{n-1} a_i x^i$ over the set of n th roots of unity, then we can recover the coefficients of the polynomial by multiplying the vector $[y_0, y_1, \dots, y_{n-1}]$ by the matrix V^{-1} . Namely,

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \frac{1}{n} \underbrace{\begin{pmatrix} 1 & \beta_0 & \beta_0^2 & \beta_0^3 & \cdots & \beta_0^{n-1} \\ 1 & \beta_1 & \beta_1^2 & \beta_1^3 & \cdots & \beta_1^{n-1} \\ 1 & \beta_2 & \beta_2^2 & \beta_2^3 & \cdots & \beta_2^{n-1} \\ 1 & \beta_3 & \beta_3^2 & \beta_3^3 & \cdots & \beta_3^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \beta_{n-1} & \beta_{n-1}^2 & \beta_{n-1}^3 & \cdots & \beta_{n-1}^{n-1} \end{pmatrix}}_{V^{-1}} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

Let us write a polynomial $W(x) = \sum_{i=0}^{n-1} (y_i/n)x^i$. It is clear that $a_i = W(\beta_i)$. That is to recover the coefficients of $p(\cdot)$, we have to compute a polynomial $W(\cdot)$ on n values: $\beta_0, \dots, \beta_{n-1}$.

The final stroke, is to observe that $\{\beta_0, \dots, \beta_{n-1}\} = \{\gamma_0, \dots, \gamma_{n-1}\}$; indeed $\beta_i^n = (\gamma_i^{-1})^n = (\gamma_i^n)^{-1} = 1^{-1} = 1$. Namely, we can apply the **FFTA** algorithm on $W(x)$ to compute a_0, \dots, a_{n-1} .

We conclude:

Theorem 23.3.2. *Given n point-value pairs of a polynomial $p(x)$ of degree $n-1$ over the set of n powers of the n th roots of unity, we can recover the polynomial $p(x)$ in $O(n \log n)$ time.*

Theorem 23.3.3. *Given two polynomials of degree n , they can be multiplied in $O(n \log n)$ time.*

23.4. The Convolution Theorem

Given two vectors: $A = [a_0, a_1, \dots, a_n]$ and $B = [b_0, \dots, b_n]$, their dot product is the quantity

$$A \cdot B = \langle A, B \rangle = \sum_{i=0}^n a_i b_i.$$

Let A_r denote the shifting of A by $n-r$ locations to the left (we pad it with zeros; namely, $a_j = 0$ for $j \notin \{0, \dots, n\}$).

$$A_r = [a_{n-r}, a_{n+1-r}, a_{n+2-r}, \dots, a_{2n-r}]$$

where $a_j = 0$ if $j \notin [0, \dots, n]$.

Observation 23.4.1. $A_n = A$.

Example 23.4.2. For $A = [3, 7, 9, 15]$, $n = 3$

$$A_2 = [7, 9, 15, 0],$$

$$A_5 = [0, 0, 3, 7].$$

Definition 23.4.3. Let $c_i = A_i \cdot B = \sum_{j=n-i}^{2n-i} a_j b_{j-n+i}$, for $i = 0, \dots, 2n$. The vector $[c_0, \dots, c_{2n}]$ is the *convolution* of A and B .

Question 23.4.4. *How to compute the convolution of two vectors of length n ?*

Definition 23.4.5. The resulting vector $[c_0, \dots, c_{2n}]$ is the **convolution** of A and B .

Let $p(x) = \sum_{i=0}^n \alpha_i x^i$, and $q(x) = \sum_{i=0}^n \beta_i x^i$. The coefficient of x^i in $r(x) = p(x)q(x)$ is

$$d_i = \sum_{j=0}^i \alpha_j \beta_{i-j}.$$

On the other hand, we would like to compute $c_i = A_i \cdot B = \sum_{j=n-i}^{2n-i} a_j b_{j-n+i}$, which seems to be a very similar expression. Indeed, setting $\alpha_i = a_i$ and $\beta_i = b_{n-i-1}$ we get what we want.

To understand what's going on, observe that the coefficient of x^2 in the product of the two respective polynomials $p(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ and $q(x) = b_0 + b_1x + b_2x^2 + b_3x^3$ is the sum of the entries on the anti diagonal in the following matrix, where the entry in the i th row and j th column is $a_i b_j$.

	$a_0 +$	$a_1 x$	$+ a_2 x^2$	$+ a_3 x^3$
b_0				$a_2 b_0 x^2$
$+ b_1 x$			$a_1 b_1 x^2$	
$+ b_2 x^2$	$a_0 b_2 x^2$			
$+ b_3 x^3$				

Theorem 23.4.6. Given two vectors $A = [a_0, a_1, \dots, a_n]$, $B = [b_0, \dots, b_n]$ one can compute their convolution in $O(n \log n)$ time.

Proof: Let $p(x) = \sum_{i=0}^n a_n - i x^i$ and let $q(x) = \sum_{i=0}^n b_i x^i$. Compute $r(x) = p(x)q(x)$ in $O(n \log n)$ time using the convolution theorem. Let c_0, \dots, c_{2n} be the coefficients of $r(x)$. It is easy to verify, as described above, that $[c_0, \dots, c_{2n}]$ is the convolution of A and B . ■

23.4.1. Complex numbers – a quick reminder

A complex number is a pair of real numbers x and y , written as $\tau = x + iy$, where x is the **real** part and y is the **imaginary** part. Here i is of course the root of -1 . In **polar form**, we can write $\tau = r \cos \phi + ir \sin \phi = r(\cos \phi + i \sin \phi) = r e^{i\phi}$, where $r = \sqrt{x^2 + y^2}$ and $\phi = \arcsin(y/x)$. To see the last part, define the following functions by their Taylor expansion

$$\begin{aligned} \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots, \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots, \\ \text{and } e^x &= 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots. \end{aligned}$$

Since $i^2 = -1$, we have that

$$e^{ix} = 1 + i \frac{x}{1!} - \frac{x^2}{2!} - i \frac{x^3}{3!} + \frac{x^4}{4!} + i \frac{x^5}{5!} - \frac{x^6}{6!} \dots = \cos x + i \sin x.$$

The nice thing about polar form, is that given two complex numbers $\tau = r e^{i\phi}$ and $\tau' = r' e^{i\phi'}$, multiplying them is now straightforward. Indeed, $\tau \cdot \tau' = r e^{i\phi} \cdot r' e^{i\phi'} = r r' e^{i(\phi+\phi')}$. Observe that the function $e^{i\phi}$ is 2π periodic (i.e., $e^{i\phi} = e^{i(\phi+2\pi)}$), and $1 = e^{i0}$. As such, an n th root of 1, is a complex number $\tau = r e^{i\phi}$ such that $\tau^n = r^n e^{in\phi} = e^{i0}$. Clearly, this implies that $r = 1$, and there must be an integer j , such that

$$n\phi = 0 + 2\pi j \implies \phi = j(2\pi/n).$$

These are all distinct values for $j = 0, \dots, n-1$, which are the n distinct roots of unity.

Chapter 24

Sorting Networks

The world is what it is; men who are nothing, who allow themselves to become nothing, have no place in it.

A bend in the river, V. S. Naipul

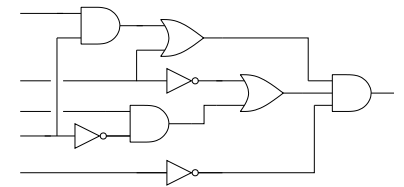
24.1. Model of Computation

It is natural to ask if one can perform a computational task considerably faster by using a different architecture (i.e., a different computational model).

The answer to this question is a resounding yes. A cute example is the *Macaroni sort* algorithm. We are given a set $S = \{s_1, \dots, s_n\}$ of n real numbers in the range (say) $[1, 2]$. We get a lot of Macaroni (this are longish and very narrow tubes of pasta), and cut the i th piece to be of length s_i , for $i = 1, \dots, n$. Next, take all these pieces of pasta in your hand, make them stand up vertically, with their bottom end lying on a horizontal surface. Next, lower your handle till it hit the first (i.e., tallest) piece of pasta. Take it out, measure its height, write down its number, and continue in this fashion till you have extracted all the pieces of pasta. Clearly, this is a sorting algorithm that works in linear time. But we know that sorting takes $\Omega(n \log n)$ time. Thus, this algorithm is much faster than the standard sorting algorithms.

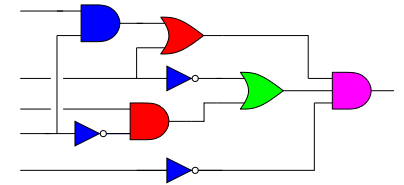
This faster algorithm was achieved by changing the computation model. We allowed new “strange” operations (cutting a piece of pasta into a certain length, picking the longest one in constant time, and measuring the length of a pasta piece in constant time). Using these operations we can sort in linear time.

If this was all we can do with this approach, that would have only been a curiosity. However, interestingly enough, there are natural computation models which are considerably stronger than the standard model of computation. Indeed, consider the task of computing the output of the circuit on the right (here, the input is boolean values on the input wires on the left, and the output is the single output on the right).



Clearly, this can be solved by ordering the gates in the “right” order (this can be done by topological sorting), and then computing the value of the gates one by one in this order, in such a way that a gate being computed knows the values arriving on its input wires. For the circuit above, this would require 8 units of time, since there are 8 gates.

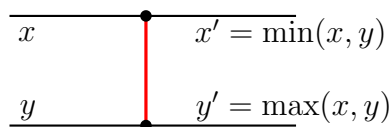
However, if you consider this circuit more carefully, one realized that we can compute this circuit in 4 time units. By using the fact that several gates are independent of each other, and we can compute them in parallel, as depicted on the right. Furthermore, circuits are inherently parallel and we should be able to take advantage of this fact.



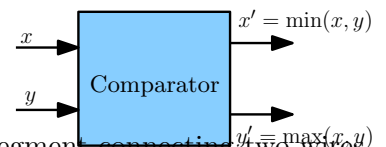
So, let us consider the classical problem of sorting n numbers. The question is whether we can sort them in *sublinear* time by allowing parallel comparisons. To this end, we need to precisely define our computation model.

24.2. Sorting with a circuit – a naive solution

We are going to design a circuit, where the inputs are the numbers and we compare two numbers using a comparator gate. Such a gate has two inputs and two outputs, and it is depicted on the right.

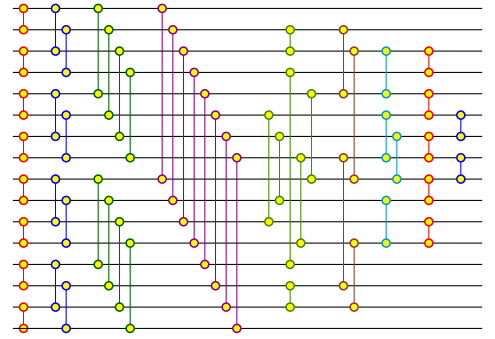


We usually depict such a gate as a vertical segment connecting two wires, as depicted on the right. This would make drawing and arguing about sorting networks easier.



Our circuits would be depicted by horizontal lines, with vertical segments (i.e., gates) connecting between them. For example, see complete sorting network depicted on the right.

The inputs come on the wires on the left, and are output on the wires on the right. The largest number is output on the bottom line. Somewhat surprisingly, one can generate circuits from known sorting algorithms.



24.2.1. Definitions

Definition 24.2.1. A *comparison network* is a DAG (directed acyclic graph), with n inputs and n outputs, where each gate (i.e., done) has two inputs and two outputs (i.e., two incoming edges, and two outgoing edges).

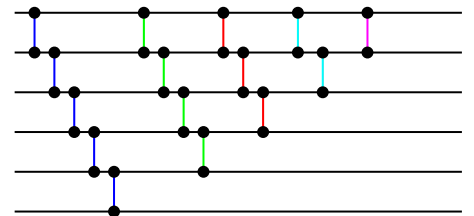
Definition 24.2.2. The *depth* of a wire is 0 at the input. For a gate with two inputs of depth d_1 and d_2 the depth on the output wire is $1 + \max(d_1, d_2)$. The *depth* of a comparison network is the maximum depth of an output wire.

Definition 24.2.3. A *sorting network* is a comparison network such that for any input, the output is monotonically sorted. The *size* of a sorting network is the number of gates in the sorting network. The *running time* of a sorting network is just its depth.

24.2.2. Sorting network based on insertion sort



Consider the sorting circuit on the left. Clearly, this is just the inner loop of the standard insertion sort. As such, if we repeat this loop, we get the sorting network on the right. It is easy to argue that this circuit sorts correctly all inputs (we removed some unnecessary gates).



An alternative way of drawing this sorting network is depicted in Figure 24.1 (ii). The next natural question, is how much time does it take for this circuit to sort the n numbers. Observe, that the running time of the algorithm is how many different time ticks we have to wait till the result stabilizes in all the gates. In our example, the alternative drawing immediately tell us how to schedule the computation of the gates. See Figure 24.1 (ii).

In particular, the above discussion implies the following result.

Lemma 24.2.4. *The sorting network based on insertion sort has $O(n^2)$ gates, and requires $2n - 1$ time units to sort n numbers.*

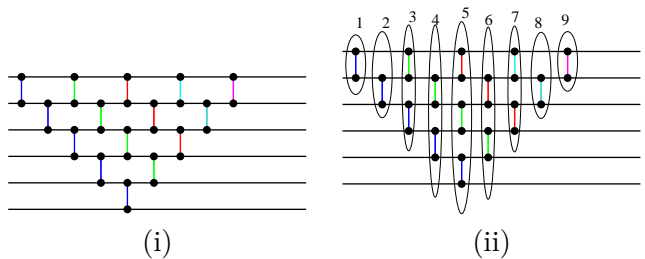


Figure 24.1: The sorting network inspired by insertion sort.

24.3. The Zero-One Principle

The *zero-one principle* states that if a comparison network sort correctly all binary inputs (i.e., every number is either 0 or 1) then it sorts correctly all inputs. We (of course) need to prove that the zero-one principle is true.

Lemma 24.3.1. *If a comparison network transforms the input sequence $a = \langle a_1, a_2, \dots, a_n \rangle$ into the output sequence $b = \langle b_1, b_2, \dots, b_n \rangle$, then for any monotonically increasing function f , the network transforms the input sequence $f(a) = \langle f(a_1), \dots, f(a_n) \rangle$ into the sequence $f(b) = \langle f(b_1), \dots, f(b_n) \rangle$.*

Proof: Consider a single comparator with inputs x and y , and outputs $x' = \min(x, y)$ and $y' = \max(x, y)$. If $f(x) = f(y)$ then the claim trivially holds for this comparator. If $f(x) < f(y)$ then clearly

$$\begin{aligned}\max(f(x), f(y)) &= f(\max(x, y)) \text{ and} \\ \min(f(x), f(y)) &= f(\min(x, y)),\end{aligned}$$

since $f(\cdot)$ is monotonically increasing. As such, for the input $\langle x, y \rangle$, for $x < y$, we have output $\langle x, y \rangle$. Thus, for the input $\langle f(x), f(y) \rangle$ the output is $\langle f(x), f(y) \rangle$. Similarly, if $x > y$, the output is $\langle y, x \rangle$. In this case, for the input $\langle f(x), f(y) \rangle$ the output is $\langle f(y), f(x) \rangle$. This establish the claim for a single comparator.

Now, we claim by induction that if a wire carry a value a_i , when the sorting network get input a_1, \dots, a_n , then for the input $f(a_1), \dots, f(a_n)$ this wire would carry the value $f(a_i)$.

This is proven by induction on the depth on the wire at each point. If the point has depth 0, then its an input and the claim trivially hold. So, assume it holds for all points in our circuits of depth at most i , and consider a point p on a wire of depth $i + 1$. Let G be the gate which this wire is an output of. By induction, we know the claim holds for the inputs of G (which have depth at most i). Now, we the claim holds for the gate G itself, which implies the claim apply the above claim to the gate G , which implies the claim holds at p . ■

Theorem 24.3.2. *If a comparison network with n inputs sorts all 2^n binary strings of length n correctly, then it sorts all sequences correctly.*

Proof: Assume for the sake of contradiction, that it sorts incorrectly the sequence a_1, \dots, a_n . Let b_1, \dots, b_n be the output sequence for this input.

Let $a_i < a_k$ be the two numbers that are output in incorrect order (i.e. a_k appears before a_i in the output). Let

$$f(x) = \begin{cases} 0 & x \leq a_i \\ 1 & x > a_i. \end{cases}$$

Clearly, by the above lemma (Lemma 24.3.1), for the input

$$\langle f(a_1), \dots, f(a_n) \rangle,$$

which is a binary sequence, the circuit would output $\langle f(b_1), \dots, f(b_n) \rangle$. But then, this sequence looks like

$$000..0????f(a_k)????f(a_i)??1111$$

but $f(a_i) = 0$ and $f(a_j) = 1$. Namely, the output is a sequence of the form $????1????0????$, which is not sorted.

Namely, we have a binary input (i.e., $\langle f(b_1), \dots, f(b_n) \rangle$) for which the comparison network does not sort it correctly. A contradiction to our assumption. ■

24.4. A bitonic sorting network

Definition 24.4.1. A *bitonic sequence* is a sequence which is first increasing and then decreasing, or can be circularly shifted to become so.

Example 24.4.2. The sequences $(1, 2, 3, \pi, 4, 5, 4, 3, 2, 1)$ and $(4, 5, 4, 3, 2, 1, 1, 2, 3)$ are bitonic, while the sequence $(1, 2, 1, 2)$ is not bitonic.

Observation 24.4.3. *A binary bitonic sequence (i.e., bitonic sequence made out only of zeroes and ones) is either of the form $0^i 1^j 0^k$ or of the form $1^i 0^j 1^k$, where 0^i (resp, 1^i) denote a sequence of i zeros (resp., ones).*

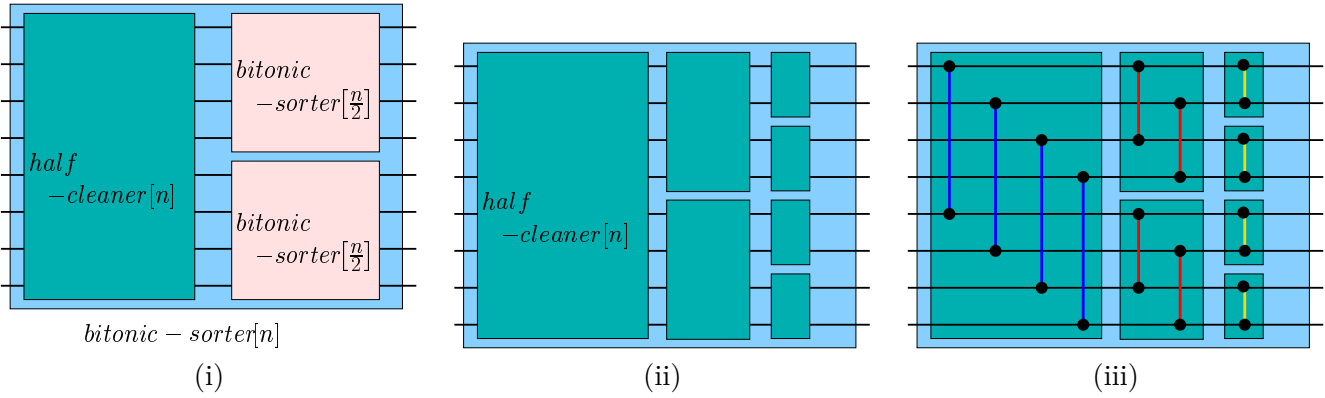
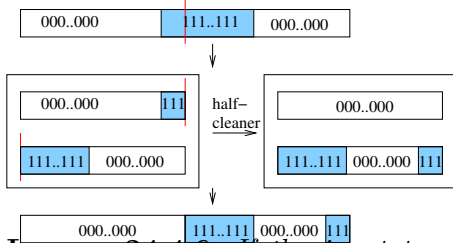
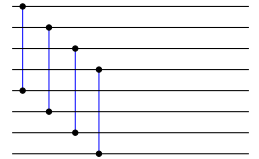


Figure 24.2: Depicted are the (i) recursive construction of **BitonicSorter** $[n]$, (ii) opening up the recursive construction, and (iii) the resulting comparison network.

Definition 24.4.4. A **bitonic sorter** is a comparison network that sorts all bitonic sequences correctly.

Definition 24.4.5. A **half-cleaner** is a comparison network, connecting line i with line $i + n/2$. In particular, let **Half-Cleaner** $[n]$ denote the half-cleaner with n inputs. Note, that the depth of a **Half-Cleaner** $[n]$ is one, see figure on the right.



It is beneficial to consider what a half-cleaner do to an input which is a (binary) bitonic sequence. Clearly, in the specific example, depicted on the left, we have that the left half size is clean and all equal to 0. Similarly, the right size of the output is bitonic.

Specifically, one can prove by simple (but tedious) case analysis that the following lemma holds.

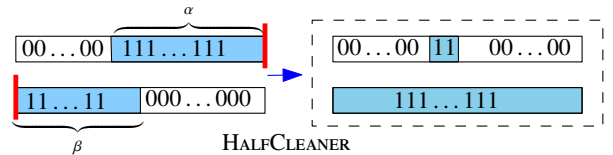
Lemma 24.4.6. *If the input to a half-cleaner (of size n) is a binary bitonic sequence then for the output sequence we have that*

- (i) *the elements in the top half are smaller than the elements in bottom half, and*
- (ii) *one of the halves is clean, and the other is bitonic.*

Proof: If the sequence is of the form $0^i 1^j 0^k$ and the block of ones is completely on the left side (i.e., its part of the first $n/2$ bits) or the right side, the claim trivially holds. So, assume that the block of ones starts at position $n/2 - \beta$ and ends at $n/2 + \alpha$.

If $n/2 - \alpha \geq \beta$ then this is exactly the case depicted above and claim holds. If $n/2 - \alpha < \beta$ then the second half is going to be all ones, as depicted on the right. Implying the claim for this case.

A similar analysis holds if the sequence is of the form $1^i 0^j 1^k$. ■



This suggests a simple recursive construction of **BitonicSorter** $[n]$, see **Figure 24.2**, and we have the following lemma.

Lemma 24.4.7. ***BitonicSorter** $[n]$ sorts bitonic sequences of length $n = 2^k$, it uses $(n/2)k = (n/2) \lg n$ gates, and it is of depth $k = \lg n$.*

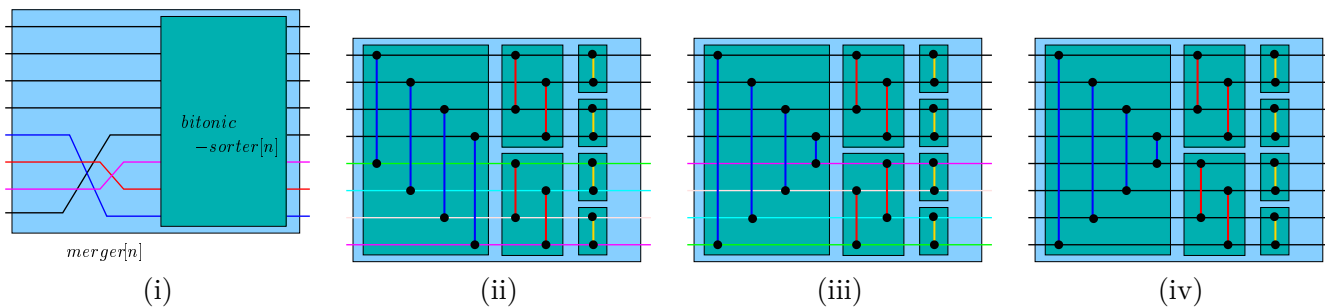


Figure 24.3: (i) **Merger** via flipping the lines of bitonic sorter. (ii) A **BitonicSorter**. (iii) The **Merger** after we “physically” flip the lines, and (iv) An equivalent drawing of the resulting **Merger**.

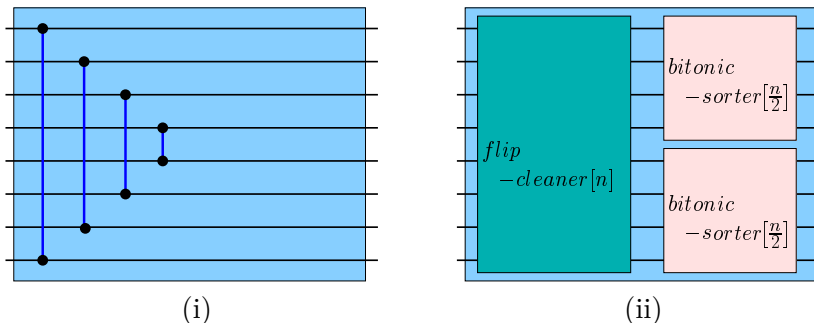


Figure 24.4: (i) **FlipCleaner**[n], and (ii) **Merger**[n] described using **FlipCleaner**.

24.4.1. Merging sequence

Next, we deal with the following merging question. Given two *sorted* sequences of length $n/2$, how do we merge them into a single sorted sequence?

The idea here is concatenate the two sequences, where the second sequence is being flipped (i.e., reversed). It is easy to verify that the resulting sequence is bitonic, and as such we can sort it using the **BitonicSorter**[n].

Specifically, given two sorted sequences $a_1 \leq a_2 \leq \dots \leq a_n$ and $b_1 \leq b_2 \leq \dots \leq b_n$, observe that the sequence $a_1, a_2, \dots, a_n, b_n, b_{n-1}, b_{n-2}, \dots, b_2, b_1$ is bitonic.

Thus, to merge two sorted sequences of length $n/2$, just flip one of them, and use **BitonicSorter**[n], see **Figure 24.3**. This is of course illegal, and as such we take **BitonicSorter**[n] and physically flip the last $n/2$ entries. The process is depicted in **Figure 24.3**. The resulting circuit **Merger** takes two sorted sequences of length $n/2$, and return a sorted sequence of length n .

It is somewhat more convenient to describe the **Merger** using a **FlipCleaner** component. See **Figure 24.4**

Lemma 24.4.8. *The circuit **Merger**[n] gets as input two sorted sequences of length $n/2 = 2^{k-1}$, it uses $(n/2)k = (n/2)\lg n$ gates, and it is of depth $k = \lg n$, and it outputs a sorted sequence.*

24.5. Sorting Network

We are now in the stage, where we can build a sorting network. To this end, we just implement *merge sort* using the *Merger*[n] component. The resulting component *Sorter*[n] is depicted on the right using a recursive construction.

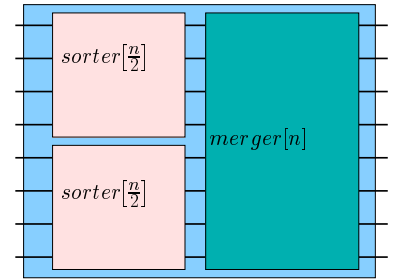
Lemma 24.5.1. *The circuit *Sorter*[n] is a sorting network (i.e., it sorts any n numbers) using $G(n) = O(n \log^2 n)$ gates. It has depth $O(\log^2 n)$. Namely, *Sorter*[n] sorts n numbers in $O(\log^2 n)$ time.*

Proof: The number of gates is

$$G(n) = 2G(n/2) + \text{Gates}(\text{Merger}[n]).$$

Which is $G(n) = 2G(n/2) + O(n \log n) = O(n \log^2 n)$.

As for the depth, we have that $D(n) = D(n/2) + \text{Depth}(\text{Merger}[n]) = D(n/2) + O(\log(n))$, and thus $D(n) = O(\log^2 n)$, as claimed. ■



24.6. Faster sorting networks

One can build a sorting network of logarithmic depth (see [AKS83]). The construction however is very complicated. A simpler parallel algorithm would be discussed sometime in the next lectures. BTW, the AKS construction [AKS83] mentioned above, is better than bitonic sort for n larger than 2^{8046} .

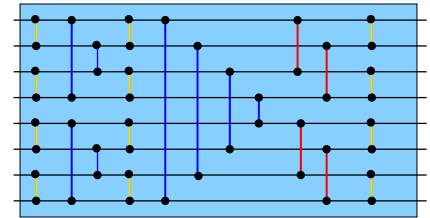


Figure 24.5: *Sorter*[8].

Chapter 25

Union Find

25.1. Union-Find

25.1.1. Requirements from the data-structure

We want to maintain a collection of sets, under the following operations.

- (i) **makeSet**(x) - creates a set that contains the single element x .
- (ii) **find**(x) - returns the set that contains x .
- (iii) **union**(A, B) - returns the set which is the union of A and B . Namely $A \cup B$. Namely, this operation merges the two sets A and B and return the merged set.

25.1.2. Amortized analysis

We use a data-structure as a black-box inside an algorithm (for example Union-Find in Kruskal algorithm for computing minimum spanning tree). So far, when we design a data-structure we cared about worst case time for operation. Note however, that this is not necessarily the right measure. Indeed, we care about the *overall* running time spend on doing operations in the data-structure, and less about its running time for a single operation.

Formally, the *amortized running-time* of an operation is the average time it takes to perform an operation on the data-structure. Formally, the amortized time of an operation is $\frac{\text{overall running time}}{\text{number of operations}}$.

25.1.3. The data-structure

To implement this operations, we are going to use Reversed Trees. In a reversed tree, every element is stored in its own node. A node has one pointer to its parent. A set is uniquely identified with the element stored in the root of such a reversed tree. See **Figure 25.1** for an example of how such a reversed tree looks like.

We implement the operations of the Union-Find data structure as follows:

- (A) **makeSet**: Create a singleton pointing to itself:

Scene: It's a fine sunny day in the forest, and a rabbit is sitting outside his burrow, tippy-tapping on his typewriter.

Along comes a fox, out for a walk.

Fox: "What are you working on?"

Rabbit: "My thesis."

Fox: "Hmmm. What's it about?"

Rabbit: "Oh, I'm writing about how rabbits eat foxes."

Fox: (incredulous pause) "That's ridiculous! Any fool knows that rabbits don't eat foxes."

Rabbit: "Sure they do, and I can prove it. Come with me."

They both disappear into the rabbit's burrow. After a few minutes, the rabbit returns, alone, to his typewriter and resumes typing.

Scene inside the rabbit's burrow: In one corner, there is a pile of fox bones. In another corner, a pile of wolf bones. On the other side of the room, a huge lion is belching and picking his teeth.

(The End)

Moral: It doesn't matter what you choose for a thesis subject.

It doesn't matter what you use for data.

What does matter is who you have for a thesis advisor.

-- Anonymous

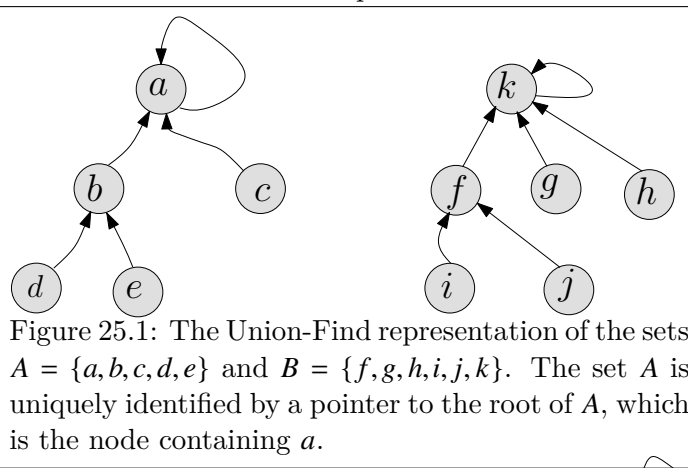


Figure 25.1: The Union-Find representation of the sets $A = \{a, b, c, d, e\}$ and $B = \{f, g, h, i, j, k\}$. The set A is uniquely identified by a pointer to the root of A , which is the node containing a .

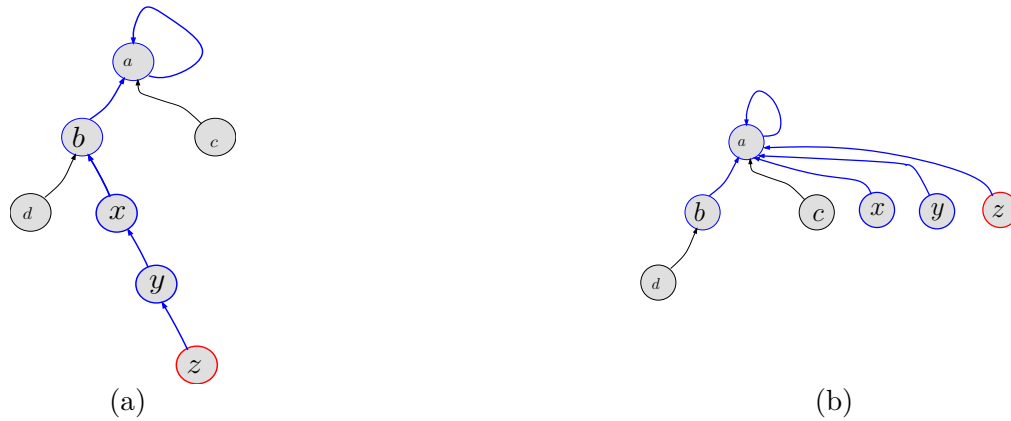


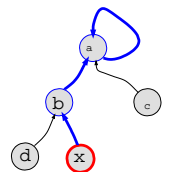
Figure 25.2: (a) The tree before performing $\text{find}(z)$, and (b) The reversed tree after performing $\text{find}(z)$ that uses path compression.

<pre> makeSet(x) $\bar{p}(x) \leftarrow x$ $\text{rank}(x) \leftarrow 0$ find(x) if $x \neq \bar{p}(x)$ then $\bar{p}(x) \leftarrow \text{find}(\bar{p}(x))$ return $\bar{p}(x)$ </pre>	<pre> union(x, y) $A \leftarrow \text{find}(x)$ $B \leftarrow \text{find}(y)$ if $\text{rank}(A) > \text{rank}(B)$ then $\bar{p}(B) \leftarrow A$ else $\bar{p}(A) \leftarrow B$ if $\text{rank}(A) = \text{rank}(B)$ then $\text{rank}(B) \leftarrow \text{rank}(B) + 1$ </pre>
---	--

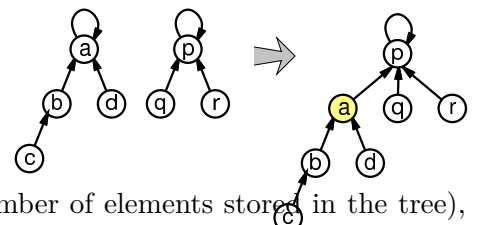
Figure 25.3: The pseudo-code for the Union-Find data-structure that uses both path-compression and union by rank. For element x , we denote the parent pointer of x by $\bar{p}(x)$.

(B) $\text{find}(x)$: We start from the node that contains x , and we start traversing up the tree, following the parent pointer of the current node, till we get to the root, which is just a node with its parent pointer pointing to itself.

Thus, doing a $\text{find}(x)$ operation in the reversed tree shown on the right, involve going up the tree from $x \rightarrow b \rightarrow a$, and returning a as the set.



(C) $\text{union}(a,p)$: We merge two sets, by hanging the root of one tree, on the root of the other. Note, that this is a destructive operation, and the two original sets no longer exist. Example of how the new tree representing the new set is depicted on the right.



Note, that in the worst case, depth of tree can be linear in n (the number of elements stored in the tree), so the find operation might require $\Omega(n)$ time. To see that this worst case is realizable perform the following sequence of operations: create n sets of size 1, and repeatedly merge the current set with a singleton. If we always merge (i.e., do union) the current set with a singleton by hanging the current set on the singleton, the end result would be a reversed tree which looks like a linked list of length n . Doing a find on the deepest element will take linear time.

So, the question is how to further improve the performance of this data-structure. We are going to do this, by using two “hacks”:

- (i) **Union by rank**: Maintain for every tree, in the root, a bound on its depth (called **rank**). Always hang the smaller tree on the larger tree.
- (ii) **Path compression**: Since, anyway, we travel the path to the root during a find operation, we might as

well hang all the nodes on the path directly on the root.

An example of the effects of path compression are depicted in [Figure 25.2](#). For the pseudo-code of the `makeSet`, `union` and `find` using path compression and union by rank, see [Figure 25.3](#).

We maintain a rank which is associated with each element in the data-structure. When a singleton is being created, its associated rank is set to zero. Whenever two sets are being merged, we update the rank of the new root of the merged trees. If the two trees have different root ranks, then the rank of the root does not change. If they are equal then we set the rank of the new root to be larger by one.

25.2. Analyzing the Union-Find Data-Structure

Definition 25.2.1. A node in the union-find data-structure is a *leader* if it is the root of a (reversed) tree.

Lemma 25.2.2. *Once a node stop being a leader (i.e., the node in top of a tree), it can never become a leader again.*

Proof: Note, that an element x can stop being a leader only because of a `union` operation that hanged x on an element y . From this point on, the only operation that might change x parent pointer, is a `find` operation that traverses through x . Since path-compression can only change the parent pointer of x to point to some other element y , it follows that x parent pointer will never become equal to x again. Namely, once x stop being a leader, it can never be a leader again. ■

Lemma 25.2.3. *Once a node stop being a leader then its rank is fixed.*

Proof: The rank of an element changes only by the `union` operation. However, the `union` operation changes the rank, only for elements that are leader after the operation is done. As such, if an element is no longer a leader, than its rank is fixed. ■

Lemma 25.2.4. *Ranks are monotonically increasing in the reversed trees, as we travel from a node to the root of the tree.*

Proof: It is enough to prove, that for every edge $u \rightarrow v$ in the data-structure, we have $\text{rank}(u) < \text{rank}(v)$. The proof is by induction. Indeed, in the beginning of time, all sets are singletons, with rank zero, and the claim trivially holds.

Next, assume that the claim holds at time t , just before we perform an operation. Clearly, if this operation is `union` (A, B), and assume that we hanged $\text{root}(A)$ on $\text{root}(B)$. In this case, it must be that $\text{rank}(\text{root}(B))$ is now larger than $\text{rank}(\text{root}(A))$, as can be easily verified. As such, if the claim held before the `union` operation, then it is also true after it was performed.

If the operation is `find`, and we traverse the path π , then all the nodes of π are made to point to the last node v of π . However, by induction, $\text{rank}(v)$ is larger than the rank of all the other nodes of π . In particular, all the nodes that get compressed, the rank of their new parent, is larger than their own rank. ■

Lemma 25.2.5. *When a node gets rank k than there are at least $\geq 2^k$ elements in its subtree.*

Proof: The proof is by induction. For $k = 0$ it is obvious since a singleton has a rank zero, and a single element in the set. Next observe that a node gets rank k only if the merged two roots has rank $k - 1$. By induction, they have 2^{k-1} nodes (each one of them), and thus the merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes. ■

Lemma 25.2.6. *The number of nodes that get assigned rank k throughout the execution of the Union-Find data-structure is at most $n/2^k$.*

Proof: Again, by induction. For $k = 0$ it is obvious. We charge a node v of rank k to the two elements u and v of rank $k - 1$ that were leaders that were used to create the new larger set. After the merge v is of rank k and u is of rank $k - 1$ and it is no longer a leader (it can not participate in a union as a leader any more). Thus, we can charge this event to the two (no longer active) nodes of degree $k - 1$. Namely, u and v .

By induction, we have that the algorithm created at most $n/2^{k-1}$ nodes of rank $k - 1$, and thus the number of nodes of rank k created by the algorithm is at most $\leq (n/2^{k-1})/2 = n/2^k$. ■

Lemma 25.2.7. *The time to perform a single **find** operation when we perform union by rank and path compression is $O(\log n)$ time.*

Proof: The rank of the leader v of a reversed tree T , bounds the depth of a tree T in the Union-Find data-structure. By the above lemma, if we have n elements, the maximum rank is $\lg n$ and thus the depth of a tree is at most $O(\log n)$. ■

Surprisingly, we can do much better.

Theorem 25.2.8. *If we perform a sequence of m operations over n elements, the overall running time of the Union-Find data-structure is $O((n + m) \log^* n)$.*

We remind the reader that $\log^*(n)$ is the number one has to take \lg of a number to get a number smaller than two (there are other definitions, but they are all equivalent, up to adding a small constant). Thus, $\log^* 2 = 1$ and $\log^* 2^2 = 2$. Similarly, $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$. Similarly, $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$. Things get really exciting, when one considers

$$\log^* 2^{2^{2^{2^2}}} = \log^* 2^{65536} = 5.$$

However, \log^* is a monotone increasing function. And $\beta = 2^{2^{2^2}} = 2^{65536}$ is a huge number (considerably larger than the number of atoms in the universe). Thus, for all practical purposes, \log^* returns a value which is smaller than 5. Intuitively, **Theorem 25.2.8** states (in the amortized sense), that the Union-Find data-structure takes constant time per operation (unless n is larger than β which is unlikely).

It would be useful to look on the inverse function to \log^* .

Definition 25.2.9. Let $\text{Tower}(b) = 2^{\text{Tower}(b-1)}$ and $\text{Tower}(0) = 1$.

So, $\text{Tower}(i)$ is just a tower of $2^{2^{\dots^2}}$ of height i . Observe that $\log^*(\text{Tower}(i)) = i$.

Definition 25.2.10. For $i \geq 0$, let $\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]$; that is

$$\text{Block}(i) = [z, 2^{z-1}] \quad \text{for} \quad z = \text{Tower}(i - 1) + 1.$$

For technical reasons, we define $\text{Block}(0) = [0, 1]$. As such,

$$\begin{aligned} \text{Block}(0) &= [0, 1] \\ \text{Block}(1) &= [2, 2] \\ \text{Block}(2) &= [3, 4] \\ \text{Block}(3) &= [5, 16] \\ \text{Block}(4) &= [17, 65536] \\ \text{Block}(5) &= [65537, 2^{65536}] \\ &\vdots \end{aligned}$$

The running time of $\text{find}(x)$ is proportional to the length of the path from x to the root of the tree that contains x . Indeed, we start from x and we visit the sequence:

$$x_1 = x, x_2 = \bar{p}(x) = \bar{p}(x_1), \dots, x_i = \bar{p}(x_{i-1}), \dots, x_m = \text{root of tree}.$$

Clearly, we have for this sequence: $\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \dots < \text{rank}(x_m)$, and the time it takes to perform $\text{find}(x)$ is proportional to m , the length of the path from x to the root of the tree containing x .

Definition 25.2.11. A node x is in the i th block if $\text{rank}(x) \in \text{Block}(i)$.

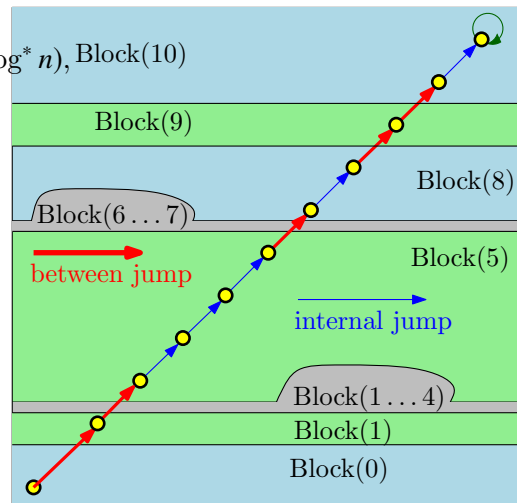
We are now looking for ways to pay for the find operation, since the other two operations take constant time.

Observe, that the maximum rank of a node v is $O(\log n)$, and the number of blocks is $O(\log^* n)$, since $O(\log n)$ is in the block $\text{Block}(c \log^* n)$, $\text{Block}(10)$ for c a constant sufficiently large.

In particular, consider a $\text{find}(x)$ operation, and let π be the path visited. Next, consider the ranks of the elements of π , and imagine partitioning π into which blocks each element rank belongs to. An example of such a path is depicted on the right. The price of the find operation is the length of π .

Formally, for a node x , $v = \text{index}_B(x)$ is the index of the block that contains $\text{rank}(x)$. Namely, $\text{rank}(x) \in \text{Block}(\text{index}_B(x))$. As such, $\text{index}_B(x)$ is the **block of x** .

Now, during a find operation, since the ranks of the nodes we visit are monotone increasing, once we pass through from a node v in the i th block into a node in the $(i + 1)$ th block, we can never go back to the i th block (i.e., visit elements with rank in the i th block). As such, we can charge the visit to nodes in π that are next to a element in a different block, to the number of blocks (which is $O(\log^* n)$).



Definition 25.2.12. Consider a path π traversed by a find operation. Along the path π , an element x , such that $\bar{p}(x)$ is in a different block, is a **jump between blocks**.

On the other hand, a jump during a find operation inside a block is called an **internal jump**; that is, x and $\bar{p}(x)$ are in the same block.

Lemma 25.2.13. During a single $\text{find}(x)$ operation, the number of jumps between blocks along the search path is $O(\log^* n)$.

Proof: Consider the search path $\pi = x_1, \dots, x_m$, and consider the list of numbers $0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \dots \leq \text{index}_B(x_m)$. We have that $\text{index}_B(x_m) = O(\log^* n)$. As such, the number of elements x in π such that $\text{index}_B(x) \neq \text{index}_B(\bar{p}(x))$ is at most $O(\log^* n)$. ■

Consider the case that x and $\bar{p}(x)$ are both the same block (i.e., $\text{index}_B(x) = \text{index}_B(\bar{p}(x))$) and we perform a find operation that passes through x . Let $r_{\text{bef}} = \text{rank}(\bar{p}(x))$ before the find operation, and let r_{aft} be $\text{rank}(\bar{p}(x))$ after the find operation. Observe, that because of path compression, we have $r_{\text{aft}} > r_{\text{bef}}$. Namely, when we jump inside a block, we do some work: we make the parent pointer of x jump forward and the new parent has higher rank. We will charge such internal block jumps to this “progress”.

Lemma 25.2.14. At most $|\text{Block}(i)| \leq \text{Tower}(i)$ find operations can pass through an element x , which is in the i th block (i.e., $\text{index}_B(x) = i$) before $\bar{p}(x)$ is no longer in the i th block. That is $\text{index}_B(\bar{p}(x)) > i$.

Proof: Indeed, by the above discussion, the parent of x increases its rank every-time an internal jump goes through x . Since there at most $|\text{Block}(i)|$ different values in the i th block, the claim follows. The inequality $|\text{Block}(i)| \leq \text{Tower}(i)$ holds by definition, see **Definition 25.2.10**. ■

Lemma 25.2.15. *There are at most $n/\text{Tower}(i)$ nodes that have ranks in the i th block throughout the algorithm execution.*

Proof: By Lemma 25.2.6, we have that the number of elements with rank in the i th block is at most

$$\sum_{k \in \text{Block}(i)} \frac{n}{2^k} = \sum_{k=\text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{n}{2^k} = n \cdot \sum_{k=\text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{1}{2^k} \leq \frac{n}{2^{\text{Tower}(i-1)}} = \frac{n}{\text{Tower}(i)}. \quad \blacksquare$$

Lemma 25.2.16. *The number of internal jumps performed, inside the i th block, during the lifetime of the union-find data-structure is $O(n)$.*

Proof: An element x in the i th block, can have at most $|\text{Block}(i)|$ internal jumps, before all jumps through x are jumps between blocks, by Lemma 25.2.14. There are at most $n/\text{Tower}(i)$ elements with ranks in the i th block, throughout the algorithm execution, by Lemma 25.2.15. Thus, the total number of internal jumps is

$$|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n. \quad \blacksquare$$

We are now ready for the last step.

Lemma 25.2.17. *The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.*

Proof: Every internal jump can be associated with the block it is being performed in. Every block contributes $O(n)$ internal jumps throughout the execution of the union-find data-structures, by Lemma 25.2.16. There are $O(\log^* n)$ blocks. As such there are at most $O(n \log^* n)$ internal jumps. \blacksquare

Lemma 25.2.18. *The overall time spent on m find operations, throughout the lifetime of a union-find data-structure defined over n elements, is $O((m+n) \log^* n)$.*

Theorem 25.2.8 now follows readily from the above discussion.

Chapter 26

Approximate Max Cut

We had encountered in the previous lecture examples of using rounding techniques for approximating discrete optimization problems. So far, we had seen such techniques when the relaxed optimization problem is a linear program. Interestingly, it is currently known how to solve optimization problems that are considerably more general than linear programs. Specifically, one can solve *convex programming*. Here the feasible region is convex. How to solve such an optimization problems is outside the scope of this course. It is however natural to ask what can be done if one assumes that one can solve such general continuous optimization problems exactly.

In the following, we show that (optimization problem) max cut can be relaxed into a weird continuous optimization problem. Furthermore, this semi-definite program can be solved exactly efficiently. Maybe more surprisingly, we can round this continuous solution and get an improved approximation.

26.1. Problem Statement

Given an undirected graph $G = (V, E)$ and nonnegative weights ω_{ij} , for all $ij \in E$, the **maximum cut problem** (**MAX CUT**) is that of finding the set of vertices S that maximizes the weight of the edges in the cut (S, \bar{S}) ; that is, the weight of the edges with one endpoint in S and the other in \bar{S} . For simplicity, we usually set $\omega_{ij} = 0$ for $ij \notin E$ and denote the weight of a cut (S, \bar{S}) by $w(S, \bar{S}) = \sum_{i \in S, j \in \bar{S}} \omega_{ij}$.

This problem is **NP-COMplete**, and hard to approximate within a certain constant.

Given a graph with vertex set $V = \{1, \dots, n\}$ and nonnegative weights ω_{ij} , the weight of the maximum cut $w(S, \bar{S})$ is given by the following integer quadratic program:

$$(Q) \quad \max \quad \frac{1}{2} \sum_{i < j} \omega_{ij} (1 - y_i y_j)$$

$$\text{subject to: } \quad y_i \in \{-1, 1\} \quad \forall i \in V.$$

Indeed, set $S = \{i \mid y_i = 1\}$. Clearly, $w(S, \bar{S}) = \frac{1}{2} \sum_{i < j} \omega_{ij} (1 - y_i y_j)$.

Solving quadratic integer programming is of course **NP-HARD**. Thus, we will relax it, by thinking about the numbers y_i as unit vectors in higher dimensional space. If so, the multiplication of the two vectors, is now replaced by dot product. We have:

$$(P) \quad \max \quad \gamma = \frac{1}{2} \sum_{i < j} \omega_{ij} (1 - \langle v_i, v_j \rangle)$$

$$\text{subject to: } \quad v_i \in \mathbb{S}^{(n)} \quad \forall i \in V,$$

where $\mathbb{S}^{(n)}$ is the n dimensional unit sphere in \mathbb{R}^{n+1} . This is an instance of semi-definite programming, which is a special case of convex programming, which can be solved in polynomial time (solved here means approximated within a factor of $(1 + \varepsilon)$ of optimal, for any arbitrarily small $\varepsilon > 0$, in polynomial time). Namely, the solver finds a feasible solution with a the target function being arbitrarily close to the optimal solution. Observe that (P) is a relaxation of (Q), and as such the optimal solution of (P) has value larger than the optimal value of (Q).

The intuition is that vectors that correspond to vertices that should be on one side of the cut, and vertices on the other sides, would have vectors which are faraway from each other in (P). Thus, we compute the optimal solution for (P), and we uniformly generate a random vector \mathbf{r} on the unit sphere $\mathbb{S}^{(n)}$. This induces a hyperplane h which passes through the origin and is orthogonal to \mathbf{r} . We next assign all the vectors that are on one side of h to S , and the rest to \bar{S} .

Summarizing, the algorithm is as follows: First, we solve (P), next, we pick a random vector \mathbf{r} uniformly on the unit sphere $\mathbb{S}^{(n)}$. Finally, we set

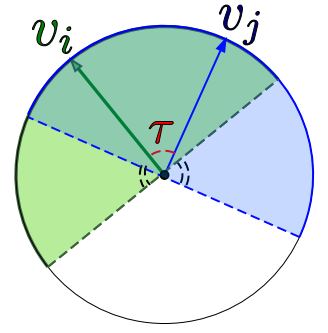
$$S = \left\{ v_i \mid \langle v_i, \mathbf{r} \rangle \geq 0 \right\}.$$

26.1.1. Analysis

The intuition of the above rounding procedure, is that with good probability, vectors in the solution of (P) that have large angle between them would be separated by this cut.

Lemma 26.1.1. *We have $\mathbb{P}[\text{sign}(\langle v_i, \mathbf{r} \rangle) \neq \text{sign}(\langle v_j, \mathbf{r} \rangle)] = \frac{1}{\pi} \arccos(\langle v_i, v_j \rangle)$.*

Proof: Let us think about the vectors v_i, v_j and \mathbf{r} as being in the plane. To see why this is a reasonable assumption, consider the plane g spanned by v_i and v_j , and observe that for the random events we consider, only the direction of \mathbf{r} matter, which can be decided by projecting \mathbf{r} on g , and normalizing it to have length 1. Now, the sphere is symmetric, and as such, sampling \mathbf{r} randomly from $\mathbb{S}^{(n)}$, projecting it down to g , and then normalizing it, is equivalent to just choosing uniformly a vector from the unit circle.



Now, $\text{sign}(\langle v_i, \mathbf{r} \rangle) \neq \text{sign}(\langle v_j, \mathbf{r} \rangle)$ happens only if \mathbf{r} falls in the double wedge formed by the lines perpendicular to v_i and v_j . The angle of this double wedge is exactly the angle between v_i and v_j . Now, since v_i and v_j are unit vectors, we have $\langle v_i, v_j \rangle = \cos(\tau)$, where $\tau = \angle v_i v_j$. Thus,

$$\mathbb{P}[\text{sign}(\langle v_i, \mathbf{r} \rangle) \neq \text{sign}(\langle v_j, \mathbf{r} \rangle)] = \frac{2\tau}{2\pi} = \frac{1}{\pi} \cdot \arccos(\langle v_i, v_j \rangle),$$

as claimed. ■

Theorem 26.1.2. *Let W be the random variable which is the weight of the cut generated by the algorithm. We have*

$$\mathbb{E}[W] = \frac{1}{\pi} \sum_{i < j} \omega_{ij} \arccos(\langle v_i, v_j \rangle).$$

Proof: Let X_{ij} be an indicator variable which is 1 if and only if the edge ij is in the cut. We have

$$\mathbb{E}[X_{ij}] = \mathbb{P}[\text{sign}(\langle v_i, \mathbf{r} \rangle) \neq \text{sign}(\langle v_j, \mathbf{r} \rangle)] = \frac{1}{\pi} \arccos(\langle v_i, v_j \rangle),$$

by [Lemma 26.1.1](#). Clearly, $W = \sum_{i < j} \omega_{ij} X_{ij}$, and by linearity of expectation, we have

$$\mathbb{E}[W] = \sum_{i < j} \omega_{ij} \mathbb{E}[X_{ij}] = \frac{1}{\pi} \sum_{i < j} \omega_{ij} \arccos(\langle v_i, v_j \rangle). \quad \blacksquare$$

Lemma 26.1.3. *For $-1 \leq y \leq 1$, we have $\frac{\arccos(y)}{\pi} \geq \alpha \cdot \frac{1}{2}(1 - y)$, where $\alpha = \min_{0 \leq \psi \leq \pi} \frac{2}{\pi} \frac{\psi}{1 - \cos(\psi)}$.*

Proof: Set $y = \cos(\psi)$. The inequality now becomes $\frac{\psi}{\pi} \geq \alpha \frac{1}{2}(1 - \cos \psi)$. Reorganizing, the inequality becomes $\frac{2}{\pi} \frac{\psi}{1 - \cos \psi} \geq \alpha$, which trivially holds by the definition of α . ■

Lemma 26.1.4. $\alpha > 0.87856$.

Proof: Using simple calculus, one can see that α achieves its value for $\psi = 2.331122\dots$, the nonzero root of $\cos \psi + \psi \sin \psi = 1$. ■

Theorem 26.1.5. *The above algorithm computes in expectation a cut with total weight $\alpha \cdot \text{Opt} \geq 0.87856 \text{Opt}$, where Opt is the weight of the maximal cut.*

Proof: Consider the optimal solution to (P) , and lets its value be $\gamma \geq \text{Opt}$. We have

$$\mathbb{E}[W] = \frac{1}{\pi} \sum_{i < j} \omega_{ij} \arccos(\langle v_i, v_j \rangle) \geq \sum_{i < j} \omega_{ij} \alpha \frac{1}{2}(1 - \langle v_i, v_j \rangle) = \alpha \gamma \geq \alpha \cdot \text{Opt},$$

by [Lemma 26.1.3](#). ■

26.2. Semi-definite programming

Let us define a variable $x_{ij} = \langle v_i, v_j \rangle$, and consider the n by n matrix M formed by those variables, where $x_{ii} = 1$ for $i = 1, \dots, n$. Let V be the matrix having v_1, \dots, v_n as its columns. Clearly, $M = V^T V$. In particular, this implies that for any non-zero vector $v \in \mathbb{R}^n$, we have $v^T M v = v^T A^T A v = (Av)^T (Av) \geq 0$. A matrix that has this property, is called **positive semidefinite**. Interestingly, any positive semidefinite matrix P can be represented as a product of a matrix with its transpose; namely, $P = B^T B$. Furthermore, given such a matrix P of size $n \times n$, we can compute B such that $P = B^T B$ in $O(n^3)$ time. This is known as **Cholesky decomposition**.

Observe, that if a semidefinite matrix $P = B^T B$ has a diagonal where all the entries are one, then B has columns which are unit vectors. Thus, if we solve (P) and get back a semi-definite matrix, then we can recover the vectors realizing the solution, and use them for the rounding.

In particular, (P) can now be restated as

$$\begin{aligned}
 (SD) \quad & \max && \frac{1}{2} \sum_{i < j} \omega_{ij} (1 - x_{ij}) \\
 & \text{subject to:} && x_{ii} = 1 && \text{for } i = 1, \dots, n \\
 & && (x_{ij})_{i=1, \dots, n, j=1, \dots, n} \text{ is a positive semi-definite matrix.}
 \end{aligned}$$

We are trying to find the optimal value of a linear function over a set which is the intersection of linear constraints and the set of positive semi-definite matrices.

Lemma 26.2.1. *Let \mathcal{U} be the set of $n \times n$ positive semidefinite matrices. The set \mathcal{U} is convex.*

Proof: Consider $A, B \in \mathcal{U}$, and observe that for any $t \in [0, 1]$, and vector $v \in \mathbb{R}^n$, we have:

$$v^T (tA + (1-t)B)v = v^T (tAv + (1-t)Bv) = tv^T Av + (1-t)v^T Bv \geq 0 + 0 \geq 0,$$

since A and B are positive semidefinite. ■

Positive semidefinite matrices corresponds to ellipsoids. Indeed, consider the set $x^T A x = 1$: the set of vectors that solve this equation is an ellipsoid. Also, the eigenvalues of a positive semidefinite matrix are all non-negative real numbers. Thus, given a matrix, we can in polynomial time decide if it is positive semidefinite or not (by computing the eigenvalues of the matrix).

Thus, we are trying to optimize a linear function over a convex domain. There is by now machinery to approximately solve those problems to within any additive error in polynomial time. This is done by using the interior point method, or the ellipsoid method. See [BV04, GLS93] for more details. The key ingredient that is required to make these methods work, is the ability to decide in polynomial time, given a solution, whether its feasible or not. As demonstrated above, this can be done in polynomial time.

26.3. Bibliographical Notes

The approximation algorithm presented is from the work of Goemans and Williamson [GW95]. Håstad [Hås01b] showed that MAX CUT can not be approximated within a factor of $16/17 \approx 0.941176$. Recently, Khot *et al.* [KKMO04] showed a hardness result that matches the constant of Goemans and Williamson (i.e., one can not approximate it better than α , unless $P = NP$). However, this relies on two conjectures, the first one is the “Unique Games Conjecture”, and the other one is “Majority is Stablest”. The “Majority is Stablest” conjecture was recently proved by Mossel *et al.* [MOO05]. However, it is not clear if the “Unique Games Conjecture” is true, see the discussion in [KKMO04].

The work of Goemans and Williamson was very influential and spurred wide research on using SDP for approximation algorithms. For an extension of the MAX CUT problem where negative weights are allowed and relevant references, see the work by Alon and Naor [AN04].

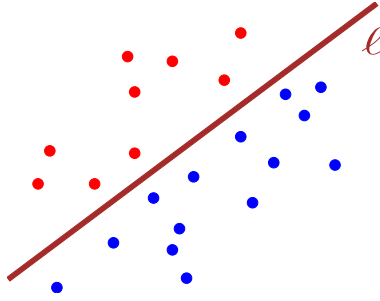


Figure 27.1: Linear separable red and blue point sets.

Chapter 27

The Perceptron Algorithm

27.1. The perceptron algorithm

Assume, that we are given examples, say a database of cars, and you would like to determine which cars are sport cars, and which are regular cars. Each car record, can be interpreted as a point in high dimensions. For example, a sport car with 4 doors, manufactured in 1997, by Quaky (with manufacturer ID 6) will be represented by the point $(4, 1997, 6)$, marked as a sport car. A tractor made by General Mess (manufacturer ID 3) in 1998, would be stored as $(0, 1997, 3)$ and would be labeled as not a sport car.

Naturally, in a real database there might be hundreds of attributes in each record, for engine size, weight, price, maximum speed, cruising speed, etc, etc, etc.

We would like to automate this *classification* process, so that tagging the records whether they correspond to race cars be done automatically without a specialist being involved. We would like to have a learning algorithm, such that given several classified examples, develop its own conjecture about what is the rule of the classification, and we can use it for classifying new data.

That is, there are two stages for *learning*: *training* and *classifying*. More formally, we are trying to learn a function

$$f : \mathbb{R}^d \rightarrow \{-1, 1\}.$$

The challenge is, of course, that f might have infinite complexity – informally, think about a label assigned to items where the label is completely random – there is nothing to learn except knowing the label for all possible items.

This situation is extremely rare in the real world, and we would limit ourselves to a set of functions that can be easily described. For example, consider a set of red and blue points that are linearly separable, as demonstrated in Figure 27.1. That is, we are trying to learn a line ℓ that separates the red points from the blue points.

The natural question is now, given the red and blue points, how to compute the line ℓ ? Well, a line or more generally a plane (or even a hyperplane) is the zero set of a linear function, that has the form

$$\forall x \in \mathbb{R}^d \quad f(x) = \langle a, x \rangle + b,$$

where $a = (a_1, \dots, a_d), b = (b_1, \dots, b_d) \in \mathbb{R}^d$, and $\langle a, x \rangle = \sum_i a_i x_i$ is the *dot product* of a and x . The classification itself, would be done by computing the sign of $f(x)$; that is $\text{sign}(f(x))$. Specifically, if $\text{sign}(f(x))$ is negative, it is outside the class, if it is positive it is inside.

```

Algorithm perceptron( $S$ : a set of  $l$  examples)
 $\mathbf{w}_0 \leftarrow 0, k \leftarrow 0$ 
 $R = \max_{(\mathbf{x}, y) \in S} \|\mathbf{x}\|$ .
repeat
  for  $(\mathbf{x}, y) \in S$  do
    if  $\text{sign}(\langle \mathbf{w}_k, \mathbf{x} \rangle) \neq y$  then
       $\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + y * \mathbf{x}$ 
       $k \leftarrow k + 1$ 
until no mistakes are made in the classification
return  $\mathbf{w}_k$  and  $k$ 

```

Figure 27.2: The **perceptron** algorithm.

A set of training examples is a set of pairs

$$S = \{(x_1, y_1), \dots, (x_n, y_n)\},$$

where $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$, for $i = 1, \dots, n$.

A **linear classifier** h is a pair (w, b) where $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$. The classification of $x \in \mathbb{R}^d$ is $\text{sign}(\langle w, x \rangle + b)$. For a **labeled** example (x, y) , h classifies (x, y) correctly if $\text{sign}(\langle w, x \rangle + b) = y$.

Assume that the underlying label we are trying to learn has a linear classifier (this is a problematic assumption – more on this later), and you are given “enough” examples (i.e., n). How to compute the linear classifier for these examples?

One natural option is to use linear programming. Indeed, we are looking for (\mathbf{w}, b) , such that for an (\mathbf{x}_i, y_i) we have $\text{sign}(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) = y_i$, which is

$$\begin{aligned} & \langle \mathbf{w}, \mathbf{x}_i \rangle + b \geq 0 && \text{if } y_i = 1, \\ \text{and} & \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq 0 && \text{if } y_i = -1. \end{aligned}$$

Or equivalently, let $\mathbf{x}_i = (x_i^1, \dots, x_i^d) \in \mathbb{R}^d$, for $i = 1, \dots, m$, and let $\mathbf{w} = (w^1, \dots, w^d)$, then we get the linear constraint

$$\begin{aligned} & \sum_{k=1}^d w^k x_i^k + b \geq 0 && \text{if } y_i = 1, \\ \text{and} & \sum_{k=1}^d w^k x_i^k + b \leq 0 && \text{if } y_i = -1. \end{aligned}$$

Thus, we get a set of linear constraints, one for each training example, and we need to solve the resulting linear program.

The main stumbling block is that linear programming is very sensitive to noise. Namely, if we have points that are misclassified, we would not find a solution, because no solution satisfying all of the constraints exists. Instead, we are going to use an iterative algorithm that converges to the optimal solution if it exists, see **Figure 27.2**.

Why does the **perceptron** algorithm converges to the right solution? Well, assume that we made a mistake on a sample (\mathbf{x}, y) and $y = 1$. Then, $\langle \mathbf{w}_k, \mathbf{x} \rangle < 0$, and

$$\langle \mathbf{w}_{k+1}, \mathbf{x} \rangle = \langle \mathbf{w}_k + y * \mathbf{x}, \mathbf{x} \rangle = \langle \mathbf{w}_k, \mathbf{x} \rangle + y \langle \mathbf{x}, \mathbf{x} \rangle = \langle \mathbf{w}_k, \mathbf{x} \rangle + y \|\mathbf{x}\|^2 > \langle \mathbf{w}_k, \mathbf{x} \rangle.$$

Namely, we are “walking” in the right direction, in the sense that the new value assigned to \mathbf{x} by \mathbf{w}_{k+1} is larger (“more positive”) than the old value assigned to \mathbf{x} by \mathbf{w}_k .

Theorem 27.1.1. Let S be a training set of examples, and let $R = \max_{(x,y) \in S} \|x\|$. Suppose that there exists a vector w_{opt} such that $\|w_{opt}\| = 1$, and a number $\gamma > 0$, such that

$$y \langle w_{opt}, x \rangle \geq \gamma \quad \forall (x, y) \in S.$$

Then, the number of mistakes made by the online *perceptron* algorithm on S is at most

$$\left(\frac{R}{\gamma} \right)^2.$$

Proof: Intuitively, the *perceptron* algorithm weight vector converges to w_{opt} . To see that, let us define the distance between w_{opt} and the weight vector in the k th update:

$$\alpha_k = \left\| w_k - \frac{R^2}{\gamma} w_{opt} \right\|^2.$$

We next quantify the change between α_k and α_{k+1} (the example being misclassified is (x, y)):

$$\begin{aligned} \alpha_{k+1} &= \left\| w_{k+1} - \frac{R^2}{\gamma} w_{opt} \right\|^2 \\ &= \left\| w_k + y\mathbf{x} - \frac{R^2}{\gamma} w_{opt} \right\|^2 \\ &= \left\| \left(w_k - \frac{R^2}{\gamma} w_{opt} \right) + y\mathbf{x} \right\|^2 \\ &= \left\langle \left(w_k - \frac{R^2}{\gamma} w_{opt} \right) + y\mathbf{x}, \left(w_k - \frac{R^2}{\gamma} w_{opt} \right) + y\mathbf{x} \right\rangle. \end{aligned}$$

Expanding this we get:

$$\begin{aligned} \alpha_{k+1} &= \left\langle \left(w_k - \frac{R^2}{\gamma} w_{opt} \right), \left(w_k - \frac{R^2}{\gamma} w_{opt} \right) \right\rangle + 2y \left\langle \left(w_k - \frac{R^2}{\gamma} w_{opt} \right), \mathbf{x} \right\rangle + \langle \mathbf{x}, \mathbf{x} \rangle \\ &= \alpha_k + 2y \left\langle \left(w_k - \frac{R^2}{\gamma} w_{opt} \right), \mathbf{x} \right\rangle + \|x\|^2. \end{aligned}$$

As (\mathbf{x}, y) is misclassified, it must be that $\text{sign}(\langle w_k, \mathbf{x} \rangle) \neq y$, which implies that $\text{sign}(y \langle w_k, \mathbf{x} \rangle) = -1$; that is $y \langle w_k, \mathbf{x} \rangle < 0$. Now, since $\|x\| \leq R$, we have

$$\begin{aligned} \alpha_{k+1} &\leq \alpha_k + R^2 + 2y \langle w_k, \mathbf{x} \rangle - 2y \left\langle \frac{R^2}{\gamma} w_{opt}, \mathbf{x} \right\rangle \\ &\leq \alpha_k + R^2 + \quad \quad \quad -2 \frac{R^2}{\gamma} y \langle w_{opt}, \mathbf{x} \rangle. \end{aligned}$$

Next, since $y \langle w_{opt}, \mathbf{x} \rangle \geq \gamma$ for $\forall (x, y) \in S$, we have that

$$\begin{aligned} \alpha_{k+1} &\leq \alpha_k + R^2 - 2 \frac{R^2}{\gamma} \gamma \\ &\leq \alpha_k + R^2 - 2R^2 \\ &\leq \alpha_k - R^2. \end{aligned}$$

We have: $\alpha_{k+1} \leq \alpha_k - R^2$, and

$$\alpha_0 = \left\| 0 - \frac{R^2}{\gamma} w_{opt} \right\|^2 = \frac{R^4}{\gamma^2} \|w_{opt}\|^2 = \frac{R^4}{\gamma^2}.$$

Finally, observe that $\alpha_i \geq 0$ for all i . Thus, what is the maximum number of classification errors the algorithm can make?

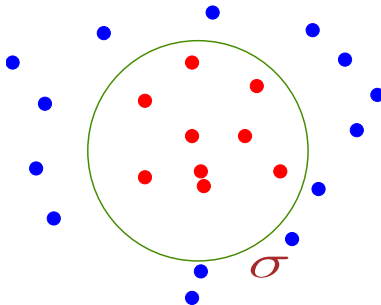
$$\left(\frac{R^2}{\gamma^2}\right).$$

■

It is important to observe that any linear program can be written as the problem of separating red points from blue points. As such, the **perceptron** algorithm can be used to solve linear programs.

27.2. Learning A Circle

Given a set of red points, and blue points in the plane, we want to learn a circle that contains all the red points, and does not contain the blue points.



How to compute the circle σ ?

It turns out we need a simple but very clever trick. For every point $(x, y) \in P$ map it to the point $(x, y, x^2 + y^2)$. Let $z(P) = \{(x, y, x^2 + y^2) \mid (x, y) \in P\}$ be the resulting point set.

Theorem 27.2.1. *Two sets of points R and B are separable by a circle in two dimensions, if and only if $z(R)$ and $z(B)$ are separable by a plane in three dimensions.*

Proof: Let $\sigma \equiv (x - a)^2 + (y - b)^2 = r^2$ be the circle containing all the points of R and having all the points of B outside. Clearly, $(x - a)^2 + (y - b)^2 \leq r^2$ for all the points of R . Equivalently

$$-2ax - 2by + (x^2 + y^2) \leq r^2 - a^2 - b^2.$$

Setting $z = x^2 + y^2$ we get that

$$h \equiv -2ax - 2by + z - r^2 + a^2 + b^2 \leq 0.$$

Namely, $p \in \sigma$ if and only if $h(z(p)) \leq 0$. We just proved that if the point set is separable by a circle, then the lifted point set $z(R)$ and $z(B)$ are separable by a plane.

As for the other direction, assume that $z(R)$ and $z(B)$ are separable in $3d$ and let

$$h \equiv ax + by + cz + d = 0$$

be the separating plane, such that all the point of $z(R)$ evaluate to a negative number by h . Namely, for $(x, y, x^2 + y^2) \in z(R)$ we have $ax + by + c(x^2 + y^2) + d \leq 0$

and similarly, for $(x, y, x^2 + y^2) \in z(B)$ we have $ax + by + c(x^2 + y^2) + d \geq 0$.

Let $U(h) = \{(x, y) \mid h((x, y, x^2 + y^2)) \leq 0\}$. Clearly, if $U(h)$ is a circle, then this implies that $R \subset U(h)$ and $B \cap U(h) = \emptyset$, as required.

So, $U(h)$ are all the points in the plane, such that

$$ax + by + c(x^2 + y^2) \leq -d.$$

Equivalently

$$\left(x^2 + \frac{a}{c}x\right) + \left(y^2 + \frac{b}{c}y\right) \leq -\frac{d}{c}$$

$$\left(x + \frac{a}{2c}\right)^2 + \left(y + \frac{b}{2c}\right)^2 \leq \frac{a^2 + b^2}{4c^2} - \frac{d}{c}$$

but this defines the interior of a circle in the plane, as claimed. ■

This example shows that linear separability is a powerful technique that can be used to learn complicated concepts that are considerably more complicated than just hyperplane separation. This lifting technique showed above is the *kernel technique* or *linearization*.

27.3. A Little Bit On VC Dimension

As we mentioned, inherent to the learning algorithms, is the concept of how complex is the function we are trying to learn. VC-dimension is one of the most natural ways of capturing this notion. (VC = Vapnik, Chervonenkis, 1971).

A matter of expressivity. What is harder to learn:

1. A rectangle in the plane.
2. A halfplane.
3. A convex polygon with k sides.

Let $X = \{p_1, p_2, \dots, p_m\}$ be a set of m points in the plane, and let R be the set of all halfplanes.

A half-plane r defines a binary vector

$$r(X) = (b_1, \dots, b_m)$$

where $b_i = 1$ if and only if p_i is inside r .

Let

$$U(X, R) = \{r(X) \mid r \in R\}.$$

A set X of m elements is *shattered* by R if

$$|U(X, R)| = 2^m.$$

What does this mean?

The VC-dimension of a set of ranges R is the size of the largest set that it can shatter.

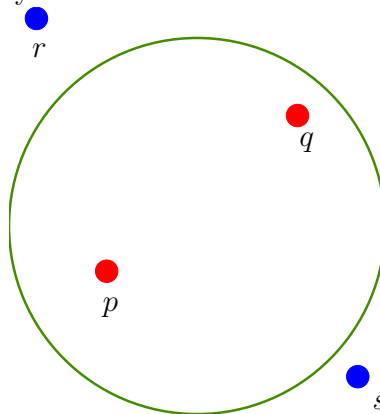
27.3.1. Examples

What is the VC dimensions of circles in the plane?

Namely, X is set of n points in the plane, and R is a set of all circles.

$X = \{p, q, r, s\}$

What subsets of X can we generate by circle?



$\{\}, \{r\}, \{p\}, \{q\}, \{s\}, \{p, s\}, \{p, q\}, \{p, r\}, \{r, q\}, \{q, s\}$ and $\{r, p, q\}, \{p, r, s\}, \{p, s, q\}, \{s, q, r\}$ and $\{r, p, q, s\}$

We got only 15 sets. There is one set which is not there. Which one?

The VC dimension of circles in the plane is 3.

Lemma 27.3.1 (Sauer Lemma). *If R has VC dimension d then $|U(X, R)| = O(m^d)$, where m is the size of X .*

Part VIII

Compression, entropy, and randomness

Chapter 28

Huffman Coding

28.1. Huffman coding

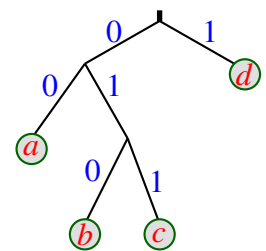
(This portion of the class notes is based on Jeff Erickson class notes.)

A *binary code* assigns a string of 0s and 1s to each character in the alphabet. A code assigns for each symbol in the input a codeword over some other alphabet. Such a coding is necessary, for example, for transmitting messages over a wire, where you can send only 0 or 1 on the wire (i.e., for example, consider the good old telegraph and Morse code). The receiver gets a binary stream of bits and needs to decode the message sent. A prefix code, is a code where one can decipher the message, a character by character, by reading a prefix of the input binary string, matching it to a code word (i.e., string), and continuing to decipher the rest of the stream. Such a code is a *prefix code*.

A binary code (or a prefix code) is *prefix-free* if no code is a prefix of any other. ASCII and Unicode's UTF-8 are both prefix-free binary codes. Morse code is a binary code (and also a prefix code), but it is not prefix-free; for example, the code for S (\dots) includes the code for E (\cdot) as a prefix. (Hopefully the receiver knows that when it gets \dots that it is extremely unlikely that this should be interpreted as EEE, but rather S.

Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves. The code word for any symbol is given by the path from the root to the corresponding leaf; 0 for left, 1 for right. The length of a codeword for a symbol is the depth of the corresponding leaf. Such trees are usually referred to as *prefix trees* or *code trees*.

The beauty of prefix trees (and thus of prefix codes) is that decoding is easy. As a concrete example, consider the tree on the right. Given a string '010100', we can traverse down the tree from the root, going left if we get a '0' and right if we get '1'. Whenever we get to a leaf, we output the character output in the leaf, and we jump back to the root for the next character we are about to read. For the example '010100', after reading '010' our traversal in the tree leads us to the leaf marked with 'b', we jump back to the root and read the next input digit, which is '1', and this leads us to the leaf marked with 'd', which we output, and jump back to the root. Finally, '00' leads us to the leaf marked by 'a', which the algorithm outputs. Thus, the binary string '010100' encodes the string "bda".



newline	16,492	'0'	20	'A'	48,165	'N'	42,380
space	130,376	'1'	61	'B'	8,414	'O'	46,499
'!	955	'2'	10	'C'	13,896	'P'	9,957
'"	5,681	'3'	12	'D'	28,041	'Q'	667
'\$'	2	'4'	10	'E'	74,809	'R'	37,187
'%'	1	'5'	14	'F'	13,559	'S'	37,575
'"	1,174	'6'	11	'G'	12,530	'T'	54,024
'('	151	'7'	13	'H'	38,961	'U'	16,726
')'	151	'8'	13	'I'	41,005	'V'	5,199
'*'	70	'9'	14	'J'	710	'W'	14,113
','	13,276	':'	267	'K'	4,782	'X'	724
'_'	2,430	','	1,108	'L'	22,030	'Y'	12,177
'.'	6,769	'?'	913	'M'	15,298	'Z'	215

'_'	182
'"'	93
'@'	2
'/'	26

Figure 28.1: Frequency of characters in the book “A tale of two cities” by Dickens. For the sake of brevity, small letters were counted together with capital letters.

char	frequency	code
'A'	48165	1110
'B'	8414	101000
'C'	13896	00100
'D'	28041	0011
'E'	74809	011
'F'	13559	111111
'G'	12530	111110
'H'	38961	1001

char	frequency	code
'I'	41005	1011
'J'	710	1111011010
'K'	4782	11110111
'L'	22030	10101
'M'	15298	01000
'N'	42380	1100
'O'	46499	1101
'P'	9957	101001
'Q'	667	1111011001

char	frequency	code
'R'	37187	0101
'S'	37575	1000
'T'	54024	000
'U'	16726	01001
'V'	5199	1111010
'W'	14113	00101
'X'	724	1111011011
'Y'	12177	111100
'Z'	215	1111011000

Figure 28.2: The resulting prefix code for the frequencies of Figure 28.1. Here, for the sake of simplicity of exposition, the code was constructed only for the A—Z characters.

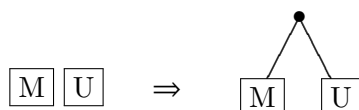
Suppose we want to encode messages in an n -character alphabet so that the encoded message is as short as possible. Specifically, given an array frequency counts $f[1 \dots n]$, we want to compute a prefix-free binary code that minimizes the total encoded length of the message. That is we would like to compute a tree \mathcal{T} that minimizes

$$\text{cost}(\mathcal{T}) = \sum_{i=1}^n f[i] * \text{len}(\text{code}(i)), \quad (28.1)$$

where $\text{code}(i)$ is the binary string encoding the i th character and $\text{len}(s)$ is the length (in bits) of the binary string s .

As a concrete example, consider Figure 28.1, which shows the frequency of characters in the book “A tale of two cities”, which we would like to encode. Consider the characters ‘E’ and ‘Q’. The first appears $> 74,000$ times in the text, and other appears only 667 times in the text. Clearly, it would be logical to give ‘E’, the most frequent letter in English, a very short prefix code, and a very long (as far as number of bits) code to ‘Q’.

A nice property of this problem is that given two trees for some parts of the alphabet, we can easily put them together into a larger tree by just creating a new node and hanging the trees from this common node. For example, putting two characters together, we have the following.



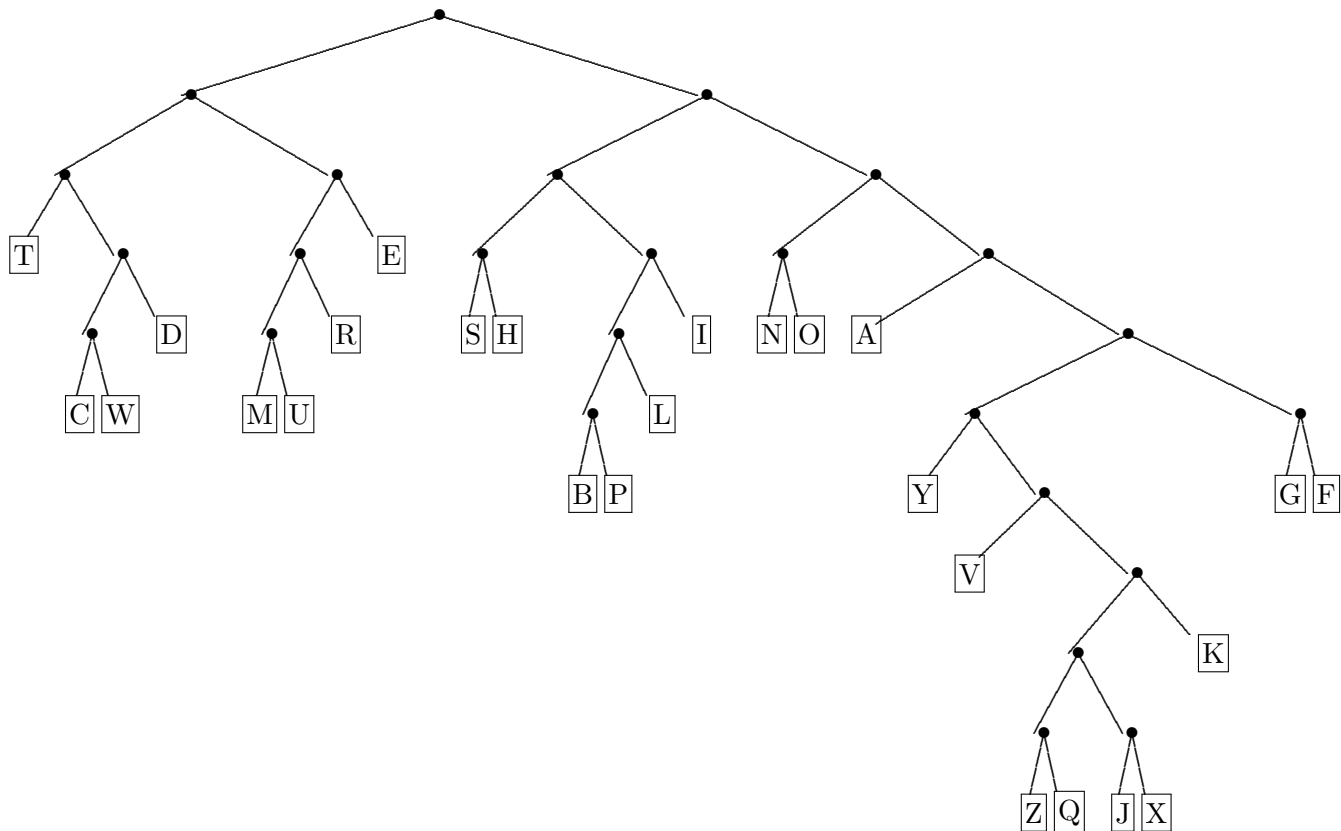
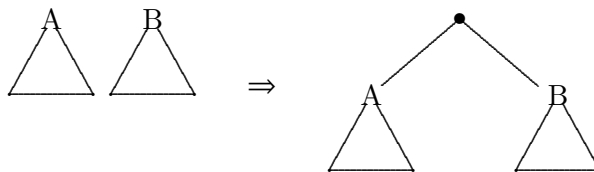


Figure 28.3: The Huffman tree generating the code of Figure 28.2.

Similarly, we can put together two subtrees.



28.1.1. The algorithm to build Huffman's code

This suggests a simple algorithm that takes the two least frequent characters in the current frequency table, merge them into a tree, and put the merged tree back into the table (instead of the two old trees). The algorithm stops when there is a single tree. The intuition is that infrequent characters would participate in a large number of merges, and as such would be low in the tree – they would be assigned a long code word.

This algorithm is due to David Huffman, who developed it in 1952. Shockingly, this code is the best one can do. Namely, the resulting code is *asymptotically* gives the best possible compression of the data (of course, one can do better compression in practice using additional properties of the data and careful hacking). This **Huffman coding** is used widely and is the basic building block used by numerous other compression algorithms.

To see how such a resulting tree (and the associated code) looks like, see Figure 28.2 and Figure 28.3.

28.1.2. Analysis

Lemma 28.1.1. *Let \mathcal{T} be an optimal code tree. Then \mathcal{T} is a full binary tree (i.e., every node of \mathcal{T} has either 0 or 2 children).*

In particular, if the height of \mathcal{T} is d , then there are leafs nodes of height d that are sibling.

Proof: If there is an internal node in \mathcal{T} that has one child, we can remove this node from \mathcal{T} , by connecting its only child directly with its parent. The resulting code tree is clearly a better compressor, in the sense of Eq. (28.1).

As for the second claim, consider a leaf u with maximum depth d in \mathcal{T} , and consider its parent $v = \bar{p}(u)$. The node v has two children, and they are both leaves (otherwise u would not be the deepest node in the tree), as claimed. ■

Lemma 28.1.2. *Let x and y be the two least frequent characters (breaking ties between equally frequent characters arbitrarily). There is an optimal code tree in which x and y are siblings.*

Proof: More precisely, there is an optimal code in which x and y are siblings and have the largest depth of any leaf. Indeed, let \mathcal{T} be an optimal code tree with depth d . The tree \mathcal{T} has at least two leaves at depth d that are siblings, by Lemma 28.1.1.

Now, suppose those two leaves are not x and y , but some other characters α and β . Let \mathcal{T}' be the code tree obtained by swapping x and α . The depth of x increases by some amount Δ , and the depth of α decreases by the same amount. Thus,

$$\text{cost}(\mathcal{T}') = \text{cost}(\mathcal{T}) - (f[\alpha] - f[x])\Delta.$$

By assumption, x is one of the two least frequent characters, but α is not, which implies that $f[\alpha] > f[x]$. Thus, swapping x and α does not increase the total cost of the code. Since \mathcal{T} was an optimal code tree, swapping x and α does not decrease the cost, either. Thus, \mathcal{T}' is also an optimal code tree (and incidentally, $f[\alpha]$ actually equals $f[x]$). Similarly, swapping y and β must give yet another optimal code tree. In this final optimal code tree, x and y are maximum-depth siblings, as required. ■

Theorem 28.1.3. *Huffman codes are optimal prefix-free binary codes.*

Proof: If the message has only one or two different characters, the theorem is trivial. Otherwise, let $f[1 \dots n]$ be the original input frequencies, where without loss of generality, $f[1]$ and $f[2]$ are the two smallest. To keep things simple, let $f[n+1] = f[1] + f[2]$. By the previous lemma, we know that some optimal code for $f[1 \dots n]$ has characters 1 and 2 as siblings. Let \mathcal{T}_{opt} be this optimal tree, and consider the tree formed by it by removing 1 and 2 as its leaves. We remain with a tree $\mathcal{T}'_{\text{opt}}$ that has as leaves the characters $3, \dots, n$ and a “special” character $n+1$ (which is the parent of 1 and 2 in \mathcal{T}_{opt}) that has frequency $f[n+1]$. Now, since $f[n+1] = f[1] + f[2]$, we have

$$\begin{aligned} \text{cost}(\mathcal{T}_{\text{opt}}) &= \sum_{i=1}^n f[i] \text{depth}_{\mathcal{T}_{\text{opt}}}(i) \\ &= \sum_{i=3}^{n+1} f[i] \text{depth}_{\mathcal{T}'_{\text{opt}}}(i) + f[1] \text{depth}_{\mathcal{T}'_{\text{opt}}}(1) + f[2] \text{depth}_{\mathcal{T}'_{\text{opt}}}(2) - f[n+1] \text{depth}_{\mathcal{T}'_{\text{opt}}}(n+1) \\ &= \text{cost}(\mathcal{T}'_{\text{opt}}) + (f[1] + f[2]) \text{depth}(\mathcal{T}_{\text{opt}}) - (f[1] + f[2]) (\text{depth}(\mathcal{T}_{\text{opt}}) - 1) \\ &= \text{cost}(\mathcal{T}'_{\text{opt}}) + f[1] + f[2]. \end{aligned} \tag{28.2}$$

This implies that minimizing the cost of \mathcal{T}_{opt} is equivalent to minimizing the cost of $\mathcal{T}'_{\text{opt}}$. In particular, $\mathcal{T}'_{\text{opt}}$ must be an optimal coding tree for $f[3 \dots n+1]$. Now, consider the Huffman tree \mathcal{T}'_H constructed for $f[3, \dots, n+1]$ and the overall Huffman tree \mathcal{T}_H constructed for $f[1, \dots, n]$. By the way the construction algorithm works, we have that \mathcal{T}'_H is formed by removing the leaves of 1 and 2 from \mathcal{T} . Now, by induction, we know that the Huffman tree generated for $f[3, \dots, n+1]$ is optimal; namely, $\text{cost}(\mathcal{T}'_{\text{opt}}) = \text{cost}(\mathcal{T}'_H)$. As such, arguing as above, we have

$$\text{cost}(\mathcal{T}_H) = \text{cost}(\mathcal{T}'_H) + f[1] + f[2] = \text{cost}(\mathcal{T}'_{\text{opt}}) + f[1] + f[2] = \text{cost}(\mathcal{T}_{\text{opt}}),$$

by Eq. (28.2). Namely, the Huffman tree has the same cost as the optimal tree. ■

28.1.3. What do we get

For the book “A tale of two cities” which is made out of 779,940 bytes, and using the above Huffman compression results in a compression to a file of size 439,688 bytes. A far cry from what `gzip` can do (301,295 bytes) or `bzip2` can do (220,156 bytes!), but still very impressive when you consider that the Huffman encoder can be easily written in a few hours of work.

(These numbers ignore the space required to store the code with the file. This is pretty small, and would not change the compression numbers stated above significantly.)

28.1.4. A formula for the average size of a code word

Assume that our input is made out of n characters, where the i th character is p_i fraction of the input (one can think about p_i as the probability of seeing the i th character, if we were to pick a random character from the input).

Now, we can use these probabilities instead of frequencies to build a Huffman tree. The natural question is what is the length of the codewords assigned to characters as a function of their probabilities?

In general this question does not have a trivial answer, but there is a simple elegant answer, if all the probabilities are power of 2.

Lemma 28.1.4. *Let $1, \dots, n$ be n symbols, such that the probability for the i th symbol is p_i , and furthermore, there is an integer $l_i \geq 0$, such that $p_i = 1/2^{l_i}$. Then, in the Huffman coding for this input, the code for i is of length l_i .*

Proof: The proof is by easy induction of the Huffman algorithm. Indeed, for $n = 2$ the claim trivially holds since there are only two characters with probability $1/2$. Otherwise, let i and j be the two characters with lowest probability. It must hold that $p_i = p_j$ (otherwise, $\sum_k p_k$ can not be equal to one). As such, Huffman’s merges this two letters, into a single “character” that have probability $2p_i$, which would have encoding of length $l_i - 1$, by induction (on the remaining $n - 1$ symbols). Now, the resulting tree encodes i and j by code words of length $(l_i - 1) + 1 = l_i$, as claimed. ■

In particular, we have that $l_i = \lg 1/p_i$. This implies that the average length of a code word is

$$\sum_i p_i \lg \frac{1}{p_i}.$$

If we consider X to be a random variable that takes a value i with probability p_i , then this formula is

$$\mathbb{H}(X) = \sum_i \mathbb{P}[X = i] \lg \frac{1}{\mathbb{P}[X = i]},$$

which is the *entropy* of X .

Chapter 29

Entropy, Randomness, and Information

“If only once - only once - no matter where, no matter before what audience - I could better the record of the great Rastelli and juggle with thirteen balls, instead of my usual twelve, I would feel that I had truly accomplished something

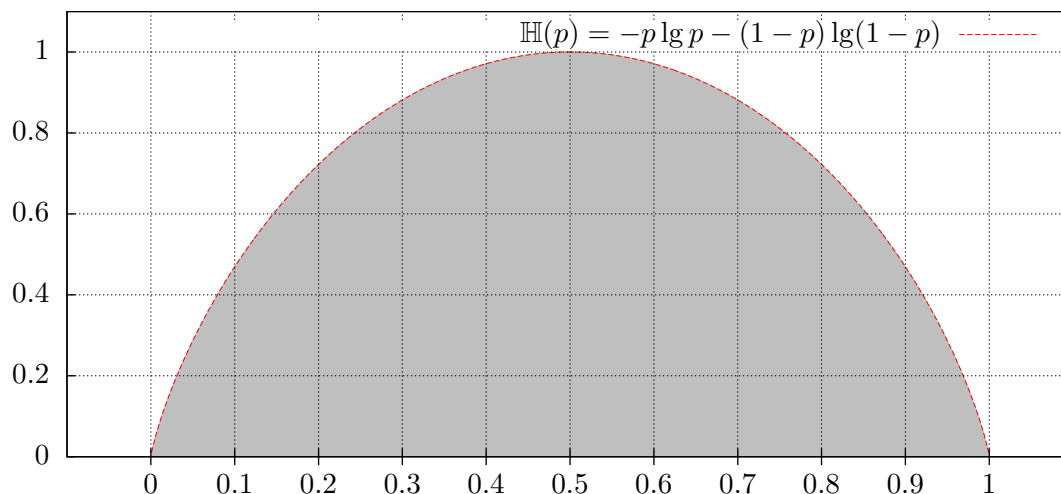


Figure 29.1: The binary entropy function.

for my country. But I am not getting any younger, and although I am still at the peak of my powers there are moments - why deny it? - when I begin to doubt - and there is a time limit on all of us.”
 – Romain Gary, The talent scout..

29.1. Entropy

Definition 29.1.1. The **entropy** in bits of a discrete random variable X is given by

$$\mathbb{H}(X) = - \sum_x \mathbb{P}[X = x] \lg \mathbb{P}[X = x].$$

Equivalently, $\mathbb{H}(X) = \mathbb{E} \left[\lg \frac{1}{\mathbb{P}[X]} \right]$.

The **binary entropy** function $\mathbb{H}(p)$ for a random binary variable that is 1 with probability p , is $\mathbb{H}(p) = -p \lg p - (1-p) \lg(1-p)$. We define $\mathbb{H}(0) = \mathbb{H}(1) = 0$. See [Figure 29.1](#).

The function $\mathbb{H}(p)$ is a concave symmetric around $1/2$ on the interval $[0, 1]$ and achieves its maximum at $1/2$. For a concrete example, consider $\mathbb{H}(3/4) \approx 0.8113$ and $\mathbb{H}(7/8) \approx 0.5436$. Namely, a coin that has $3/4$ probably to be heads have higher amount of “randomness” in it than a coin that has probability $7/8$ for heads.

We have $\mathbb{H}'(p) = -\lg p + \lg(1-p) = \lg \frac{1-p}{p}$ and $\mathbb{H}''(p) = \frac{p}{1-p} \cdot \left(-\frac{1}{p^2} \right) = -\frac{1}{p(1-p)}$. Thus, $\mathbb{H}''(p) \leq 0$, for all $p \in (0, 1)$, and the $\mathbb{H}(\cdot)$ is concave in this range. Also, $\mathbb{H}'(1/2) = 0$, which implies that $\mathbb{H}(1/2) = 1$ is a maximum of the binary entropy. Namely, a balanced coin has the largest amount of randomness in it.

Example 29.1.2. A random variable X that has probability $1/n$ to be i , for $i = 1, \dots, n$, has entropy $\mathbb{H}(X) = -\sum_{i=1}^n \frac{1}{n} \lg \frac{1}{n} = \lg n$.

Note, that the entropy is oblivious to the exact values that the random variable can have, and it is sensitive only to the probability distribution. Thus, a random variables that accepts $-1, +1$ with equal probability has the same entropy (i.e., 1) as a fair coin.

Lemma 29.1.3. *Let X and Y be two independent random variables, and let Z be the random variable (X, Y) . Then $\mathbb{H}(Z) = \mathbb{H}(X) + \mathbb{H}(Y)$.*

Proof: In the following, summation are over all possible values that the variables can have. By the independence of X and Y we have

$$\begin{aligned}
\mathbb{H}(Z) &= \sum_{x,y} \mathbb{P}[(X,Y) = (x,y)] \lg \frac{1}{\mathbb{P}[(X,Y) = (x,y)]} \\
&= \sum_{x,y} \mathbb{P}[X = x] \mathbb{P}[Y = y] \lg \frac{1}{\mathbb{P}[X = x] \mathbb{P}[Y = y]} \\
&= \sum_x \sum_y \mathbb{P}[X = x] \mathbb{P}[Y = y] \lg \frac{1}{\mathbb{P}[X = x]} \\
&\quad + \sum_y \sum_x \mathbb{P}[X = x] \mathbb{P}[Y = y] \lg \frac{1}{\mathbb{P}[Y = y]} \\
&= \sum_x \mathbb{P}[X = x] \lg \frac{1}{\mathbb{P}[X = x]} + \sum_y \mathbb{P}[Y = y] \lg \frac{1}{\mathbb{P}[Y = y]} \\
&= \mathbb{H}(X) + \mathbb{H}(Y). \quad \blacksquare
\end{aligned}$$

Lemma 29.1.4. *Suppose that nq is integer in the range $[0, n]$. Then $\frac{2^{n\mathbb{H}(q)}}{n+1} \leq \binom{n}{nq} \leq 2^{n\mathbb{H}(q)}$.*

Proof: This trivially holds if $q = 0$ or $q = 1$, so assume $0 < q < 1$. We know that

$$\binom{n}{nq} q^{nq} (1-q)^{n-nq} \leq (q + (1-q))^n = 1.$$

As such, since $q^{-nq} (1-q)^{-(1-q)n} = 2^{n(-q \lg q - (1-q) \lg(1-q))} = 2^{n\mathbb{H}(q)}$, we have

$$\binom{n}{nq} \leq q^{-nq} (1-q)^{-(1-q)n} = 2^{n\mathbb{H}(q)}.$$

As for the other direction, let $\mu(k) = \binom{n}{k} q^k (1-q)^{n-k}$. We claim that $\mu(nq) = \binom{n}{nq} q^{nq} (1-q)^{n-nq}$ is the largest term in $\sum_{k=0}^n \mu(k) = 1$. Indeed,

$$\Delta_k = \mu(k) - \mu(k+1) = \binom{n}{k} q^k (1-q)^{n-k} \left(1 - \frac{n-k}{k+1} \frac{q}{1-q} \right),$$

and the sign of this quantity is the sign of the last term, which is

$$\text{sign}(\Delta_k) = \text{sign} \left(1 - \frac{(n-k)q}{(k+1)(1-q)} \right) = \text{sign} \left(\frac{(k+1)(1-q) - (n-k)q}{(k+1)(1-q)} \right).$$

Now,

$$(k+1)(1-q) - (n-k)q = k+1 - kq - q - nq + kq = 1 + k - q - nq.$$

Namely, $\Delta_k \geq 0$ when $k \geq nq + q - 1$, and $\Delta_k < 0$ otherwise. Namely, $\mu(k) < \mu(k+1)$, for $k < nq$, and $\mu(k) \geq \mu(k+1)$ for $k \geq nq$. Namely, $\mu(nq)$ is the largest term in $\sum_{k=0}^n \mu(k) = 1$, and as such it is larger than the average. We have $\mu(nq) = \binom{n}{nq} q^{nq} (1-q)^{n-nq} \geq \frac{1}{n+1}$, which implies

$$\binom{n}{nq} \geq \frac{1}{n+1} q^{-nq} (1-q)^{-(n-nq)} = \frac{1}{n+1} 2^{n\mathbb{H}(q)}. \quad \blacksquare$$

Lemma 29.1.4 can be extended to handle non-integer values of q . This is straightforward, and we omit the easy but tedious details.

Corollary 29.1.5. *We have:*

$$(i) \quad q \in [0, 1/2] \Rightarrow \binom{n}{\lfloor nq \rfloor} \leq 2^{n\mathbb{H}(q)}.$$

$$(ii) \quad q \in [1/2, 1] \quad \binom{n}{\lceil nq \rceil} \leq 2^{n\mathbb{H}(q)}.$$

$$(iii) \quad q \in [1/2, 1] \Rightarrow \frac{2^{n\mathbb{H}(q)}}{n+1} \leq \binom{n}{\lfloor nq \rfloor}.$$

$$(iv) \quad q \in [0, 1/2] \Rightarrow \frac{2^{n\mathbb{H}(q)}}{n+1} \leq \binom{n}{\lceil nq \rceil}.$$

The bounds of [Lemma 29.1.4](#) and [Corollary 29.1.5](#) are loose but sufficient for our purposes. As a sanity check, consider the case when we generate a sequence of n bits using a coin with probability q for head, then by the Chernoff inequality, we will get roughly nq heads in this sequence. As such, the generated sequence Y belongs to $\binom{n}{nq} \approx 2^{n\mathbb{H}(q)}$ possible sequences that have similar probability. As such, $\mathbb{H}(Y) \approx \lg \binom{n}{nq} = n\mathbb{H}(q)$, by [Example 29.1.2](#), a fact that we already know from [Lemma 29.1.3](#).

29.1.1. Extracting randomness

Entropy can be interpreted as the amount of unbiased random coin flips can be extracted from a random variable.

Definition 29.1.6. An extraction function *Ext* takes as input the value of a random variable X and outputs a sequence of bits y , such that $\mathbb{P}[\text{Ext}(X) = y \mid |y| = k] = \frac{1}{2^k}$, whenever $\mathbb{P}[|y| = k] > 0$, where $|y|$ denotes the length of y .

As a concrete (easy) example, consider X to be a uniform random integer variable out of $0, \dots, 7$. All that *Ext*(X) has to do in this case, is to compute the binary representation of x . However, note that [Definition 29.1.6](#) is somewhat more subtle, as it requires that all extracted sequence of the same length would have the same probability.

Thus, for X a uniform random integer variable in the range $0, \dots, 11$, the function *Ext*(x) can output the binary representation for x if $0 \leq x \leq 7$. However, what do we do if x is between 8 and 11? The idea is to output the binary representation of $x - 8$ as a two bit number. Clearly, [Definition 29.1.6](#) holds for this extraction function, since $\mathbb{P}[\text{Ext}(X) = 00 \mid |\text{Ext}(X)| = 2] = \frac{1}{4}$, as required. This scheme can be of course extracted for any range.

The following is obvious, but we provide a proof anyway.

Lemma 29.1.7. *Let x/y be a fraction, such that $x/y < 1$. Then, for any i , we have $x/y < (x+i)/(y+i)$.*

Proof: We need to prove that $x(y+i) - (x+i)y < 0$. The left side is equal to $i(x-y)$, but since $y > x$ (as $x/y < 1$), this quantity is negative, as required. ■

Theorem 29.1.8. *Suppose that the value of a random variable X is chosen uniformly at random from the integers $\{0, \dots, m-1\}$. Then there is an extraction function for X that outputs on average at least $\lfloor \lg m \rfloor - 1 = \lfloor \mathbb{H}(X) \rfloor - 1$ independent and unbiased bits.*

Proof: We represent m as a sum of unique powers of 2, namely $m = \sum_i a_i 2^i$, where $a_i \in \{0, 1\}$. Thus, we decomposed $\{0, \dots, m-1\}$ into a disjoint union of blocks that have sizes which are distinct powers of 2. If a number falls inside such a block, we output its relative location in the block, using binary representation of the appropriate length (i.e., k if the block is of size 2^k). One can verify that this is an extraction function, fulfilling [Definition 29.1.6](#).

Now, observe that the claim holds trivially if m is a power of two. Thus, consider the case that m is not a power of 2. If X falls inside a block of size 2^k then the entropy is k . Thus, for the inductive proof, assume that are looking at the largest block in the decomposition, that is $m < 2^{k+1}$, and let $u = \lfloor \lg(m - 2^k) \rfloor < k$. There must be a block of size u in the decomposition of m . Namely, we have two blocks that we known in the

decomposition of m , of sizes 2^k and 2^u . Note, that these two blocks are the largest blocks in the decomposition of m . In particular, $2^k + 2 * 2^u > m$, implying that $2^{u+1} + 2^k - m > 0$.

Let Y be the random variable which is the number of bits output by the extractor algorithm.

By Lemma 29.1.7, since $\frac{m-2^k}{m} < 1$, we have

$$\frac{m-2^k}{m} \leq \frac{m-2^k + (2^{u+1} + 2^k - m)}{m + (2^{u+1} + 2^k - m)} = \frac{2^{u+1}}{2^{u+1} + 2^k}.$$

Thus, by induction (we assume the claim holds for all integers smaller than m), we have

$$\begin{aligned} \mathbb{E}[Y] &\geq \frac{2^k}{m}k + \frac{m-2^k}{m} \left(\underbrace{\lfloor \lg(m-2^k) \rfloor}_{u} - 1 \right) = \frac{2^k}{m}k + \frac{m-2^k}{m} \underbrace{(k-k+u-1)}_{=0} \\ &= k + \frac{m-2^k}{m}(u-k-1) \\ &\geq k + \frac{2^{u+1}}{2^{u+1} + 2^k}(u-k-1) = k - \frac{2^{u+1}}{2^{u+1} + 2^k}(1+k-u), \end{aligned}$$

since $u-k-1 \leq 0$ as $k > u$. If $u = k-1$, then $\mathbb{E}[Y] \geq k - \frac{1}{2} \cdot 2 = k-1$, as required. If $u = k-2$ then $\mathbb{E}[Y] \geq k - \frac{1}{3} \cdot 3 = k-1$. Finally, if $u < k-2$ then

$$\mathbb{E}[Y] \geq k - \frac{2^{u+1}}{2^k}(1+k-u) = k - \frac{k-u+1}{2^{k-u-1}} = k - \frac{2+(k-u-1)}{2^{k-u-1}} \geq k-1,$$

since $(2+i)/2^i \leq 1$ for $i \geq 2$. ■

Chapter 30

Even more on Entropy, Randomness, and Information

“It had been that way even before, when for years at a time he had not seen blue sky, and each second of those years could have been his last. But it did not benefit an Assaultman to think about death. Though on the other hand you had to think a lot about possible defeats. Gorbovsky had once said that death is worse than any defeat, even the most shattering. Defeat was always really only an accident, a setback which you could surmount. You had to surmount it. Only the dead couldn’t fight on.”

– – Defeat, Arkady and Boris Strugatsky.

30.1. Extracting randomness

30.1.1. Enumerating binary strings with j ones

Consider a binary string of length n with j ones. $S(n, j)$ denote the set of all such binary strings. There are $\binom{n}{j}$ such strings. For the following, we need an algorithm that given a string U of n bits with j ones, maps it into

a number in the range $0, \dots, \binom{n}{j} - 1$.

To this end, consider the full binary tree \mathcal{T} of height n . Each leaf, encodes a string of length n , and mark each leaf that encodes a string of $S(n, j)$. Consider a node v in the tree, that is of height k ; namely, the path π_v from the root of \mathcal{T} to v is of length k . Furthermore, assume there are m ones written on the path π_v . Clearly, any leaf in the subtree of v that is in $S(n, j)$ is created by selecting $j - m$ ones in the remaining $n - k$ positions. The number of possibilities to do so is $\binom{n-k}{j-m}$. Namely, given a node v in this tree \mathcal{T} , we can quickly compute the number of elements of $S(n, j)$ stored in this subtree.

As such, let traverse \mathcal{T} using a standard **DFS** algorithm, which would always first visit the ‘0’ child before the ‘1’ child, and use it to enumerate the marked leaves. Now, given a string x of S_j , we would like to compute what number would be assigned to by the above **DFS** procedure. The key observation is that calls made by the **DFS** on nodes that are not on the path, can be skipped by just computing directly how many marked leaves are there in the subtrees on this nodes (and this we can do using the above formula). As such, we can compute the number assigned to x in linear time.

The cool thing about this procedure, is that we do not need \mathcal{T} to carry it out. We can think about \mathcal{T} as being a virtual tree.

Formally, given a string x made out of n bits, with j ones, we can in $O(n)$ time map it to an integer in the range $0, \dots, \binom{n}{j} - 1$, and this mapping is one-to-one. Let **EnumBinomCoeffAlg** denote this procedure.

30.1.2. Extracting randomness

Theorem 30.1.1. *Consider a coin that comes up heads with probability $p > 1/2$. For any constant $\delta > 0$ and for n sufficiently large:*

- (A) *One can extract, from an input of a sequence of n flips, an output sequence of $(1 - \delta)n\mathbb{H}(p)$ (unbiased) independent random bits.*
- (B) *One can not extract more than $n\mathbb{H}(p)$ bits from such a sequence.*

Proof: There are $\binom{n}{j}$ input sequences with exactly j heads, and each has probability $p^j(1-p)^{n-j}$. We map this sequence to the corresponding number in the set $S_j = \{0, \dots, \binom{n}{j} - 1\}$. Note, that this, conditional distribution on j , is uniform on this set, and we can apply the extraction algorithm of **Theorem 29.1.8** to S_j . Let Z be the random variable which is the number of heads in the input, and let B be the number of random bits extracted. We have

$$\mathbb{E}[B] = \sum_{k=0}^n \mathbb{P}[Z = k] \mathbb{E}[B \mid Z = k],$$

and by **Theorem 29.1.8**, we have $\mathbb{E}[B \mid Z = k] \geq \left\lfloor \lg \binom{n}{k} \right\rfloor - 1$. Let $\varepsilon < p - 1/2$ be a constant to be determined shortly. For $n(p - \varepsilon) \leq k \leq n(p + \varepsilon)$, we have

$$\binom{n}{k} \geq \binom{n}{\lfloor n(p + \varepsilon) \rfloor} \geq \frac{2^{n\mathbb{H}(p + \varepsilon)}}{n + 1},$$

by **Corollary 29.1.5** (iii). We have

$$\begin{aligned} \mathbb{E}[B] &\geq \sum_{k=\lfloor n(p - \varepsilon) \rfloor}^{\lfloor n(p + \varepsilon) \rfloor} \mathbb{P}[Z = k] \mathbb{E}[B \mid Z = k] \geq \sum_{k=\lfloor n(p - \varepsilon) \rfloor}^{\lfloor n(p + \varepsilon) \rfloor} \mathbb{P}[Z = k] \left(\left\lfloor \lg \binom{n}{k} \right\rfloor - 1 \right) \\ &\geq \sum_{k=\lfloor n(p - \varepsilon) \rfloor}^{\lfloor n(p + \varepsilon) \rfloor} \mathbb{P}[Z = k] \left(\lg \frac{2^{n\mathbb{H}(p + \varepsilon)}}{n + 1} - 2 \right) \\ &= (n\mathbb{H}(p + \varepsilon) - \lg(n + 1) - 2) \mathbb{P}[|Z - np| \leq \varepsilon n] \\ &\geq (n\mathbb{H}(p + \varepsilon) - \lg(n + 1) - 2) \left(1 - 2 \exp\left(-\frac{n\varepsilon^2}{4p}\right) \right), \end{aligned}$$

since $\mu = \mathbb{E}[Z] = np$ and $\mathbb{P}\left[|Z - np| \geq \frac{\varepsilon}{p}pn\right] \leq 2 \exp\left(-\frac{np}{4}\left(\frac{\varepsilon}{p}\right)^2\right) = 2 \exp\left(-\frac{n\varepsilon^2}{4p}\right)$, by the Chernoff inequality. In particular, fix $\varepsilon > 0$, such that $\mathbb{H}(p + \varepsilon) > (1 - \delta/4)\mathbb{H}(p)$, and since p is fixed $n\mathbb{H}(p) = \Omega(n)$, in particular, for n sufficiently large, we have $-\lg(n+1) \geq -\frac{\delta}{10}n\mathbb{H}(p)$. Also, for n sufficiently large, we have $2 \exp\left(-\frac{n\varepsilon^2}{4p}\right) \leq \frac{\delta}{10}$. Putting it together, we have that for n large enough, we have

$$\mathbb{E}[B] \geq \left(1 - \frac{\delta}{4} - \frac{\delta}{10}\right)n\mathbb{H}(p)\left(1 - \frac{\delta}{10}\right) \geq (1 - \delta)n\mathbb{H}(p),$$

as claimed.

As for the upper bound, observe that if an input sequence x has probability $\mathbb{P}[X = x]$, then the output sequence $y = \mathbf{Ext}(x)$ has probability to be generated which is at least $\mathbb{P}[X = x]$. Now, all sequences of length $|y|$ have equal probability to be generated. Thus, we have the following (trivial) inequality

$$2^{|\mathbf{Ext}(x)|} \mathbb{P}[X = x] \leq 2^{|\mathbf{Ext}(x)|} \mathbb{P}[y = \mathbf{Ext}(x)] \leq 1,$$

implying that $|\mathbf{Ext}(x)| \leq \lg(1/\mathbb{P}[X = x])$. Thus,

$$\mathbb{E}[B] = \sum_x \mathbb{P}[X = x] |\mathbf{Ext}(x)| \leq \sum_x \mathbb{P}[X = x] \lg \frac{1}{\mathbb{P}[X = x]} = \mathbb{H}(X). \quad \blacksquare$$

30.2. Bibliographical Notes

The presentation here follows [MU05, Sec. 9.1-Sec 9.3].

Chapter 31

Shannon's theorem

“This has been a novel about some people who were punished entirely too much for what they did. They wanted to have a good time, but they were like children playing in the street; they could see one after another of them being killed - run over, maimed, destroyed - but they continued to play anyhow. We really all were very happy for a while, sitting around not toiling but just bullshitting and playing, but it was for such a terrible brief time, and then the punishment was beyond belief; even when we could see it, we could not believe it.”

– A Scanner Darkly, Philip K. Dick.

31.1. Coding: Shannon's Theorem

We are interested in the problem sending messages over a noisy channel. We will assume that the channel noise is behave “nicely”.

Definition 31.1.1. The input to a *binary symmetric channel* with parameter p is a sequence of bits x_1, x_2, \dots , and the output is a sequence of bits y_1, y_2, \dots , such that $\mathbb{P}[x_i = y_i] = 1 - p$ independently for each i .

Translation: Every bit transmitted have the same probability to be flipped by the channel. The question is how much information can we send on the channel with this level of noise. Naturally, a channel would have some capacity constraints (say, at most 4,000 bits per second can be sent on the channel), and the question is how to send the largest amount of information, so that the receiver can recover the original information sent.

Now, its important to realize that handling noise is unavoidable in the real world. Furthermore, there are tradeoffs between channel capacity and noise levels (i.e., we might be able to send considerably more bits on the channel but the probability of flipping [i.e., p] might be much larger). In designing a communication protocol over this channel, we need to figure out where is the optimal choice as far as the amount of information sent.

Definition 31.1.2. A (k, n) *encoding function* $\text{Enc} : \{0, 1\}^k \rightarrow \{0, 1\}^n$ takes as input a sequence of k bits and outputs a sequence of n bits. A (k, n) *decoding function* $\text{Dec} : \{0, 1\}^n \rightarrow \{0, 1\}^k$ takes as input a sequence of n bits and outputs a sequence of k bits.

Thus, the sender would use the encoding function to send its message, and the receiver would use the transmitted string (with the noise in it), to recover the original message. Thus, the sender starts with a message with k bits, it blow it up to n bits, using the encoding function (to get some robustness to noise), it send it over the (noisy) channel to the receiver. The receiver takes the given (noisy) message with n bits, and use the decoding function to recover the original k bits of the message.

Naturally, we would like k to be as large as possible (for a fixed n), so that we can send as much information as possible on the channel.

The following celebrated result of Shannon^① in 1948 states exactly how much information can be sent on such a channel.

Theorem 31.1.3 (Shannon’s theorem). *For a binary symmetric channel with parameter $p < 1/2$ and for any constants $\delta, \gamma > 0$, where n is sufficiently large, the following holds:*

- (i) *For an $k \leq n(1 - \mathbb{H}(p) - \delta)$ there exists (k, n) encoding and decoding functions such that the probability the receiver fails to obtain the correct message is at most γ for every possible k -bit input messages.*
- (ii) *There are no (k, n) encoding and decoding functions with $k \geq n(1 - \mathbb{H}(p) + \delta)$ such that the probability of decoding correctly is at least γ for a k -bit input message chosen uniformly at random.*

31.1.0.1. Intuition behind Shanon’s theorem

Let assume the senders has sent a string $S = s_1 s_2 \dots s_n$. The receiver got a string $T = t_1 t_2 \dots t_n$, where $p = \mathbb{P}[t_i \neq s_i]$, for all i . In particular, let U be the Hamming distance between S and T ; that is, $U = \sum_i [s_i \neq t_i]$. Under our assumptions $\mathbb{E}[U] = pn$, and U is a binomial variable. By Chernoff inequality, we know that $U \in [(1 - \delta)np, (1 + \delta)np]$ with high probability, where δ is some tiny constant. So lets assume this indeed happens. This means that T is in a ring R centered at S , with inner radius $(1 - \delta)np$ and outer radius $(1 + \delta)np$. This ring has

$$\sum_{i=(1-\delta)np}^{(1+\delta)np} \binom{n}{i} \leq 2 \binom{n}{(1+\delta)np} \leq \alpha = 2 \cdot 2^{n\mathbb{H}((1+\delta)p)}.$$

Let us pick as many rings as possible in the hypercube so that they are disjoint: R_1, \dots, R_κ . If somehow magically, every word in the hypercube would be covered, then we could use all the possible 2^n codewords, then the number of rings κ we would pick would be at least

$$\kappa \geq \frac{2^n}{|R|} \geq \frac{2^n}{2 \cdot 2^{n\mathbb{H}((1+\delta)p)}} \approx 2^{n(1-\mathbb{H}((1+\delta)p))}.$$

^①Claude Elwood Shannon (April 30, 1916 - February 24, 2001), an American electrical engineer and mathematician, has been called “the father of information theory”.

In particular, consider all possible strings of length k such that $2^k \leq \kappa$. We map the i th string in $\{0,1\}^k$ to the center C_i of the i th ring R_i . Assuming that when we send C_i , the receiver gets a string in R_i , then the decoding is easy - find the ring R_i containing the received string, take its center string C_i , and output the original string it was mapped to. Now, observe that

$$k = \lfloor \log \kappa \rfloor = n(1 - \mathbb{H}((1 + \delta)p)) \approx n(1 - \mathbb{H}(p)),$$

as desired.

31.1.0.2. What is wrong with the above?

The problem is that we can not find such a large set of disjoint rings. The reason is that when you pack rings (or balls) you are going to have wasted spaces around. To overcome this, we would allow rings to overlap somewhat. That makes things considerably more involved. The details follow.

31.2. Proof of Shannon's theorem

The proof is not hard, but requires some care, and we will break it into parts.

31.2.1. How to encode and decode efficiently

31.2.1.1. The scheme

Our scheme would be simple. Pick $k \leq n(1 - \mathbb{H}(p) - \delta)$. For any number $i = 0, \dots, \widehat{K} = 2^{k+1} - 1$, randomly generate a binary string Y_i made out of n bits, each one chosen independently and uniformly. Let $Y_0, \dots, Y_{\widehat{K}}$ denote these code words. Here, we have

$$\widehat{K} = 2^{n(1-\mathbb{H}(p)-\delta)}.$$

For each of these codewords we will compute the probability that if we send this codeword, the receiver would fail. Let X_0, \dots, X_K , where $K = 2^k - 1$, be the K codewords with the lowest probability to fail. We assign these words to the 2^k messages we need to encode in an arbitrary fashion.

The decoding of a message w is done by going over all the codewords, and finding all the codewords that are in (Hamming) distance in the range $[p(1 - \varepsilon)n, p(1 + \varepsilon)n]$ from w . If there is only a single word X_i with this property, we return i as the decoded word. Otherwise, if there are no such words or there is more than one word, the decoder stops and report an error.

31.2.1.2. The proof

Intuition. Let S_i be all the binary strings (of length n) such that if the receiver gets this word, it would decipher it to be i (here are still using the extended codeword $Y_0, \dots, Y_{\widehat{K}}$). Note, that if we remove some codewords from consideration, the set S_i just increases in size. Let W_i be the probability that X_i was sent, but it was not deciphered correctly. Formally, let r denote the received word. We have that

$$W_i = \sum_{r \notin S_i} \mathbb{P}[r \text{ received when } X_i \text{ was sent}].$$

To bound this quantity, let $\Delta(x, y)$ denote the Hamming distance between the binary strings x and y . Clearly, if x was sent the probability that y was received is

$$w(x, y) = p^{\Delta(x,y)}(1 - p)^{n-\Delta(x,y)}.$$

As such, we have

$$\mathbb{P}[r \text{ received when } X_i \text{ was sent}] = w(X_i, r).$$

Let $\overline{S_{i,r}}$ be an indicator variable which is 1 if $r \notin S_i$. We have that

$$W_i = \sum_{r \notin S_i} \mathbb{P}[r \text{ received when } X_i \text{ was sent}] = \sum_{r \notin S_i} w(X_i, r) = \sum_r \overline{S_{i,r}} w(X_i, r).$$

The value of W_i is a random variable of our choice of $Y_0, \dots, Y_{\widehat{K}}$. As such, its natural to ask what is the expected value of W_i .

Consider the ring

$$R(r) = \left\{ x \mid (1 - \varepsilon)np \leq \Delta(x, r) \leq (1 + \varepsilon)np \right\},$$

where $\varepsilon > 0$ is a small enough constant. Suppose, that the code word Y_i was sent, and r was received. The decoder return i if Y_i is the only codeword that falls inside $R(r)$.

Lemma 31.2.1. *Given that Y_i was sent, and r was received and furthermore $r \in R(Y_i)$, then the probability of the decoder failing, is*

$$\tau = \mathbb{P}\left[r \notin S_i \mid r \in R(Y_i)\right] \leq \frac{\gamma}{8},$$

where γ is the parameter of [Theorem 31.1.3](#).

Proof: The decoder fails here, only if $R(r)$ contains some other codeword Y_j ($j \neq i$) in it. As such,

$$\tau = \mathbb{P}\left[r \notin S_i \mid r \in R(Y_i)\right] \leq \mathbb{P}\left[Y_j \in R(r), \text{ for any } j \neq i\right] \leq \sum_{j \neq i} \mathbb{P}\left[Y_j \in R(r)\right].$$

Now, we remind the reader that the Y_j s are generated by picking each bit randomly and independently, with probability $1/2$. As such, we have

$$\mathbb{P}\left[Y_j \in R(r)\right] = \sum_{m=(1-\varepsilon)np}^{(1+\varepsilon)np} \frac{\binom{n}{m}}{2^n} \leq \frac{n}{2^n} \binom{n}{\lfloor (1+\varepsilon)np \rfloor},$$

since $(1 + \varepsilon)p < 1/2$ (for ε sufficiently small), and as such the last binomial coefficient in this summation is the largest. By [Corollary 29.1.5](#) (i), we have

$$\mathbb{P}\left[Y_j \in R(r)\right] \leq \frac{n}{2^n} \binom{n}{\lfloor (1+\varepsilon)np \rfloor} \leq \frac{n}{2^n} 2^{n\mathbb{H}((1+\varepsilon)p)} = n2^{n(\mathbb{H}((1+\varepsilon)p)-1)}.$$

As such, we have

$$\begin{aligned} \tau &= \mathbb{P}\left[r \notin S_i \mid r \in R(Y_i)\right] \leq \sum_{j \neq i} \mathbb{P}\left[Y_j \in R(r)\right] \\ &\leq \widehat{K} \mathbb{P}\left[Y_1 \in R(r)\right] \leq 2^{k+1} n2^{n(\mathbb{H}((1+\varepsilon)p)-1)} \\ &\leq n2^{n(1-\mathbb{H}(p)-\delta)+1} n2^{n(\mathbb{H}((1+\varepsilon)p)-1)} \leq n2^{n(\mathbb{H}((1+\varepsilon)p)-\mathbb{H}(p)-\delta)+1} \end{aligned}$$

since $k \leq n(1-\mathbb{H}(p)-\delta)$. Now, we choose ε to be a small enough constant, so that the quantity $\mathbb{H}((1 + \varepsilon)p) - \mathbb{H}(p) - \delta$ is equal to some (absolute) negative (constant), say $-\beta$, where $\beta > 0$. Then, $\tau \leq n2^{-\beta n+1}$, and choosing n large enough, we can make τ smaller than $\gamma/2$, as desired. As such, we just proved that

$$\tau = \mathbb{P}\left[r \notin S_i \mid r \in R(Y_i)\right] \leq \frac{\gamma}{2}. \quad \blacksquare$$

Lemma 31.2.2. *We have, that $\sum_{r \in R(Y_i)} w(Y_i, r) \leq \gamma/8$, where γ is the parameter of [Theorem 31.1.3](#).*

Proof: This quantity, is the probability of sending Y_i when every bit is flipped with probability p , and receiving a string r such that more than εpn bits were flipped. But this quantity can be bounded using the Chernoff inequality. Let $Z = \Delta(Y_i, r)$, and observe that $\mathbb{E}[Z] = pn$, and it is the sum of n independent indicator variables. As such

$$\sum_{r \notin R(Y_i)} w(Y_i, r) = \mathbb{P}[|Z - \mathbb{E}[Z]| > \varepsilon pn] \leq 2 \exp\left(-\frac{\varepsilon^2}{4} pn\right) < \frac{\gamma}{4},$$

since ε is a constant, and for n sufficiently large. ■

Lemma 31.2.3. *For any i , we have $\mu = \mathbb{E}[W_i] \leq \gamma/4$, where γ is the parameter of [Theorem 31.1.3](#).*

Proof: By linearity of expectations, we have

$$\begin{aligned} \mu &= \mathbb{E}[W_i] = \mathbb{E}\left[\sum_r \overline{S_{i,r}} w(Y_i, r)\right] = \sum_r \mathbb{E}\left[\overline{S_{i,r}} w(Y_i, r)\right] \\ &= \sum_r \mathbb{E}\left[\overline{S_{i,r}}\right] w(Y_i, r) = \sum_r \mathbb{P}[x \notin S_i] w(Y_i, r), \end{aligned}$$

since $\overline{S_{i,r}}$ is an indicator variable. Setting, $\tau = \mathbb{P}[r \notin S_i \mid r \in R(Y_i)]$ and since $\sum_r w(Y_i, r) = 1$, we get

$$\begin{aligned} \mu &= \sum_{r \in R(Y_i)} \mathbb{P}[x \notin S_i] w(Y_i, r) + \sum_{r \notin R(Y_i)} \mathbb{P}[x \notin S_i] w(Y_i, r) \\ &= \sum_{r \in R(Y_i)} \mathbb{P}[x \notin S_i \mid r \in R(Y_i)] w(Y_i, r) + \sum_{r \notin R(Y_i)} \mathbb{P}[x \notin S_i] w(Y_i, r) \\ &\leq \sum_{r \in R(Y_i)} \tau \cdot w(Y_i, r) + \sum_{r \notin R(Y_i)} w(Y_i, r) \leq \tau + \sum_{r \notin R(Y_i)} w(Y_i, r) \leq \frac{\gamma}{4} + \frac{\gamma}{4} = \frac{\gamma}{2}. \end{aligned}$$

Now, the receiver got r (when we sent Y_i), and it would miss encode it only if (i) r is outside of $R(Y_i)$, or $R(r)$ contains some other codeword Y_j ($j \neq i$) in it. As such,

$$\tau = \mathbb{P}[r \notin S_i \mid r \in R(Y_i)] \leq \mathbb{P}[Y_j \in R(r), \text{ for any } j \neq i] \leq \sum_{j \neq i} \mathbb{P}[Y_j \in R(r)].$$

Now, we remind the reader that the Y_j s are generated by picking each bit randomly and independently, with probability $1/2$. As such, we have

$$\mathbb{P}[Y_j \in R(r)] = \sum_{m=(1-\varepsilon)np}^{(1+\varepsilon)np} \frac{\binom{n}{m}}{2^n} \leq \frac{n}{2^n} \binom{n}{\lfloor (1+\varepsilon)np \rfloor},$$

since $(1+\varepsilon)p < 1/2$ (for ε sufficiently small), and as such the last binomial coefficient in this summation is the largest. By [Corollary 29.1.5](#) (i), we have

$$\mathbb{P}[Y_j \in R(r)] \leq \frac{n}{2^n} \binom{n}{\lfloor (1+\varepsilon)np \rfloor} \leq \frac{n}{2^n} 2^{n\mathbb{H}((1+\varepsilon)p)} = n 2^{n(\mathbb{H}((1+\varepsilon)p)-1)}.$$

As such, we have

$$\begin{aligned} \tau &= \mathbb{P}[r \notin S_i \mid r \in R(Y_i)] \leq \sum_{j \neq i} \mathbb{P}[Y_j \in R(r)] \leq \widehat{K} \mathbb{P}[Y_1 \in R(r)] \leq 2^{k+1} n 2^{n(\mathbb{H}((1+\varepsilon)p)-1)} \\ &\leq n 2^{n(1-\mathbb{H}(p)-\delta)+1+n(\mathbb{H}((1+\varepsilon)p)-1)} \leq n 2^{n(\mathbb{H}((1+\varepsilon)p)-\mathbb{H}(p)-\delta)+1} \end{aligned}$$

since $k \leq n(1 - \mathbb{H}(p) - \delta)$. Now, we choose ε to be a small enough constant, so that the quantity $\mathbb{H}((1 + \varepsilon)p) - \mathbb{H}(p) - \delta$ is negative (constant). Then, choosing n large enough, we can make τ smaller than $\gamma/2$, as desired. As such, we just proved that

$$\tau = \mathbb{P}\left[r \notin S_i \mid r \in R(Y_i)\right] \leq \frac{\gamma}{2}. \quad \blacksquare$$

In the following, we need the following trivial (but surprisingly deep) observation.

Observation 31.2.4. *For a random variable X , if $\mathbb{E}[X] \leq \psi$, then there exists an event in the probability space, that assigns X a value $\leq \mu$.*

This holds, since $\mathbb{E}[X]$ is just the average of X over the probability space. As such, there must be an event in the universe where the value of X does not exceed its average value.

The above observation is one of the main tools in a powerful technique to proving various claims in mathematics, known as the *probabilistic method*.

Lemma 31.2.5. *For the codewords X_0, \dots, X_K , the probability of failure in recovering them when sending them over the noisy channel is at most γ .*

Proof: We just proved that when using $Y_0, \dots, Y_{\widehat{K}}$, the expected probability of failure when sending Y_i , is $\mathbb{E}[W_i] \leq \gamma/2$, where $\widehat{K} = 2^{k+1} - 1$. As such, the expected total probability of failure is

$$\mathbb{E}\left[\sum_{i=0}^{\widehat{K}} W_i\right] = \sum_{i=0}^{\widehat{K}} \mathbb{E}[W_i] \leq \frac{\gamma}{2} 2^{k+1} = \gamma 2^k,$$

by [Lemma 31.2.3](#) (here we are using the facts that all the random variables we have are symmetric and behave in the same way). As such, by [Observation 31.2.4](#), there exist a choice of Y_i s, such that

$$\sum_{i=0}^{\widehat{K}} W_i \leq 2^k \gamma.$$

Now, we use a similar argument used in proving Markov's inequality. Indeed, the W_i are always positive, and it can not be that 2^k of them have value larger than γ , because in the summation, we will get that

$$\sum_{i=0}^{\widehat{K}} W_i > 2^k \gamma.$$

Which is a contradiction. As such, there are 2^k codewords with failure probability smaller than γ . We set our 2^k codeword to be these words. Since we picked only a subset of the codewords for our code, the probability of failure for each codeword shrinks, and is at most γ . ■

[Lemma 31.2.5](#) concludes the proof of the constructive part of Shannon's theorem.

31.2.2. Lower bound on the message size

We omit the proof of this part.

31.3. Bibliographical Notes

The presentation here follows [[MU05](#), Sec. 9.1-Sec 9.3].

Part IX

Miscellaneous topics II

Chapter 32

Matchings

I've never touched the hard stuff, only smoked grass a few times with the boys to be polite, and that's all, though ten is the age when the big guys come around teaching you all sorts of things. But happiness doesn't mean much to me, I still think life is better. Happiness is a mean son of a bitch and needs to be put in his place. Him and me aren't on the same team, and I'm cutting him dead. I've never gone in for politics, because somebody always stand to gain by it, but happiness is an even crummier racket, and their ought to be laws to put it out of business.

Momo, Emile Ajar

32.1. Definitions and basic properties

32.1.1. Definitions

Definition 32.1.1. For a graph $G = (V, E)$ a set $M \subseteq E$ of edges is a *matching* if no pair of edges of M has a common vertex.

Definition 32.1.2. A matching is *perfect* if it covers all the vertices of G . For a weight function w , which assigns real weight to the edges of G , a matching M is a *maximum weight matching*, if M is a matching and $w(M) = \sum_{e \in M} w(e)$ is maximum.

Definition 32.1.3. A matching M is a *maximal*, if M is a matching and it can not be made bigger by adding any edge.

Thus, a maximal matching is locally optimal, while a maximum matching is the global largest/heaviest possible matching.

Problem 32.1.4 (Maximum size matching). If there is no weight on the edges, we consider the weight of every edge to be one, and in this case, we are trying to compute a *maximum size matching* (aka *maximum cardinality matching*).

Problem 32.1.5 (Maximum weight matching). Given a graph G and a weight function on the edges, compute the maximum weight matching in G .

Remark 32.1.6. There is a simple way to compute a maximum size matching in a bipartite graph using network flow. Here we present an alternative algorithm that does not use network flow.

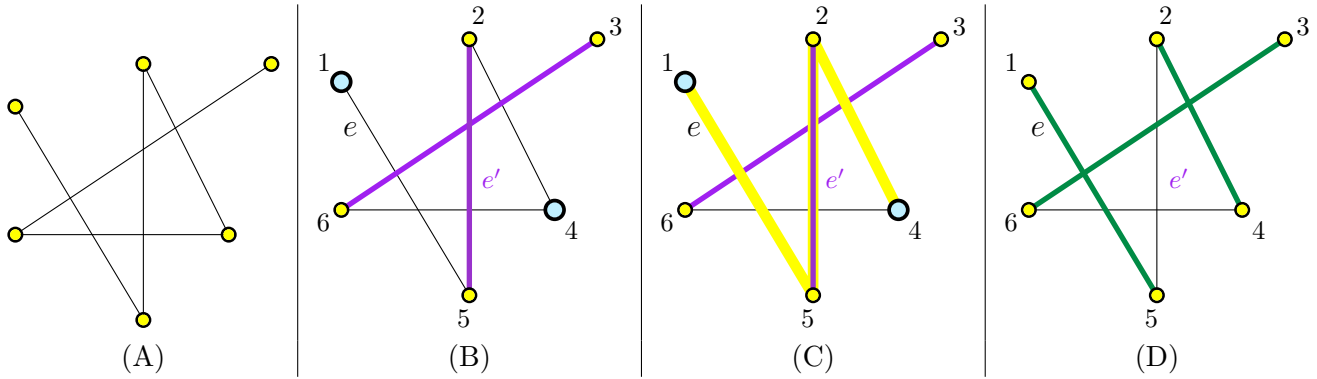


Figure 32.1: (A) The input graph. (B) A maximal matching in G . The edge e is free, and vertices 1 and 4 are free. (C) An alternating path. (D) The resulting matching from applying the augmenting path.

32.1.2. Matchings and alternating paths

Consider a matching M . An edge $e \in M$ is a **matching edge**. Naturally, Any edge $e' \in E(G) \setminus M$ is **free**. In particular, a vertex $v \in V(G)$ is **matched** if it is adjacent to an edge in M . Naturally, a vertex v' which is not matched is **free**.

An **alternating path** is a simple path that its edges are alternately matched and free. An **alternating cycle** is defined similarly. The **length** of a path/cycle is the number of edges in it.

Definition 32.1.7. A path $\pi = v_1v_2, \dots, v_{2k+2}$ is an **augmenting path** for a matching M in a graph G if

- (i) π is simple,
- (ii) for all i , $e_i = v_iv_{i+1} \in E(G)$,
- (iii) v_1 and v_{2k+1} are free vertices for M ,
- (iv) $e_1, e_3, \dots, e_{2k+1} \notin M$, and
- (v) $e_2, e_4, \dots, e_{2k} \in M$.

An augmenting path is an alternating path that starts and end with a free edge, and the two endpoints of the path are also free.

Lemma 32.1.8. *If M is a matching and π is an augmenting path relative to M , then*

$$M' = M \oplus \pi = \{e \in E \mid e \in (M \setminus \pi) \cup (\pi \setminus M)\}$$

is a matching of size $|M| + 1$.

Proof: Think about removing π from the graph all together. What is left of M , is a matching of size $|M| - |M \cap \pi|$. Now, add back π and alternate the edges of the matching M with the free edges of π . Clearly, the new set of edges is a matching, since π is disjoint from the rest of the matching, this alternation results in a valid matching, and its size is $|M'| = |M| - |M \cap \pi| + |\pi \setminus M| = |M| + 1$. ■

Lemma 32.1.9. *Let M be a matching, and T be a maximum matching, and $k = |T| - |M|$. Then M has (at least) k vertex disjoint augmenting paths. At least one of length $\leq u/k - 1$, where $u = 2(|T| + |M|)$.*

Proof: Let $E' = M \oplus T$, and let $H = (V, E')$, where V is the set of vertices used by the edges of E' , see Figure 32.2. Clearly, every vertex in H has at most degree 2 because every vertex is adjacent to at most one edge of M and one edge of T . Thus, H is a collection of *disjoint* paths and (even length) cycles. The cycles are of even length since the edges of the cycle are alternating between two matchings (i.e., you can think about the cycle edges as being 2-colorable).

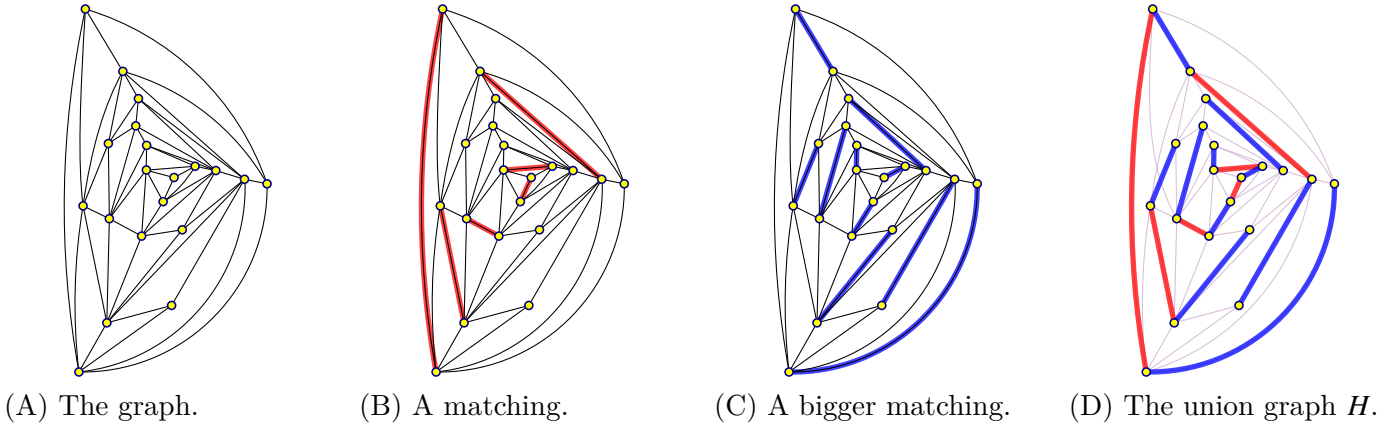


Figure 32.2: The graph formed by the union of matchings.

Now, there are k more edges of T in $M \oplus T$ than of M . Every cycle have the same number of edges of M and T . Thus, a path in H can have at most one more edge of T than of M . In such a case, this path is an augmenting path for M . It follows that there are at least k augmenting paths for M in H .

As for the claim on the length of the shortest augmenting path. Let $u = |V(H)| \leq 2(|T| + |M|)$. Observe that if all these k (vertex disjoint) augmenting paths were of length $\geq u/k$ then the total number of vertices in H would be at least $(u/k + 1)k > u$, since a path of length ℓ has $\ell + 1$ vertices. A contradiction. ■

The lemma readily implies:

Corollary 32.1.10. *A matching M is maximum \iff there is no augmenting path for M .*

32.2. Unweighted matching in bipartite graph

32.2.1. The slow algorithm; **algSlowMatch**

The algorithm. Let $G = (L \cup R, E)$ be a bipartite graph. Let $M_0 = \emptyset$ be an empty matching. In the i th iteration of **algSlowMatch**, let L_i and R_i be the free vertices in L and R , relative to the matching M_{i-1} . If there an edge in G between a vertex of L_i and R_i , we just add this edge to the matching, and go on to the next iteration.

Otherwise, we build a new graph H_i . We orient all the edges of $E \setminus M_{i-1}$ from left to the right. Formally, an edge $lr \in E \setminus M_{i-1}$, with $l \in L$ and $r \in R$, is going to induced the directed edge (l, r) in H_i . Similarly, the matching edges $lr \in M_{i-1}$ are oriented from the right to left, as the new directed edge (r, l) .

Now, using **BFS**, compute the *shortest path* π_i from a vertex of L_i to a vertex of R_i . If there is no such path, the algorithm stops and outputs the current matching (i.e., it is a maximum matching). Otherwise, the algorithm updates $M_i = M_{i-1} \oplus \pi_i$, and continues to the next iteration.

Analysis. An augmenting path has an odd number of edges. As such, if it starts in a free vertex on the left side, then it must ends in a free vertex on the right side. As such, such an augmenting path, corresponds to a path between a vertex of L_i to a vertex of R_i in H_i . By **Corollary 32.1.10**, as long as the algorithm has not computed the maximum matching, there is an augmenting path, and this path increases the size of the matching by one.

Observe, that any shortest path found in H_i between L_i and R_i is an augmenting path. Namely, if there is an augmenting path for M_{i-1} , then there is a path from a vertex of L_i to a vertex of R_i in H_i , and the algorithm computes the shortest such path.

We conclude, that after at most n iterations, the algorithm would be done. Clearly, each iteration of the algorithm can be implemented in linear time. We thus have the following result:

Lemma 32.2.1. *Given a bipartite undirected graph $G = (L \cup R, E)$, with n vertices and m edges, one can compute the maximum matching in G in $O(nm)$ time.*

32.2.2. The Hopcroft-Karp algorithm

We next improve the running time – this requires quite a bit of work, but hopefully exposes some interesting properties of matchings in bipartite graphs.

32.2.2.1. Some more structural observations

We need three basic observations:

- (A) If we augmenting along a shortest path, then the next augmenting path must be longer (or at least not shorter). See [Lemma 32.2.2](#) below.
- (B) As such, if we always augment along shortest paths, then the augmenting paths get longer as the algorithm progress, see [Corollary 32.2.3](#) below.
- (C) Furthermore, all the augmenting paths of the same length used by the algorithm are vertex-disjoint (!). See [Lemma 32.2.4](#) below. (The main idea of the faster algorithm is to compute this block of vertex-disjoint paths of the same length in one go, thus getting the improved running time.)

Lemma 32.2.2. *Let M be a matching, and π be the shortest augmenting path for M , and let π' be any augmenting path for $M' = M \oplus \pi$. Then $|\pi'| \geq |\pi|$. Specifically, we have $|\pi'| \geq |\pi| + 2|\pi \cap \pi'|$.*

Proof: Consider the matching $N = M \oplus \pi \oplus \pi'$. Observe that $|N| = |M| + 2$. As such, ignoring cycles and balanced paths, $M \oplus N$ contains two augmenting paths, say σ_1 and σ_2 – importantly, both σ_1 and σ_2 are augmenting paths of the original matching M .

Observe that for any sets B, C, D , we have $B \oplus (C \oplus D) = (B \oplus C) \oplus D$. This implies that

$$M \oplus N = M \oplus (M \oplus \pi \oplus \pi') = \pi \oplus \pi'.$$

As such, we have

$$|\pi \oplus \pi'| = |M \oplus N| \geq |\sigma_1| + |\sigma_2|.$$

Since π was the shortest augmenting path for M , it follows that $|\sigma_1| \geq |\pi|$ and $|\sigma_2| \geq |\pi|$. We conclude that

$$|\pi \oplus \pi'| \geq |\sigma_1| + |\sigma_2| \geq |\pi| + |\pi| = 2|\pi|.$$

By definition, we have that $|\pi \oplus \pi'| = |\pi| + |\pi'| - 2|\pi \cap \pi'|$. To see why the factor 2 is there, observe that for $e \in \pi \cap \pi'$ we have $e \notin \pi \oplus \pi'$. Combining with the above, we have

$$|\pi| + |\pi'| - 2|\pi \cap \pi'| \geq 2|\pi| \quad \implies \quad |\pi'| \geq |\pi| + 2|\pi \cap \pi'|. \quad \blacksquare$$

The above lemma immediately implies the following.

Corollary 32.2.3. *Let $\pi_1, \pi_2, \dots, \pi_t$ be the sequence of augmenting paths used by the algorithm of [Section 32.2.1](#) (which always augments the matching along the shortest augmenting path). We have that $|\pi_1| \leq |\pi_2| \leq \dots \leq |\pi_t|$.*

Lemma 32.2.4. *For all i and j , such that $|\pi_i| = \dots = |\pi_j|$, we have that the paths π_i and π_j are vertex disjoint.*

Proof: Assume for the sake of contradiction, that $|\pi_i| = |\pi_j|$, $i < j$, and π_i and π_j are not vertex disjoint, and assume that $j - i$ is minimal. As such, for any k , such that $i < k < j$, we have that π_k is disjoint from π_i and π_j .

Now, let M_i be the matching after π_i was applied. We have that π_j is not using any of the edges of $\pi_{i+1}, \dots, \pi_{j-1}$. As such, π_j is an augmenting path for M_i . Now, π_j and π_i share vertices. It definitely can not be that they share the two endpoints of π_j (since they are free) - so it must be some interval vertex of π_j . But

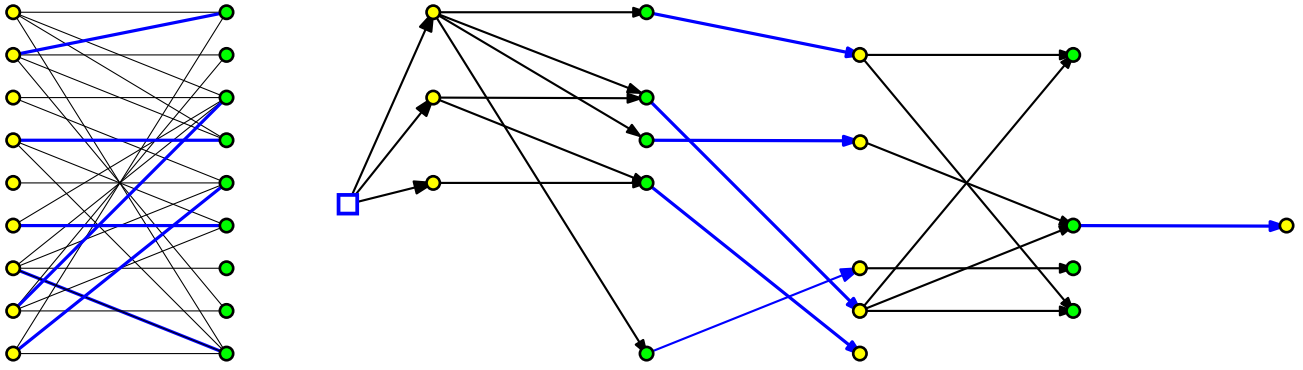


Figure 32.3: (A) A bipartite graph and its matching. (B) Its layered graph.

then, π_i and π_j must share an edge – indeed, assume the shared vertex is $v - \pi_j$ uses a matching edge of M_i adjacent to v , but this must belong to π_j - since it contains the only matching edge adjacent to v in M_i . Namely, $|\pi_i \cap \pi_j| \geq 1$. Now, by [Lemma 32.2.2](#), we conclude that

$$|\pi_j| \geq |\pi_i| + 2|\pi_i \cap \pi_j| > |\pi_i|.$$

A contradiction. ■

32.2.2.2. Improved algorithm

The idea is going to extract all possible augmenting shortest paths of a certain length in one iteration. Indeed, assume for the time being, that given a matching we can extract all augmenting paths of length k for M in G in $O(m)$ time, for $k = 1, 3, 5, \dots$. Specifically, we apply this extraction algorithm, till $k = 1 + 2 \lceil \sqrt{n} \rceil$. This would take $O(km) = O(\sqrt{nm})$ time.

The key observation is that the matching M_k , at the end of this process, is of size $|T| - \Omega(\sqrt{n})$, see [Lemma 32.2.5](#) below, where T is the maximum matching. As such, we resume the regular algorithm that augments one augmenting path at a time. After $O(\sqrt{n})$ regular iterations we would be done.

Lemma 32.2.5. *Consider the iterative algorithm that applies shortest path augmenting path to the current matching, and let M be the first matching such that the shortest path augmenting path for it is of length $\geq \sqrt{n}$, where n is the number of vertices in the input graph G . Let T be the maximum matching. Then $|T| \leq |M| + O(\sqrt{n})$.*

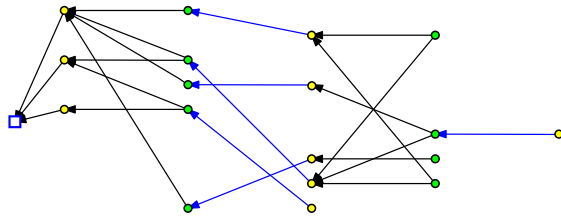
Proof: At this point, the shortest augmenting path for the current matching M is of length at $\geq \sqrt{n}$. By [Lemma 32.1.9](#), this implies that if T is the maximum matching, then we have that there is an augmenting path of length $\leq 2n/(|T| - |M|) + 1$. Combining these two inequalities, we have that

$$\sqrt{n} \leq \frac{2n}{|T| - |M|} + 1,$$

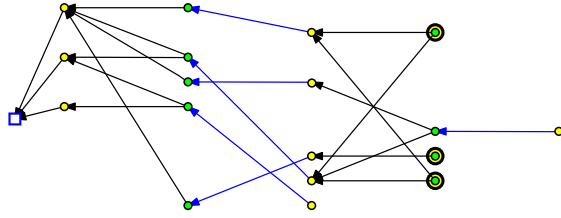
which implies that $|T| - |M| \leq 3\sqrt{n}$, for $n \geq 4$. ■

32.2.2.3. Extracting many augmenting paths: `algExtManyPaths`

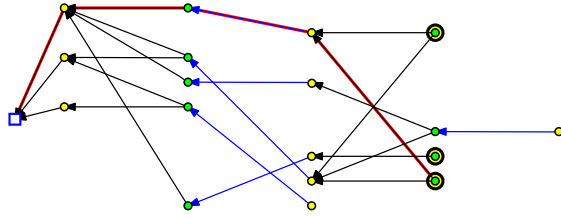
The basic idea is to build a data-structure that is similar to a **BFS** tree, but enable us to extract many augmenting paths simultaneously. So, assume we are given a graph G , as above, a matching M , and a parameter k , where k is an odd integer. Furthermore, assume that the shortest augmenting path for M in relation to G is of length k . Our purpose is to extract as many augmenting paths as possible that are vertex disjoint that are of length k ($k = 1$ is exactly the greedy algorithm for maximal matching!).



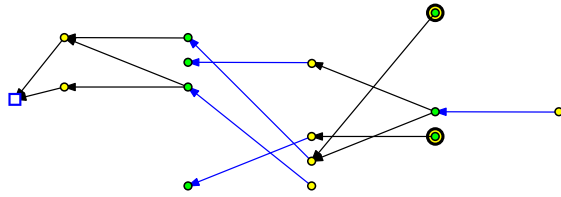
The reverse graph.



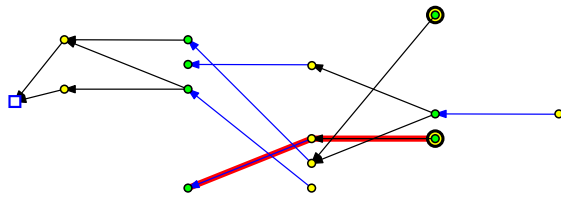
The free vertices at layer L_2 .



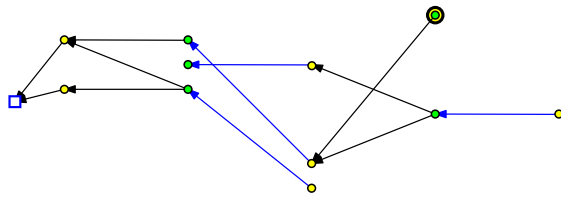
Doing **DFS** from a free vertex reveals an augmenting path.



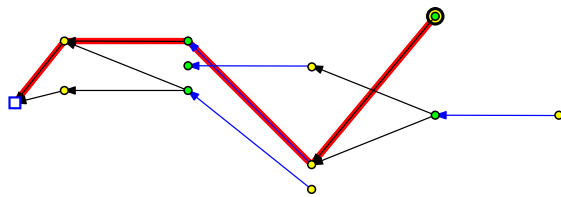
We remove the path and all the vertices it uses (except the last one, naturally).



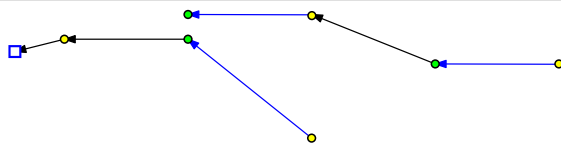
A **DFS** from a free vertex that fails to arrive to the source.



We delete (i.e., mark as visited) all the edges/vertices visited by the failed **DFS**.



Another augmenting path from a free vertex resulting in a new augmenting path.



The layered graph is empty of free vertices in the layer of interest. Time to move on to the next iteration.

Figure 32.4: Extracting augmenting paths from the reverse layered graph.

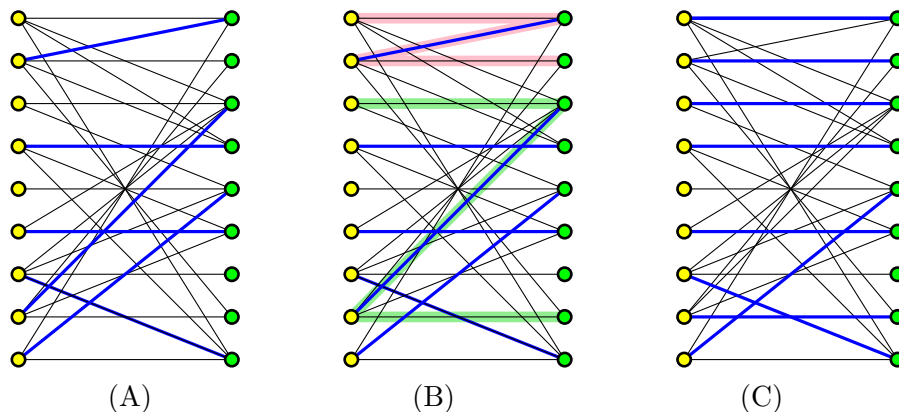


Figure 32.5: (A) A bipartite graph and its current matching. (B) Augmenting paths computed using the layered graph (see Figure 32.3). (C) The new matching after we apply the augmenting paths.

To this end, let F be the set of free vertices in G . We build a directed graph, having a source vertex s , and that is connected to all the vertices of $L_1 = L \cap F$ (all the free vertices in L). Now, we direct the edges of G , as done above, and let H be the resulting graph (i.e., non-matching edges are directed from left to right, and matching edges are directed from right to left). Now, compute **BFS** on the graph H starting at s , and let \mathcal{T} be the resulting tree.

Let $L_1, R_1, L_2, R_2, L_3, \dots$ be the layers of the **BFS**. By assumption, the first free vertex below L_1 encountered in the tree is of level R_τ , where $\tau = \lceil k/2 \rceil$ (note, that no free vertex can be encountered on L_i , for $i > 1$, since all the free vertices of L are in L_1).

Scan the edges of H . A back edge connects a vertex to a vertex that is in a higher level of the tree – we ignore such edges. The other possibilities, is an edge that is a forward edge – an edge between two vertices that belong to two consecutive levels of the **BFS** tree \mathcal{T} . Let J be the resulting graph of removing all backward and cross edges from H (a cross edge connects two vertices in the same layer of the **BFS**, which is impossible for bipartite graphs, so there are no such edges here). All the remaining edges are either **BFS** edges or forward edges, and we direct them according to the **BFS** layers from the shallower layer to the deeper layer. The resulting graph is a DAG (which is an enrichment of the original tree \mathcal{T}). Compute also the reverse graph J^{rev} (where, we just reverse the edges).

Now, let $F_\tau = R_\tau \cap F$ be the free vertices of distance k from the free vertices of L_1 (which are all free vertices). For every vertex $v \in F_\tau$ do a **DFS** in J^{rev} till the **DFS** reaches a vertex of L_1 . Mark all the vertices visited by the **DFS** as “used” – thus not allowing any future **DFS** to use these vertices (i.e., the **DFS** ignore edges leading to used vertices). If the **DFS** succeeds, we extract the shortest path found, and add it to the collection of augmenting paths. Otherwise, we move on to the next vertex in F_τ , till we visit all such vertices.

This algorithm results in a collection of augmenting paths P_τ , which are vertex disjoint. We claim that P_τ is the desired set maximal cardinality disjoint set of augmenting paths of length k .

Analysis. Building the initial graphs J and J^{rev} takes $O(m)$ time. We charge the running time of the second stage to the edges and vertices visited. Since any vertex visited by any **DFS** is never going to be visited again, this imply that an edge of J^{rev} is going to be considered only once by the algorithm. As such, the running time of the algorithm is $O(n + m)$ as desired.

Repeated application of **Lemma 32.2.2** implies the following.

Observation 32.2.6. *Assume M is a matching, such that the shortest augmenting path for it is of length k . Then, augmenting it with a sequence of paths of length k , results in matching M' , with its shortest augmenting path being of length at least k .*

Lemma 32.2.7. *The set P_k is a maximal set of vertex-disjoint augmenting paths of length k for M .*

Proof: Let M' be the result of augmenting M with the paths of P_k . And, assume for the sake of contradiction, that P_k is not maximal. Namely, there is an augmenting path σ that is disjoint from the vertices of the paths of P_k . By the above observation, the path σ is of length at least k .

The interesting case here is if σ is of length exactly k . Then, we could traverse σ in J , and this would go through unused vertices. Indeed, if any of the vertices of σ were used by any of the DFS, then it would have resulted in a path that goes to a free vertex in L_1 . But that is a contradiction, as σ is supposedly disjoint from the paths of P_k . ■

32.2.2.4. The result

Theorem 32.2.8. *Given a bipartite unweighted graph G with n vertices and m edges, one can compute maximum matching in G in $O(\sqrt{nm})$ time.*

Proof: The `algMatchingHK` algorithm is described in Section 32.2.2.2, and the running time analysis is done above.

The main challenge is the correctness. The idea is to interpret the execution of this algorithm as simulating the slower the simpler algorithm of Section 32.2.1. Indeed, the `algMatchingHK` algorithm computes a sequence of sets of augmenting paths P_1, P_3, P_5, \dots . We order these augmenting paths in an arbitrary order inside each such set. This results in a sequence of augmenting paths that are shortest augmenting paths for the current matching, and furthermore by Lemma 32.2.7 each set P_k contains a maximal set of such vertex-disjoint augmenting paths of length k . By Lemma 32.2.4, all augmenting paths of length k computed are vertex disjoint.

As such, now by induction, we can argue that if `algMatchingHK` simulates correctly `algSlowMatch`, for the augmenting paths in $P_1 \cup P_3 \cup \dots P_i$, then it simulates it correctly for $P_1 \cup P_3 \cup \dots P_i \cup P_{i+1}$, and we are done. ■

32.3. Bibliographical notes

The description here follows the original paper of Hopcroft and Karp [HK73].

Chapter 33

Matchings II

He left his desk, bore down on me on tiptoe and, bending toward me, whispered in tones full of hatred that the elephants were a mere political diversion and that of course he knew what the real issue was, but if communism triumphed in Africa, the elephants would be the first to die.

The roots of heaven, Romain Gary

33.1. Maximum weight matchings in a bipartite graph

33.1.1. On the structure of the problem

For an alternating path/cycle π , its *weight*, in relation to a matching M , is

$$\gamma(\pi, M) = \sum_{e \in \pi \setminus M} \omega(e) - \sum_{e \in \pi \cap M} \omega(e). \quad (33.1)$$

Namely, it is the total weight of the free edges in π minus the weight of the matched edges. This is a natural concept because of the following lemma.

Lemma 33.1.1. *Let M be a matching, and let π be an alternating path/cycle with positive weight relative to M ; that is $\gamma(\pi, M) > 0$. Furthermore, assume that*

$$M' = M \oplus \pi = (M \setminus \pi) \cup (\pi \setminus M)$$

is a matching. Then $\omega(M')$ is bigger; namely, $\omega(M') > \omega(M)$.

Proof: We have that

$$\omega(M') - \omega(M) = \sum_{e \in M'} \omega(e) - \sum_{e \in M} \omega(e) = \sum_{e \in M' \setminus M} \omega(e) - \sum_{e \in M \setminus M'} \omega(e) = \sum_{e \in \pi \setminus M} \omega(e) - \sum_{e \in M \setminus \pi} \omega(e) = \gamma(\pi, M).$$

As such, we have that $\omega(M') = \omega(M) + \gamma(\pi, M)$. ■

We remind the reader that an alternating path is *augmenting* if it starts and ends in a free vertex.

Observation 33.1.2. *If M has an augmenting path π then M is not of maximum size matching (this is for the unweighted case), since $M \oplus \pi$ is a larger matching.*

Theorem 33.1.3. *Let M be a matching of maximum weight among matchings of size $|M|$. Let π be an augmenting path for M of maximum weight, and let T be the matching formed by augmenting M using π . Then T is of maximum weight among matchings of size $|M| + 1$.*

Proof: Let S be a matching of maximum weight among all matchings with $|M| + 1$ edges. And consider $H = (V, M \oplus S)$.

Consider a cycle σ in H . The weight $\gamma(\sigma, M)$ (see Eq. (33.1)) must be zero. Indeed, if $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ is a matching of the same size as M which is heavier than M . A contradiction to the definition of M as the maximum weight such matching.

Similarly, if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S . A contradiction.

By the same argumentation, if σ is a path of even length in the graph H then $\gamma(\sigma, M) = 0$ by the same argumentation.

Let U_S be all the odd length paths in H that have one edge more in S than in M , and similarly, let U_M be the odd length paths in H that have one edge more of M than an edge of S .

We know that $|U_S| - |U_M| = 1$ since S has one more edge than M . Now, consider a path $\pi \in U_S$ and a path $\pi' \in U_M$. It must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$. Indeed, if $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ would have bigger weight than M while having the same number of edges. Similarly, if $\gamma(\pi, M) + \gamma(\pi', M) < 0$ (compared to M) then $S \oplus \pi \oplus \pi'$ would have the same number of edges as S while being a heavier matching. A contradiction.

Thus, $\gamma(\pi, M) + \gamma(\pi', M) = 0$. Thus, we can pair up the paths in U_S to paths in U_M , and the total weight of such a pair is zero, by the above argumentation. There is only one path μ in U_S which is not paired, and it must be that $\gamma(\mu, M) = \omega(S) - \omega(M)$ (since everything else in H has zero weight as we apply it to M to get S).

This establishes the claim that we can augment M with a single path to get a maximum weight matching of cardinality $|M| + 1$. Clearly, this path must be the heaviest augmenting path that exists for M . Otherwise, there would be a heavier augmenting path σ' for M such that $\omega(M \oplus \sigma') > \omega(S)$. A contradiction to the maximality of S . ■

The above theorem imply that if we always augment along the maximum weight augmenting path, than we would get the maximum weight matching in the end.

33.1.2. Maximum Weight Matchings in a bipartite Graph

33.1.2.1. Building the residual graph

Let $G = (L \cup R, E)$ be the given bipartite graph, with $w : E \rightarrow \mathbb{R}$ be the non-negative weight function. Given a matching M , let G_M to be the directed graph, where

- (i) For all edges $rl \in M$, $l \in L$ and $r \in R$, the edge (r, l) is added to G_M , with weight $\alpha((r, l)) = \omega(rl)$.
- (ii) For all edges $rl \in E \setminus M$, the edge $(l \rightarrow r)$ is added to G_M , with weight $\alpha((l, r)) = -\omega(rl)$.

Namely, we direct all the matching edges from right to left, and assign them their weight, and we direct all other edges from left to right, with their negated weight. Let G_M denote the resulting graph.

An augmenting path π in G must have an odd number of edges. Since G is bipartite, π must have one endpoint on the left side and one endpoint on the right side. Observe, that a path π in G_M has weight $\alpha(\pi) = -\gamma(\pi, M)$.

Let U_L be all the unmatched vertices in L and let U_R be all the unmatched vertices in R .

Thus, what we are looking for is a path π in G_M starting U_L going to U_R with maximum weight $\gamma(\pi)$, namely with minimum weight $\alpha(\pi)$.

Lemma 33.1.4. *If M is a maximum weight matching with k edges in G , then there is no negative cycle in G_M where $\alpha(\cdot)$ is the associated weight function.*

Proof: Assume for the sake of contradiction that there is a cycle C , and observe that $\gamma(C) = -\alpha(C) > 0$. Namely, $M \oplus C$ is a new matching with bigger weight and the same number of edges. A contradiction to the maximality of M . ■

33.1.2.2. The algorithm

So, we now can find a maximum weight in the bipartite graph G as follows: Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M , apply it, and repeat till M is maximal.

Thus, we need to find a minimum weight path in G_M between U_L and U_R (because we flip weights). This is the problem of computing a shortest path in the graph G_M which does not have negative cycles, and this can be done by using the **Bellman-Ford** algorithm. Indeed, collapse all the vertices of U_L into a single vertex, and all the uncovered vertices of U_R into a single vertex. Let H_M be the resulting graph. Clearly, we are looking for the shortest path between the two vertices corresponding to U_L and U_R in H_M and since this graph has no negative cycles, this can be done using the **Bellman-Ford** algorithm, which takes $O(nm)$ time. We conclude:

Lemma 33.1.5. *Given a bipartite graph G and a maximum weight matching M of size k one can find a maximum weight augmenting path for M in G , in $O(nm)$ time, where n is the number of vertices of G and m is the number of edges.*

We need to apply this algorithm $n/2$ times at most, as such, we get:

Theorem 33.1.6. *Given a weight bipartite graph G , with n vertices and m edges, one can compute a maximum weight matching in G in $O(n^2m)$ time.*

33.1.3. Faster Algorithm

It turns out that the graph here is very special, and one can use the Dijkstra algorithm. We omit any further details, and state the result. The interested student can figure out the details (warning: this is not easy). or lookup the literature.

Theorem 33.1.7. *Given a weight bipartite graph G , with n vertices and m edges, one can compute a maximum weight matching in G in $O(n(n \log n + m))$ time.*

33.2. The Bellman-Ford algorithm - a quick reminder

The **Bellman-Ford** algorithm computes the shortest path from a single source s in a graph G that has no negative cycles to all the vertices in the graph. Here G has n vertices and m edges. The algorithm works by initializing all distances to the source to be ∞ (formally, for all $u \in V(G)$, we set $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$). Then, it n times scans all the edges, and for every edge $(u, v) \in E(G)$ it performs a **Relax** (u, v) operation. The relax operation checks if $x = d[u] + w((u, v)) < d[v]$, and if so, it updates $d[v]$ to x , where $d[u]$ denotes the current distance from s to u . Since **Relax** (u, v) operation can be performed in constant time, and we scan all the edges n times, it follows that the overall running time is $O(mn)$.

We claim that in the end of the execution of the algorithm the shortest path length from s to u is $d[u]$, for all $u \in V(G)$. Indeed, every time we scan the edges, we set at least one vertex distance to its final value (which is its shortest path length). More formally, all vertices that their shortest path to s have i edges, are being set to their shortest path length in the i th iteration of the algorithm, as can be easily proved by induction. This implies the stated bound on the running time of the algorithm.

Notice, that if we want to detect negative cycles, we can run **Bellman-Ford** for an additional iteration. If the distances changes, we know that there is a negative cycle somewhere in the graph.

33.3. Maximum size matching in a non-bipartite graph

The results from the previous lecture suggests a natural algorithm for computing a maximum size (i.e., matching with maximum number of edges in it) matching in a general (i.e., not necessarily bipartite) graph. Start from an empty matching M and repeatedly find an augmenting path from an unmatched vertex to an unmatched vertex. Here we are discussing the unweighted case.

Notations. Let \mathcal{T} be a given tree. For two vertices $x, y \in V(\mathcal{T})$, let τ_{xy} denote the path in \mathcal{T} between x and y . For two paths π and π' that share an endpoint, let $\pi \parallel \pi'$ denotes the path resulting from concatenating π to π' . For a path π , let $|\pi|$ denote the number of edges in π .

33.3.1. Finding an augmenting path

We are given a graph G and a matching M , and we would to compute a bigger matching in G . We will do it by computing an augmenting path for M .

We first observe that if G has any edge with both endpoints being free, we can just add it to the current matching. Thus, in the following, we assume that for all edges, at least one of their endpoint is covered by the current matching M . Our task is to find an augmenting path in M .

Let H be the result of collapsing all the unmatched vertices in G into a single (special) vertex s .

Next, we compute an **alternating BFS** of H starting from s . Formally, we perform a **BFS** on H starting from s such that for the even levels of the tree the algorithm is allowed to traverse only edges in the matching M , and in odd levels the algorithm traverses the unmatched edges. Let \mathcal{T} denote the resulting tree.

An augmenting path in G corresponds to an odd cycle in H with passing through the vertex s .

Definition 33.3.1. An edge $uv \in E(G)$ is a **bridge** if the following conditions are met:

- (i) u and v have the same depth in \mathcal{T} ,
- (ii) if the depth of u in \mathcal{T} is even then uv is free (i.e., $uv \notin M$, and
- (iii) if the depth of u in \mathcal{T} is odd then $uv \in M$.

Note, that given an edge uv we can check if it is a bridge in constant time after linear time preprocessing of \mathcal{T} and G .

The following is an easy technical lemma.

Lemma 33.3.2. *Let v be a vertex of G , M a matching in G , and let π be the shortest alternating path between s and v in G . Furthermore, assume that for any vertex w of π the shortest alternating path between w and s is the path along π .*

Then, the depth $d_{\mathcal{T}}(v)$ of v in \mathcal{T} is $|\pi|$.

Proof: By induction on $|\pi|$. For $|\pi| = 1$ the proof trivially holds, since then v is a neighbor of s in G , and as such it is a child of s in \mathcal{T} .

For $|\pi| = k$, consider the vertex just before v on π , and let us denote it by u . By induction, the depth of u in \mathcal{T} is $k - 1$. Thus, when the algorithm computing the alternating BFS visited u , it tried to hang v from it in the next iteration. The only possibility for failure is if the algorithm already hanged v in earlier iteration of the algorithm. But then, there exists a shorter alternating path from s to v , which is a contradiction. ■

Lemma 33.3.3. *If there is an augmenting path in G for a matching M , then there exists an edge $uv \in E(G)$ which is a bridge in \mathcal{T} .*

Proof: Let π be an augmenting path in G . The path π corresponds to an odd length alternating cycle in H . Let σ be the shortest odd length alternating cycle in G (note that both edges in σ that are adjacent to s are unmatched).

For a vertex x of σ , let $d(x)$ be the length of the shortest alternating path between x and s in H . Similarly, let $d'(x)$ be the length of the shortest alternating path between s and x along σ . Clearly, $d(x) \leq d'(x)$.

The claim is that $d(x) = d'(x)$, for all $x \in \sigma$. Indeed, assume for the sake of contradiction that $d(x) < d'(x)$, and let π_1, π_2 be the two paths from x to s formed by σ . Let η be the shortest alternating path between s and x . We know that $|\eta| < |\pi_1|$ and $|\eta| < |\pi_2|$. It is now easy to verify that either $\pi_1 \parallel \eta$ or $\pi_2 \parallel \eta$ is an alternating cycle shorter than σ involving s , which is a contradiction.

But then, take the two vertices of σ furthest away from s . Clearly, both of them have the same depth in \mathcal{T} , since $d(u) = d'(u) = d'(v) = d(v)$. By Lemma 33.3.2, we now have that $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Establishing the first part of the claim. See Figure 33.1.

As for the second claim, observe that it easily follows as σ is created from an alternating path. ■

Thus, we can do the following: Compute the alternating BFS \mathcal{T} for H , and find a bridge uv in it. If M is not a maximal matching, then there exists an augmenting path for G , and by Lemma 33.3.3 there exists a bridge. Computing the bridge uv takes $O(m)$ time.

Extract the paths from s to u and from s to v in \mathcal{T} , and glue them together with the edge uv to form an odd cycle μ in H ; namely, $\mu = \tau_{su} \parallel uv \parallel \tau_{vs}$. If μ corresponds to an alternating path in G then we are done, since we found an alternating path, and we can apply it and find a bigger matching.

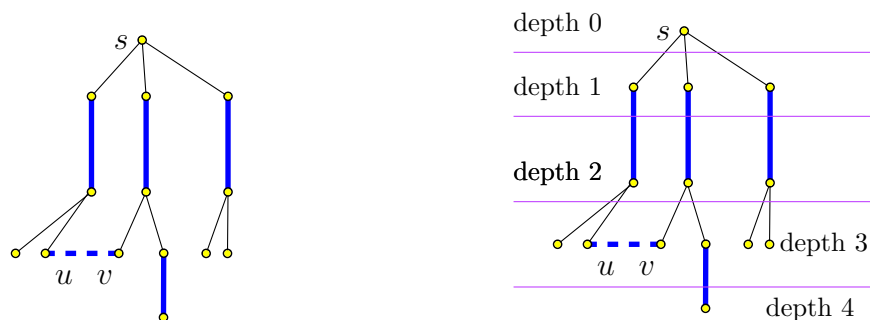
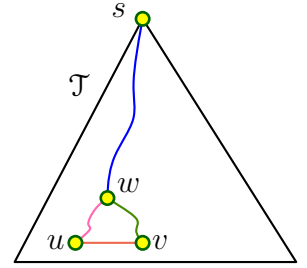


Figure 33.1: A cycle in the alternating BFS tree. Depths in the alternating BFS tree.

But μ might have repeated edges. In particular, let π_{su} and π_{sv} be the two paths from s to u and v , respectively. Let w be the lowest vertex in \mathcal{T} that is common to both π_{su} and π_{sv} .



Definition 33.3.4. Given a matching M , a **flower** for M is formed by a **stem** and a **blossom**. The stem is an even length alternating path starting at a free vertex v ending at vertex w , and the blossom is an odd length (alternating) cycle based at w .

Lemma 33.3.5. Consider a bridge edge $uv \in G$, and let w be the least common ancestor (LCA) of u and v in \mathcal{T} . Consider the path π_{sw} together with the cycle $C = \pi_{wu} \parallel uv \parallel \pi_{wv}$. Then π_{sw} and C together form a flower.

Proof: Since only the even depth nodes in \mathcal{T} have more than one child, w must be of even depth, and as such π_{sw} is of even length. As for the second claim, observe that $\alpha = |\pi_{wu}| = |\pi_{wv}|$ since the two nodes have the same depth in \mathcal{T} . In particular, $|C| = |\pi_{wu}| + |\pi_{wv}| + 1 = 2\alpha + 1$, which is an odd number. ■

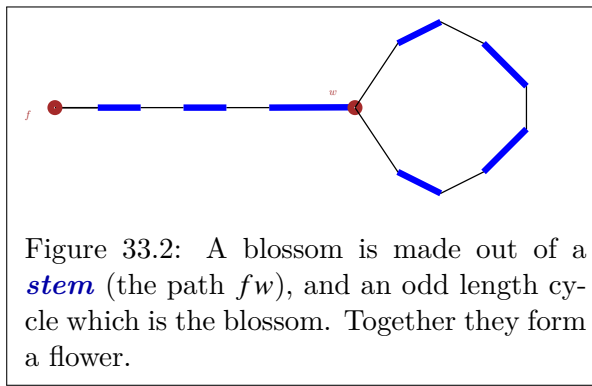


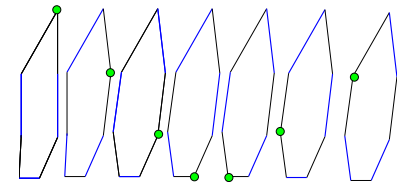
Figure 33.2: A blossom is made out of a **stem** (the path fw), and an odd length cycle which is the blossom. Together they form a flower.

Let us translate this blossom of H back to the original graph G . The path s to w corresponds to an alternating path starting at a free vertex f (of G) and ending at w , where the last edge is in the stem is in the matching, the cycle $w \dots u \dots v \dots w$ is an alternating odd length cycle in G where the two edges adjacent to w are unmatched.

We can not apply a blossom to a matching in the hope of getting better matching. In fact, this is illegal and yield something which is not a matching. On the positive side, we discovered an odd alternating cycle in the graph G . Summarizing the above algorithm, we have:

Lemma 33.3.6. Given a graph G with n vertices and m edges, and a matching M , one can find in $O(n + m)$ time, either a blossom in G or an augmenting path in G .

To see what to do next, we have to realize how a matching in G interact with an odd length cycle which is computed by our algorithm (i.e., blossom). In particular, assume that the free vertex in the cycle is unmatched. To get a maximum number of edges of the matching in the cycle, we must at most $(n - 1)/2$ edges in the cycle, but then we can rotate the matching edges in the cycle, such that any vertex on the cycle can be free. See figure on the right.



Let G/C denote the graph resulting from collapsing such an odd cycle C into single vertex. The new vertex is marked by $\{C\}$.

Lemma 33.3.7. Given a graph G , a matching M , and a flower B , one can find a matching M' with the same cardinality, such that the blossom of B contains a free (i.e., unmatched) vertex in M' .

Proof: If the stem of B is empty and B is just formed by a blossom, and then we are done. Otherwise, B was as stem π which is an even length alternating path starting from a free vertex v . Observe that the matching $M' = M \oplus \pi$ is of the same cardinality, and the cycle in B now becomes an alternating odd cycle, with a free vertex.

Intuitively, what we did is to apply the stem to the matching M . See [Figure 33.3](#). ■

Theorem 33.3.8. Let M be a matching, and let C be a blossom for M with an unmatched vertex v . Then, M is a maximum matching in G if and only if $M/C = M \setminus C$ is a maximum matching in G/C .

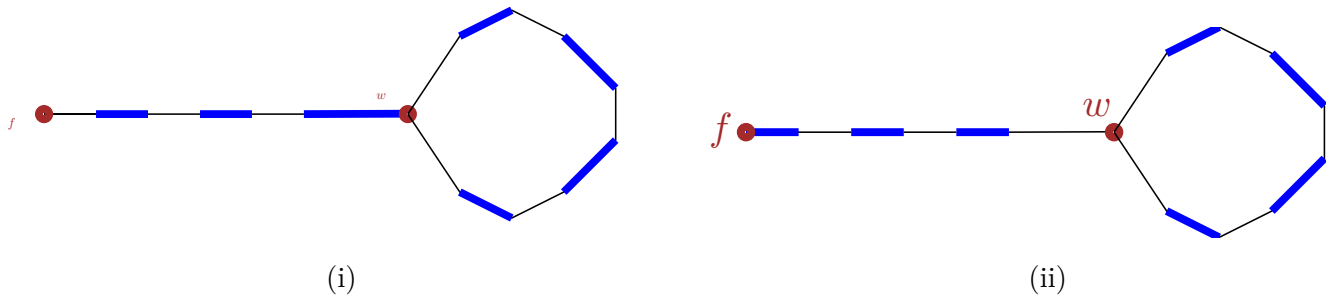


Figure 33.3: (i) the flower, and (ii) the invert stem.

Proof: Let G/C be the collapsed graph, with $\{C\}$ denoting the vertex that correspond to the cycle C .

Note, that the collapsed vertex $\{C\}$ in G/C is free. Thus, an augmenting path π in G/C either avoids the collapsed vertex $\{C\}$ altogether, or it starts or ends there. In any case, we can rotate the matching around C such that π would be an augmenting path in G . Thus, if M/C is not a maximum matching in G/C then there exists an augmenting path in G/C , which in turn is an augmenting path in G , and as such M is not a maximum matching in G .

Similarly, if π is an augmenting path in G and it avoids C then it is also an augmenting path in G/C , and then M/C is not a maximum matching in G/C .

Otherwise, since π starts and ends in two different free vertices and C has only one free vertex, it follows that π has an endpoint outside C . Let v be this endpoint of π and let u be the first vertex of π that belongs to C . Let σ be the path $\pi[v, u]$.

Let f be the free vertex of C . Note that f is unmatched. Now, if $u = f$ we are done, since then π is an augmenting path also in G/C . Note that if u is matched in C , as such, it must be that the last edge e in π is unmatched. Thus, rotate the matching M around C such that u becomes free. Clearly, then σ is now an augmenting path in G (for the rotated matching) and also an augmenting path in G/C . ■

Corollary 33.3.9. *Let M be a matching, and let C be an alternating odd length cycle with the unmatched vertex being free. Then, there is an augmenting path in G if and only if there is an augmenting path in G/C .*

33.3.2. The algorithm

Start from the empty matching M in the graph G .

Now, repeatedly, try to enlarge the matching. First, check if you can find an edge with both endpoints being free, and if so add it to the matching. Otherwise, compute the graph H (this is the graph where all the free vertices are collapsed into a single vertex), and compute an alternating BFS tree in H . From the alternating BFS, we can extract the shortest alternating cycle based in the root (by finding the highest bridge). If this alternating cycle corresponds to an alternating path in G then we are done, as we can just apply this alternating path to the matching M getting a bigger matching.

If this is a flower, with a stem ρ and a blossom C then apply the stem to M (i.e., compute the matching $M \oplus \rho$). Now, C is an odd cycle with the free vertex being unmatched. Compute recursively an augmenting path π in G/C . By the above discussing, we can easily transform this into an augmenting path in G . Apply this augmenting path to M .

Thus, we succeeded in computing a matching with one edge more in it. Repeat till the process get stuck. Clearly, what we have is a maximum size matching.

33.3.2.1. Running time analysis

Every shrink cost us $O(m + n)$ time. We need to perform $O(n)$ recursive shrink operations till we find an augmenting path, if such a path exists. Thus, finding an augmenting path takes $O(n(m + n))$ time. Finally, we have to repeat this $O(n)$ times. Thus, overall, the running time of our algorithm is $O(n^2(m + n)) = O(n^4)$.

Theorem 33.3.10. *Given a graph G with n vertices and m edges, computing a maximum size matching in G can be done in $O(n^2m)$ time.*

33.4. Maximum weight matching in a non-bipartite graph

This the hardest case and it is non-trivial to handle. There are known polynomial time algorithms, but I feel that they are too involved, and somewhat cryptic, and as such should not be presented in class. For the interested student, a nice description of such an algorithm is presented in

Combinatorial Optimization - Polyhedral and efficiency
by Alexander Schrijver
Vol. A, 453–459.

The description above also follows loosely the same book.

Chapter 34

Lower Bounds

34.1. Sorting

We all know that sorting can be done in $O(n \log n)$ time. Interestingly enough, one can show that one needs $\Omega(n \log n)$ time to solve this.

Rules of engagement. We need to define exactly what the sorting algorithm can do, or can not do. In the comparison model, we allow the sorting algorithm to do only one operation: it compare two elements. To this end, we provide the sorting algorithm a black box `compare(i, j)` that compares the i th element in the input to the j th element.

Problem statement. Our purpose is to solve the following problem.

Problem 34.1.1. Consider an input of n distinct elements, with an ordering defining over them. In the worst, how many calls to the comparison subroutine (i.e., `compare`) a deterministic sorting algorithm have to perform?

34.1.1. Decision trees

Well, we can think about a sorting algorithm as a decision procedure, at each stage, it has the current collection of comparisons it already resolved, and it need to decide which comparison to perform next. We can describe this as a decision tree, see [Figure 34.1](#). The algorithm starts at the root.

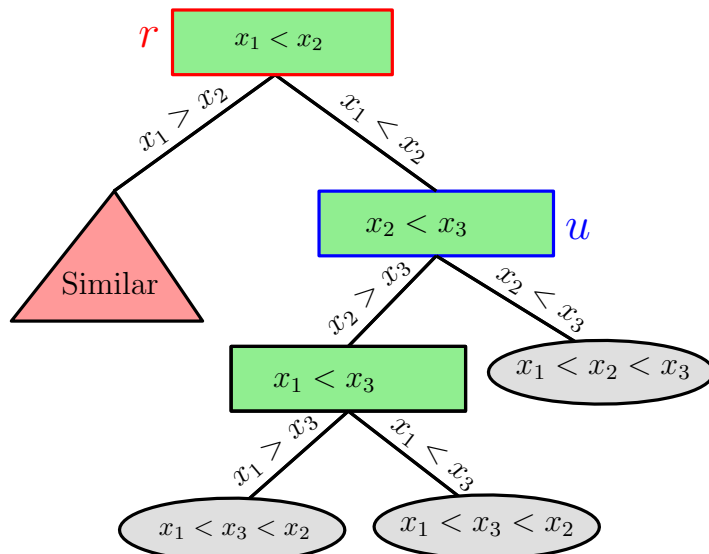


Figure 34.1: A decision tree for sorting three elements.

But what is a sorting algorithm? The output of a sorting algorithm is the input elements in a certain order. That is, a sorting algorithm for n elements outputs a permutation π of $\llbracket n \rrbracket = \{1, \dots, n\}$. Formally, if the input is x_1, \dots, x_n the output is a permutation π of $\llbracket n \rrbracket$, such that $x_{\pi(1)} < x_{\pi(2)} < \dots < x_{\pi(n)}$.

Initially all $n!$ permutations are possible, but as the algorithm performs comparisons, and as the algorithm descends in the tree it rules out some of these orderings as not being feasible. For example, the root r of the decision tree of Figure 34.1 has all possible 6 permutations as a possible output; that is, $\Phi(r) = \{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\}$. But after the comparison in the root is performed and the algorithm decides that $x_1 < x_2$, then the algorithm descends into the node u , and the possible orderings of the output that are still valid (in light of the comparison the algorithm performed), is $\Phi(u) = \{(1, 2, 3), (1, 3, 2), (3, 1, 2)\}$.

In particular, for a node v of the decision tree, let $\Phi(v)$ be the set of *feasible permutations*; that is, it is the set of all permutations that are compatible with the set of comparisons that were performed from the root to v .

Example 34.1.2. Assume the input is x_1, x_2, x_3, x_4 . If the permutation $\{(3, 4, 1, 2)\}$ is in $\Phi(v)$ then as far as the comparisons the algorithm performed in traveling from the root to v , it might be that this specific ordering of the input is a valid ordering. That is, it might be that $x_3 < x_4 < x_1 < x_2$.

Lemma 34.1.3. *Given a permutation π of $\llbracket n \rrbracket$, an input that is sorted in the ordering specified by π is the following: $x_i = \pi^{-1}(i)$, for $i = 1, \dots, n$.*

Proof: The input we construct would be made out of the numbers of $\llbracket n \rrbracket$. Now, clearly, $x_{\pi(1)}$ must be the smaller number, that is 1, namely $x_{\pi(1)} = 1$. Applying this argument repeatedly, we have that $x_{\pi(i)} = i$, for all i . In particular, take $j = \pi^{-1}(i)$, and observe that $x_i = x_{\pi(\pi^{-1}(i))} = x_{\pi(j)} = j = \pi^{-1}(i)$, as claimed. ■

Example 34.1.4. A convenient way to do the above transformation is the following. Write the permutation as a function $\llbracket n \rrbracket$ by writing it as matrix with two rows, the top row having $1, \dots, n$, and the second row having the real permutation. Computing the inverse permutation is then no more than exchanging the two lines, and sorting the columns. For example, for $\pi = (3, 4, 2, 1) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 2 & 1 \end{pmatrix}$. Then the input realizing this permutation,

is the input $\pi^{-1} = (3, 4, 1, 2) = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \end{pmatrix}$. Specifically, the input $x_1 = 4$, $x_2 = 3$, $x_3 = 1$, and $x_4 = 2$.

Observation 34.1.5. *Assume the algorithm had arrived to a node v in the decision tree, where $|\Phi(v)| > 1$. Namely, there are more than one permutation of the input that comply with the comparisons performed so far by the algorithm. Then, the algorithm must continue performing comparisons (otherwise, it would not know what to output – there are still at least two possible outputs).*

Lemma 34.1.6. *Any deterministic sorting algorithm in the comparisons model, must perform $\Omega(n \log n)$ comparisons.*

Proof: An algorithm in the comparison model is a decision tree. Indeed, an execution of the sorting algorithm on a specific input is a path in this tree. Imagine running the algorithm on all possible inputs, and generating this decision tree.

Now, the idea is to use an adversary argument, which would pick the worse possible input for the given algorithm. Importantly, the adversary need to show the input it used only in the end of the execution of the algorithm – that is, it can change the input of the algorithm on the fly, as long as it does not change the answer to the comparisons already seen so far.

So, let \mathcal{T} be the decision tree associated with the algorithm, and observe that $|\Phi(r)| = n!$, where $r = \text{root}(\mathcal{T})$.

The adversary, at the beginning, has no commitment on which of the permutations of $\Phi(r)$ it is using for the input. Specifically, the adversary computes the sets $\Phi(u)$, for all the nodes $u \in \mathcal{V}(\mathcal{T})$.

Imagine, that the algorithm performed k comparisons, and it is currently at a node v_t of the decision tree. The algorithm call `compare` to perform the comparison of x_i to x_j associated with v_k . The adversary can now decide what of the two possible results this comparison returns. Let u_L, u_R be the two children of v_t , where u_L (resp. u_R) is the child if the result of the comparison is $x_i > x_j$ (resp. $x_i < x_j$).

The adversary computes $\Phi(u_L)$ and $\Phi(u_R)$. There are two cases:

- (I) If $|\Phi(u_L)| < |\Phi(u_R)|$, the adversary prefers the algorithm to continue into u_R , and as such it returns the result of comparison of x_i and x_j as $x_i < x_j$.
- (II) If $|\Phi(u_L)| \geq |\Phi(u_R)|$, the adversary returns the comparison results $x_i > x_j$.

The adversary continues the traversal down the tree in this fashion, always picking the child that has more permutations associated with it. Let v_1, \dots, v_k be the path taken by the algorithm. The input the adversary pick, is the input realizing the single permutation of $\Phi(v_k)$.

Note, that $1 = |\Phi(v_k)| \geq \frac{|\Phi(v_{k-1})|}{2} \geq \dots \geq \frac{|\Phi(v_1)|}{2^{k-1}}$. Thus, $2^{k-1} \geq |\Phi(v_1)| = n!$. Implying that $k \geq \lg(n!) + 1 =$

$\Omega(n \log n)$. We conclude that the depth of \mathcal{T} is $\Omega(n \log n)$. Namely, there is an input which forces the given sorting algorithm to perform $\Omega(n \log n)$ comparisons. ■

34.1.2. An easier direct argument

Proof: (Proof of Lemma 34.1.6.) Consider the set Π of all permutations of $\llbracket n \rrbracket$ (each can be interpreted as a sequence of the n numbers $1, \dots, n$). We treat an element $(x_1, \dots, x_n) \in \Pi$ as an input to the algorithm. Next, stream the inputs one by one through the decision tree. Each such input ends up in a leaf of the decision tree. Note, that no leaf can have two different inputs that arrive to it – indeed, if this happened, then the sorting algorithm would have failed to sort correctly one of the two inputs.

Now, the decision tree is a binary tree, it has at least $n!$ leaves, and as such, if h is the maximum depth of a node in the decision tree, we must have that $2^h \geq n!$. That is, $h \geq \lg n! = \Omega(n \log n)$, as desired. ■

The reader might wonder why we bothered to show the original proof of Lemma 34.1.6. First, the second proof is simpler because the reader is already familiar with the language of decision trees. Secondly, the original proof brings to the forefront the idea of computation as a game against an adversary – this is a rather powerful and useful idea.

34.2. Uniqueness

Problem 34.2.1 (Uniqueness). Given an input of n real numbers x_1, \dots, x_n . Decide if all the numbers are unique (i.e., different).

Intuitively, this seems significantly easier than sorting. In particular, one can solve this in expected linear time. Nevertheless, this problem is as hard as sorting.

Theorem 34.2.2. Any deterministic algorithm in the comparison model that solves *Uniqueness*, has $\Omega(n \log n)$ running time in the worst case.

Note, that the linear time algorithm mentioned above is in a different computation model (allowing floor function, randomization, etc). The proof of the above theorem is similar to the sorting case, but it is trickier. As before, let \mathcal{T} be the decision tree (note that every node has three children).

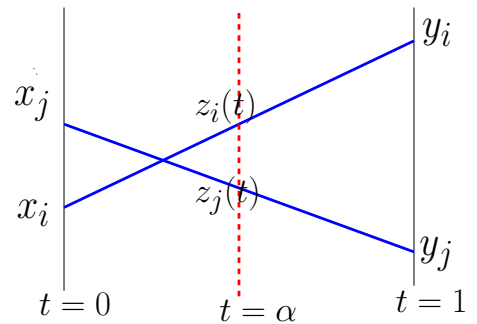
Lemma 34.2.3. For a node v in the decision tree \mathcal{T} for the given deterministic algorithm solving *Uniqueness*, if the set $\Phi(v)$ contains more than one permutation, then there exists two inputs which arrive to v , where one is unique and other is not.

Proof: Let σ and σ' be any two different permutations in $\Phi(v)$, and let $X = x_1, \dots, x_n$ be an input realizing σ , and let $Y = y_1, \dots, y_n$ be an input realizing σ' . Let $Z(t) = z_1(t), \dots, z_n(t)$ be an input where $z_i(t) = tx_i + (1-t)y_i$. Clearly, $Z(0) = x_1, \dots, x_n$ and $Z(1) = y_1, \dots, y_n$.

We claim that for any $t \in [0, 1]$ the input $Z(t)$ will arrive to the node v in \mathcal{T} .

Indeed, assume for the sake contradiction that this is false, and assume that for $t = \alpha$, that algorithm did not arrive to v in \mathcal{T} . Assume that the algorithm compared the i th element of the input to the j th element in the input, when it decided to take a different path in \mathcal{T} than the one taken for X and Y . The claim is that then $x_i < x_j$ and $y_i > y_j$ or $x_i > x_j$ and $y_i < y_j$. Namely, in such a case either X or Y will not arrive to v in \mathcal{T} .

to this end, consider the functions $z_i(t)$ and $z_j(t)$, depicted on the right. The ordering between the $z_i(t)$ and $z_j(t)$ is either the ordering between x_i and x_j or the ordering between y_i and y_j . As such, if $Z(t)$ followed a different path than X in \mathcal{T} , then Y would never arrive to v . A contradiction.



Thus, all the inputs $Z(t)$, for all $t \in [0, 1]$ arrive to the same node v .

Now, X and Y are both made of unique numbers and have a different ordering when sorted. In particular, there must be two indices, say f and g , such that, either:

- (i) $x_f < x_g$ and $y_f > y_g$, or
- (ii) $x_f > x_g$ and $y_f < y_g$.

Indeed, if there were no such indices f and g , then X and Y would have the same sorted ordering, which is a contradiction.

Now, arguing as in the above figure, there must be $\beta \in (0, 1)$ such $z_f(\beta) = z_g(\beta)$.

This is a contradiction. Indeed, there are two inputs $Z(0)$ and $Z(\beta)$ where one is unique and the other is not, such that they both arrive to the node v in the decision tree. The algorithm must continue performing comparisons to figure out what is the right output, and v can not be a leaf. ■

Proof: (of Theorem 34.2.2) We apply the same argument as in Lemma 34.1.6. If in the decision tree \mathcal{T} for *Uniqueness*, the adversary arrived to a node containing more than one permutation, it continues into the child that have more permutations associated with it. As in the sorting argument it follows that there exists a path in \mathcal{T} of length $\Omega(n \log n)$. ■

34.3. Other lower bounds

34.3.1. Algebraic tree model

In this model, at each node, we are allowed to compute a polynomial, and ask for its sign at a certain point (i.e., comparing x_i to x_j is equivalent to asking if the polynomial $x_i - x_j$ is positive/negative/zero).

One can prove things in this model, but it requires considerably stronger techniques.

Problem 34.3.1 (Degenerate points). Given a set P of n points in \mathbb{R}^d , deciding if there are $d + 1$ points in P which are co-linear (all lying on a common plane).

Theorem 34.3.2 (Jeff Erickson and Raimund Seidel [ES95]). *Solving the degenerate points problem requires $\Omega(n^d)$ time in a “reasonable” model of computation.*

34.3.2. 3SUM-Hard

Problem 34.3.3 (3SUM). Given three sets of numbers A, B, C are there three numbers $a \in A$, $b \in B$ and $c \in C$, such that $a + b = c$.

We leave the following as an exercise to the interested reader.

Lemma 34.3.4. *One can solve the 3SUM problem in $O(n^2)$ time.*

Somewhat surprisingly, no better solution is known. An interesting open problem is to find a subquadratic algorithm for 3SUM. It is widely believed that no such algorithm exists. There is a large collection problems that are 3SUM-Hard: if you solve them in subquadratic time, then you can solve 3SUM in subquadratic time. Those problems include:

- (I) For n points in the plane, is there three points that lie on the same line.
 - (II) Given a set of n triangles in the plane, do they cover the unit square
 - (III) Given two polygons P and Q can one translate P such that it is contained inside Q ?
- So, how does one prove that a problem is 3SUM-Hard? One uses reductions that have subquadratic running time. The details are interesting, but are omitted. The interested reader should check out the research on the topic [GO95].

Chapter 35

Backwards analysis

The idea of *backwards analysis* (or backward analysis) is a technique to analyze randomized algorithms by imagining as if it was running backwards in time, from output to input. Most of the more interesting applications of backward analysis are in Computational Geometry, but nevertheless, there are some other applications that are interesting and we survey some of them here.

35.1. How many times can the minimum change?

Let $\Pi = \pi_1 \dots \pi_n$ be a random permutation of $\{1, \dots, n\}$. Let \mathcal{E}_i be the event that π_i is the minimum number seen so far as we read Π ; that is, \mathcal{E}_i is the event that $\pi_i = \min_{k=1}^i \pi_k$. Let X_i be the indicator variable that is one if \mathcal{E}_i happens. We already seen, and it is easy to verify, that $\mathbb{E}[X_i] = 1/i$. We are interested in how many times the minimum might change^①; that is $Z = \sum_i X_i$, and how concentrated is the distribution of Z . The following is maybe surprising.

Lemma 35.1.1. *The events $\mathcal{E}_1, \dots, \mathcal{E}_n$ are independent (as such, variables X_1, \dots, X_n are independent).*

Proof: The trick is to think about the sampling process in a different way, and then the result readily follows. Indeed, we randomly pick a permutation of the given numbers, and set the first number to be π_n . We then, again, pick a random permutation of the remaining numbers and set the first number as the penultimate number (i.e., π_{n-1}) in the output permutation. We repeat this process till we generate the whole permutation.

Now, consider $1 \leq i_1 < i_2 < \dots < i_k \leq n$, and observe that $\mathbb{P}[\mathcal{E}_{i_1} \mid \mathcal{E}_{i_2} \cap \dots \cap \mathcal{E}_{i_k}] = \mathbb{P}[\mathcal{E}_{i_1}]$, since by our thought experiment, \mathcal{E}_{i_1} is determined after all the other variables $\mathcal{E}_{i_2}, \dots, \mathcal{E}_{i_k}$. In particular, the variable \mathcal{E}_{i_1} is inherently not effected by these events happening or not. As such, we have

$$\begin{aligned} \mathbb{P}[\mathcal{E}_{i_1} \cap \mathcal{E}_{i_2} \cap \dots \cap \mathcal{E}_{i_k}] &= \mathbb{P}[\mathcal{E}_{i_1} \mid \mathcal{E}_{i_2} \cap \dots \cap \mathcal{E}_{i_k}] \mathbb{P}[\mathcal{E}_{i_2} \cap \dots \cap \mathcal{E}_{i_k}] \\ &= \mathbb{P}[\mathcal{E}_{i_1}] \mathbb{P}[\mathcal{E}_{i_2} \cap \mathcal{E}_{i_2} \cap \dots \cap \mathcal{E}_{i_k}] = \prod_{j=1}^k \mathbb{P}[\mathcal{E}_{i_j}] = \prod_{j=1}^k \frac{1}{i_j}, \end{aligned}$$

by induction. ■

Theorem 35.1.2. *Let $\Pi = \pi_1 \dots \pi_n$ be a random permutation of $1, \dots, n$, and let Z be the number of times, that π_i is the smallest number among π_1, \dots, π_i , for $i = 1, \dots, n$. Then, we have that for $t \geq 2e$ that $\mathbb{P}[Z > t \ln n] \leq 1/n^{t \ln 2}$, and for $t \in [1, 2e]$, we have that $\mathbb{P}[Z > t \ln n] \leq 1/n^{(t-1)^2/4}$.*

Proof: Follows readily from Chernoff's inequality, as $Z = \sum_i X_i$ is a sum of independent indicator variables, and, since by linearity of expectations, we have

$$\mu = \mathbb{E}[Z] = \sum_i \mathbb{E}[X_i] = \sum_{i=1}^n \frac{1}{i} \geq \int_{x=1}^{n+1} \frac{1}{x} dx = \ln(n+1) \geq \ln n.$$

Next, we set $\delta = t - 1$, and use Chernoff inequality. ■

35.2. Yet another analysis of QuickSort

Rephrasing QuickSort. We need to restate QuickSort in a slightly different way for the backward analysis to make sense.

We conceptually can think about QuickSort as being a randomized incremental algorithm, building up a list of numbers in the order they are used as pivots. Consider the execution of QuickSort when sorting a set P of n numbers. Let $\langle p_1, \dots, p_n \rangle$ be the random permutation of the numbers picked in sequence by QuickSort. Specifically, in the i th iteration, it randomly picks a number p_i that was not handled yet, pivots based on this number, and then recursively handles the subproblems.

Specifically, assume that at the end of the i th iteration, a set $P_i = \{p_1, \dots, p_i\}$ of pivots has already been handled by the algorithm. That is, the algorithm have these pivots in sorted orders $p'_1 < p'_2 < \dots < p'_i$. In addition, the numbers that were not handled yet $P \setminus P_i$, are partitions into sets Q_0, \dots, Q_i , where all the numbers

^①The answer, my friend, is blowing in the permutation.

in $P \setminus P_i$ between p'_i and p'_{i+1} are in the set Q_i , for all i . In the $(i + 1)$ th iteration, **QuickSort** randomly picks a pivot $p_{i+1} \in P \setminus P_i$, identifies the set Q_j that contains it, splits this set according to the pivot into two sets (i.e., a set for the smaller elements, and a set for the bigger elements). The algorithm **QuickSort** continues in this fashion till all the numbers were pivots.

Lemma 35.2.1. Consider **QuickSort** being executed on a set P of n numbers. For any element $q \in P$, in expectation, q participates in $O(\log n)$ comparisons during the execution of **QuickSort**.

Proof: Consider a specific element $q \in P$. For any subset $B \subseteq P$, let $U(B)$ be the two closest numbers in B having q in between them in the original ordering of P . In other words, $U(B)$ contains the (at most) two elements that are the endpoints of the interval of $\mathbb{R} \setminus B$ that contains q . Let X_i be the indicator variable of the event that the pivot p_i used in the i th iteration is in $U(P_i)$. That is, q got compared to the i th pivot when it was inserted. Clearly, the total number of comparisons q participates in is $\sum_i X_i$.

Now, we use backward analysis. Consider the state of the algorithm just after i pivots were handled (i.e., the end of the i th iteration). Consider the set $P_i = \{p_1, \dots, p_i\}$ and imagine that you know only what elements are in this set, but the internal ordering is not known to you. As such, as there are (at most) two elements in $U(P_i)$, the probability that $p_i \in U(P_i)$ is at most $2/i$.

As such, the expected number of comparisons q participates in is $\mathbb{E}[\sum_i X_i] \leq \sum_{i=1}^n 2/i = O(\log n)$, as desired. This also implies that **QuickSort** takes $O(n \log n)$ time in expectation. ■

Exercise 35.2.2. Prove using backward analysis that **QuickSort** takes $O(n \log n)$ with high probability.

*It is not true that the indicator variables X_1, X_2, \dots are independent (this is quite subtle and not easy to see, as such extending directly the proof of **Theorem 35.1.2** for this case does not work.*

35.3. Closest pair: Backward analysis in action

We are interested in solving the following problem:

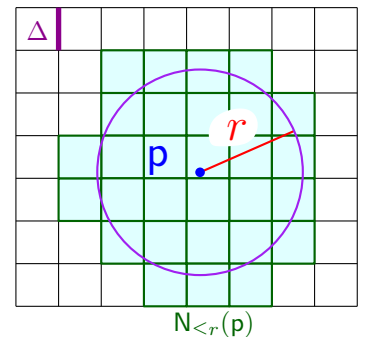
Problem 35.3.1. Given a set P of n points in the plane, find the pair of points closest to each other. Formally, return the pair of points realizing $\mathcal{CP}(P) = \min_{p \neq q, p, q \in P} \|p - q\|$.

35.3.1. Definitions

Definition 35.3.2. For a real positive number Δ and a point $p = (p_1, \dots, p_d) \in \mathbb{R}^d$, define $G_\Delta(p)$ to be the grid point $(\lfloor p_1/\Delta \rfloor \Delta, \dots, \lfloor p_d/\Delta \rfloor \Delta)$.

We call Δ the **width** or **sidelength** of the **grid** G_Δ . Observe that G_Δ partitions \mathbb{R}^d into cubes, which are grid **cells**. The grid cell of p is uniquely identified by the integer point $\text{id}(p) = (\lfloor p_1/\Delta \rfloor, \dots, \lfloor p_d/\Delta \rfloor)$.

For a number $r \geq 0$, let $N_{\leq r}(p)$ denote the set of grid cells in distance $\leq r$ from p , which is the **neighborhood** of p . Note, that the neighborhood also includes the grid cell containing p itself, and if $\Delta = \Theta(r)$ then $|N_{\leq r}(p)| = \Theta((2 + \lceil 2r/\Delta \rceil)^d) = \Theta(1)$. See figure on the right.

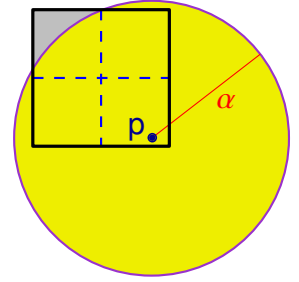


35.3.2. Back to the problem

The following is an easy standard **packing argument** that underlines, under various disguises, many algorithms in computational geometry.

Lemma 35.3.3. Let P be a set of points contained inside a square \square , such that the sidelength of \square is $\alpha = \mathcal{CP}(P)$. Then $|P| \leq 4$.

Proof: Partition \square into four equal squares $\square_1, \dots, \square_4$, and observe that each of these squares has diameter $\sqrt{2}\alpha/2 < \alpha$, and as such each can contain at most one point of P ; that is, the disk of radius α centered at a point $p \in P$ completely covers the subsquare containing it; see the figure on the right.



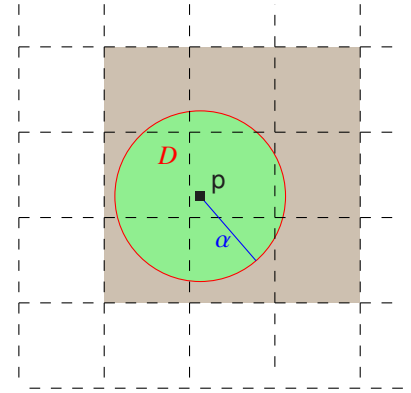
Note that the set P can have four points if it is the four corners of \square . ■

Lemma 35.3.4. Given a set P of n points in the plane and a distance α , one can verify in linear time whether $\mathcal{CP}(P) < \alpha$, $\mathcal{CP}(P) = \alpha$, or $\mathcal{CP}(P) > \alpha$.

Proof: Indeed, store the points of P in the grid G_α . For every non-empty grid cell, we maintain a linked list of the points inside it. Thus, adding a new point p takes constant time. Specifically, compute $\text{id}(p)$, check if $\text{id}(p)$ already appears in the hash table, if not, create a new linked list for the cell with this ID number, and store p in it. If a linked list already exists for $\text{id}(p)$, just add p to it. This takes $O(n)$ time overall.

Now, if any grid cell in $G_\alpha(P)$ contains more than, say, 4 points of P , then it must be that the $\mathcal{CP}(P) < \alpha$, by [Lemma 35.3.3](#).

Thus, when we insert a point p , we can fetch all the points of P that were already inserted in the cell of p and the 8 adjacent cells (i.e., all the points stored in the cluster of p); that is, these are the cells of the grid G_α that intersects the disk $D = \text{disk}(p, \alpha)$ centered at p with radius α ; see the figure on the right. If there is a point closer to p than α that was already inserted, then it must be stored in one of these 9 cells (since it must be inside D). Now, each one of those cells must contain at most 4 points of P by [Lemma 35.3.3](#) (otherwise, we would already have stopped since the $\mathcal{CP}(\cdot)$ of the inserted points is smaller than α). Let S be the set of all those points, and observe that $|S| \leq 9 \cdot 4 = O(1)$. Thus, we can compute, by brute force, the closest point to p in S . This takes $O(1)$ time. If $\mathbf{d}(p, S) < \alpha$, we stop; otherwise, we continue to the next point.



Overall, this takes at most linear time.

As for correctness, observe that the algorithm returns ' $\mathcal{CP}(P) < \alpha$ ' only after finding a pair of points of P with distance smaller than α . So, assume that p and q are the pair of points of P realizing the closest pair and that $\|p - q\| = \mathcal{CP}(P) < \alpha$. Clearly, when the later point (say p) is being inserted, the set S would contain q , and as such the algorithm would stop and return ' $\mathcal{CP}(P) < \alpha$ '. Similar argumentation works for the case that $\mathcal{CP}(P) = \alpha$. Thus if the algorithm returns ' $\mathcal{CP}(P) > \alpha$ ', it must be that $\mathcal{CP}(P)$ is not smaller than α or equal to it. Namely, it must be larger. Thus, the algorithm output is correct. ■

Remark 35.3.5. Assume that $\mathcal{CP}(P \setminus \{p\}) \geq \alpha$, but $\mathcal{CP}(P) < \alpha$. Furthermore, assume that we use [Lemma 35.3.4](#) on P , where $p \in P$ is the last point to be inserted. When p is being inserted, not only do we discover that $\mathcal{CP}(P) < \alpha$, but in fact, by checking the distance of p to all the points stored in its cluster, we can compute the closest point to p in $P \setminus \{p\}$ and denote this point by q . Clearly, pq is the closest pair in P , and this last insertion still takes only constant time.

35.3.3. Slow algorithm

[Lemma 35.3.4](#) provides a natural way of computing $\mathcal{CP}(P)$. Indeed, permute the points of P in an arbitrary fashion, and let $P = \langle p_1, \dots, p_n \rangle$. Next, let $\alpha_{i-1} = \mathcal{CP}(\{p_1, \dots, p_{i-1}\})$. We can check if $\alpha_i < \alpha_{i-1}$ by using the algorithm of [Lemma 35.3.4](#) on P_i and α_{i-1} . In fact, if $\alpha_i < \alpha_{i-1}$, the algorithm of [Lemma 35.3.4](#) would return ' $\mathcal{CP}(P_i) < \alpha_{i-1}$ ' and the two points of P_i realizing α_i .

So, consider the “good” case, where $\alpha_i = \alpha_{i-1}$; that is, the length of the shortest pair does not change when p_i is being inserted. In this case, we do not need to rebuild the data-structure of [Lemma 35.3.4](#) to store $P_i = \langle p_1, \dots, p_i \rangle$. We can just reuse the data-structure from the previous iteration that was used by P_{i-1} by inserting p_i into it. Thus, inserting a single point takes constant time, as long as the closest pair does not change.

Things become problematic when $\alpha_i < \alpha_{i-1}$, because then we need to rebuild the grid data-structure and reinsert all the points of $P_i = \langle p_1, \dots, p_i \rangle$ into the new grid $G_{\alpha_i}(P_i)$. This takes $O(i)$ time.

In the end of this process, we output the number α_n , together with the two points of P that realize the closest pair.

Observation 35.3.6. *If the closest pair distance, in the sequence $\alpha_1, \dots, \alpha_n$, changes only t times, then the running time of our algorithm would be $O(nt + n)$. Naturally, t might be $\Omega(n)$, so this algorithm might take quadratic time in the worst case.*

35.3.4. Linear time algorithm

Surprisingly^②, we can speed up the above algorithm to have linear running time by spicing it up using randomization.

We pick a random permutation of the points of P and let $\langle p_1, \dots, p_n \rangle$ be this permutation. Let $\alpha_2 = \|p_1 - p_2\|$, and start inserting the points into the data-structure of [Lemma 35.3.4](#). We will keep the invariant that α_i would be the closest pair distance in the set P_i , for $i = 2, \dots, n$.

In the i th iteration, if $\alpha_i = \alpha_{i-1}$, then this insertion takes constant time. If $\alpha_i < \alpha_{i-1}$, then we know what is the new closest pair distance α_i (see [Remark 35.3.5](#)), rebuild the grid, and reinsert the i points of P_i from scratch into the grid G_{α_i} . This rebuilding of $G_{\alpha_i}(P_i)$ takes $O(i)$ time.

Finally, the algorithm returns the number α_n and the two points of P_n realizing it, as the closest pair in P .

Lemma 35.3.7. *Let t be the number of different values in the sequence $\alpha_2, \alpha_3, \dots, \alpha_n$. Then $\mathbb{E}[t] = O(\log n)$. As such, in expectation, the above algorithm rebuilds the grid $O(\log n)$ times.*

Proof: For $i \geq 3$, let X_i be an indicator variable that is one if and only if $\alpha_i < \alpha_{i-1}$. Observe that $\mathbb{E}[X_i] = \mathbb{P}[X_i = 1]$ (as X_i is an indicator variable) and $t = \sum_{i=3}^n X_i$.

To bound $\mathbb{P}[X_i = 1] = \mathbb{P}[\alpha_i < \alpha_{i-1}]$, we (conceptually) fix the points of P_i and randomly permute them. A point $q \in P_i$ is *critical* if $\mathcal{CP}(P_i \setminus \{q\}) > \mathcal{CP}(P_i)$. If there are no critical points, then $\alpha_{i-1} = \alpha_i$ and then $\mathbb{P}[X_i = 1] = 0$ (this happens, for example, if there are two pairs of points realizing the closest distance in P_i). If there is one critical point, then $\mathbb{P}[X_i = 1] = 1/i$, as this is the probability that this critical point would be the last point in the random permutation of P_i .

Assume there are two critical points and let p, q be this unique pair of points of P_i realizing $\mathcal{CP}(P_i)$. The quantity α_i is smaller than α_{i-1} only if either p or q is p_i . The probability for that is $2/i$ (i.e., the probability in a random permutation of i objects that one of two marked objects would be the last element in the permutation).

Observe that there cannot be more than two critical points. Indeed, if p and q are two points that realize the closest distance, then if there is a third critical point s , then $\mathcal{CP}(P_i \setminus \{s\}) = \|p - q\|$, and hence the point s is not critical.

Thus, $\mathbb{P}[X_i = 1] = \mathbb{P}[\alpha_i < \alpha_{i-1}] \leq 2/i$, and by linearity of expectations, we have that $\mathbb{E}[t] = \mathbb{E}[\sum_{i=3}^n X_i] = \sum_{i=3}^n \mathbb{E}[X_i] \leq \sum_{i=3}^n 2/i = O(\log n)$. ■

[Lemma 35.3.7](#) implies that, in expectation, the algorithm rebuilds the grid $O(\log n)$ times. By [Observation 35.3.6](#), the running time of this algorithm, in expectation, is $O(n \log n)$. However, we can do better than that. Intuitively, rebuilding the grid in early iterations of the algorithm is cheap, and only late rebuilds (when $i = \Omega(n)$) are expensive, but the number of such expensive rebuilds is small (in fact, in expectation it is a constant).

^②Surprise in the eyes of the beholder. The reader might not be surprised at all and might be mildly annoyed by the whole affair. In this case, the reader should read any occurrence of “surprisingly” in the text as being “mildly annoying”.

Theorem 35.3.8. For set P of n points in the plane, one can compute the closest pair of P in expected linear time.

Proof: The algorithm is described above. As above, let X_i be the indicator variable which is 1 if $\alpha_i \neq \alpha_{i-1}$, and 0 otherwise. Clearly, the running time is proportional to

$$R = 1 + \sum_{i=3}^n (1 + X_i \cdot i).$$

Thus, the expected running time is proportional to

$$\begin{aligned} \mathbb{E}[R] &= \mathbb{E}\left[1 + \sum_{i=3}^n (1 + X_i \cdot i)\right] \leq n + \sum_{i=3}^n \mathbb{E}[X_i] \cdot i \leq n + \sum_{i=3}^n i \cdot \mathbb{P}[X_i = 1] \\ &\leq n + \sum_{i=3}^n i \cdot \frac{2}{i} \leq 3n, \end{aligned}$$

by linearity of expectation and since $\mathbb{E}[X_i] = \mathbb{P}[X_i = 1]$ and since $\mathbb{P}[X_i = 1] \leq 2/i$ (as shown in the proof of [Lemma 35.3.7](#)). Thus, the expected running time of the algorithm is $O(\mathbb{E}[R]) = O(n)$. \blacksquare

[Theorem 35.3.8](#) is a surprising result, since it implies that **uniqueness** (i.e., deciding if n real numbers are all distinct) can be solved in linear time. Indeed, compute the distance of the closest pair of the given numbers (think about the numbers as points on the x -axis). If this distance is zero, then clearly they are not all unique.

However, there is a lower bound of $\Omega(n \log n)$ on the running time to solve **uniqueness**, using the comparison model. This “reality dysfunction” can be easily explained once one realizes that the computation model of [Theorem 35.3.8](#) is considerably stronger, using hashing, randomization, and the floor function.

35.4. Computing a good ordering of the vertices of a graph

We are given a $G = (V, E)$ be an edge-weighted graph with n vertices and m edges. The task is to compute an ordering $\pi = \langle \pi_1, \dots, \pi_n \rangle$ of the vertices, and for every vertex $v \in V$, the list of vertices L_v , such that $\pi_i \in L_v$, if π_i is the closet vertex to v in the i th prefix $\langle \pi_1, \dots, \pi_i \rangle$.

This situation can arise for example in a streaming scenario, where we install servers in a network. In the i th stage there i servers installed, and every client in the network wants to know its closest server. As we install more and more servers (ultimately, every node is going to be server), each client needs to maintain its current closest server.

The purpose is to minimize the total size of these lists $\mathcal{L} = \sum_{v \in V} |L_v|$.

35.4.1. The algorithm

Take a random permutation π_1, \dots, π_n of the vertices V of G . Initially, we set $\delta(v) = +\infty$, for all $v \in V$.

In the i th iteration, set $\delta(\pi_i)$ to 0, and start Dijkstra from the i th vertex π_i . The Dijkstra propagates only if it improves the current distance associated with a vertex. Specifically, in the i th iteration, we update $\delta(u)$ to $d_G(\pi_i, u)$ if and only if $d_G(\pi_i, u) < \delta(u)$ before this iteration started. If $\delta(u)$ is updated, then we add π_i to L_u . Note, that this Dijkstra propagation process might visit only small portions of the graph in some iterations – since it improves the current distance only for few vertices.

35.4.2. Analysis

Lemma 35.4.1. The above algorithm computes a permutation π , such that $\mathbb{E}[|\mathcal{L}|] = O(n \log n)$, and the expected running time of the algorithm is $O((n \log n + m) \log n)$, where $n = |V(G)|$ and $m = |E(G)|$. Note, that both bounds also hold with high probability.

Proof: Fix a vertex $v \in V = \{v_1, \dots, v_n\}$. Consider the set of n numbers $\{d_G(v, v_1), \dots, d_G(v, v_n)\}$. Clearly, $d_G(v, \pi_1), \dots, d_G(v, \pi_n)$ is a random permutation of this set, and by [Lemma 35.1.1](#) the random permutation π changes this minimum $O(\log n)$ time in expectations (and also with high probability). This readily implies that $|L_v| = O(\log n)$ both in expectations and high probability.

The more interesting claim is the running time. Consider an edge $uv \in E(G)$, and observe that $\delta(u)$ or $\delta(v)$ changes $O(\log n)$ times. As such, an edge gets visited $O(\log n)$ times, which implies overall running time of $O(n \log^2 n + m \log n)$, as desired.

Indeed, overall there are $O(n \log n)$ changes in the value of $\delta(\cdot)$. Each such change might require one **delete-min** operation from the queue, which takes $O(\log n)$ time operation. Every edge, by the above, might trigger $O(\log n)$ **decrease-key** operations. Using Fibonacci heaps, each such operation takes $O(1)$ time. ■

35.5. Computing nets

35.5.1. Basic definitions

35.5.1.1. Metric spaces

Definition 35.5.1. A **metric space** is a pair (X, \mathbf{d}) where X is a set and $\mathbf{d} : X \times X \rightarrow [0, \infty)$ is a **metric** satisfying the following axioms: (i) $\mathbf{d}(x, y) = 0$ if and only if $x = y$, (ii) $\mathbf{d}(x, y) = \mathbf{d}(y, x)$, and (iii) $\mathbf{d}(x, y) + \mathbf{d}(y, z) \geq \mathbf{d}(x, z)$ (triangle inequality).

For example, \mathbb{R}^2 with the regular Euclidean distance is a metric space. In the following, we assume that we are given **black-box access** to \mathbf{d}_M . Namely, given two points $\mathbf{p}, \mathbf{q} \in X$, we assume that $\mathbf{d}(\mathbf{p}, \mathbf{q})$ can be computed in constant time.

Another standard example for a finite metric space is a graph G with non-negative weights $\omega(\cdot)$ defined on its edges. Let $d_G(x, y)$ denote the shortest path (under the given weights) between any $x, y \in V(G)$. It is easy to verify that $d_G(\cdot, \cdot)$ is a metric. In fact, any **finite metric** (i.e., a metric defined over a finite set) can be represented by such a weighted graph.

35.5.1.2. Nets

Definition 35.5.2. For a point set P in a metric space with a metric \mathbf{d} , and a parameter $r > 0$, an **r -net** of P is a subset $C \subseteq P$, such that

- (i) for every $\mathbf{p}, \mathbf{q} \in C$, $\mathbf{p} \neq \mathbf{q}$, we have that $\mathbf{d}(\mathbf{p}, \mathbf{q}) \geq r$, and
- (ii) for all $\mathbf{p} \in P$, we have that $\min_{\mathbf{q} \in C} \mathbf{d}(\mathbf{p}, \mathbf{q}) < r$.

Intuitively, an r -net represents P in resolution r .

35.5.2. Computing nets quickly for a point set in \mathbb{R}^d

The results here have nothing to do with backward analysis and are included here only for the sake of completeness.

There is a simple algorithm for computing r -nets. Namely, let all the points in P be initially unmarked. While there remains an unmarked point, \mathbf{p} , add \mathbf{p} to C , and mark it and all other points in distance $< r$ from \mathbf{p} (i.e. we are scooping away balls of radius r). By using grids and hashing one can modify this algorithm to run in linear time. The following is implicit in previous work [[Har04](#)], and we include it here for the sake of completeness^③ – it was also described by the authors in [[ERH12](#)].

Lemma 35.5.3. *Given a point set $P \subseteq \mathbb{R}^d$ of size n and a parameter $r > 0$, one can compute an r -net for P in $O(n)$ time.*

^③Specifically, the algorithm of Har-Peled [[Har04](#)] is considerably more complicated than [Lemma 35.5.3](#), and does not work in this settings, as the number of clusters it can handle is limited to $O(n^{1/6})$. [Lemma 35.5.3](#) has no such restriction.

Proof: Let G denote the grid in \mathbb{R}^d with side length $\Delta = r/(2\sqrt{d})$. First compute for every point $p \in P$ the grid cell in G that contains p ; that is, $\text{id}(p)$. Let \mathcal{G} denote the set of grid cells of G that contain points of P . Similarly, for every cell $\square \in \mathcal{G}$ we compute the set of points of P which it contains. This task can be performed in linear time using hashing and bucketing assuming the floor function can be computed in constant time. Specifically, store the $\text{id}(\cdot)$ values in a hash table, and in constant time hash each point into its appropriate bin.

Scan the points of P one at a time, and let p be the current point. If p is marked then move on to the next point. Otherwise, add p to the set of net points, C , and mark it and each point $q \in P$ such that $\|p - q\| < r$. Since the cells of $N_{\leq r}(p)$ contain all such points, we only need to check the lists of points stored in these grid cells. At the end of this procedure every point is marked. Since a point can only be marked if it is in distance $< r$ from some net point, and a net point is only created if it is unmarked when visited, this implies that C is an r -net.

As for the running time, observe that a grid cell, c , has its list scanned only if c is in the neighborhood of some created net point. As $\Delta = \Theta(r)$, there are only $O(1)$ cells which could contain a net point p such that $c \in N_{\leq r}(p)$. Furthermore, at most one net point lies in a single cell since the diameter of a grid cell is strictly smaller than r . Therefore each grid cell had its list scanned $O(1)$ times. Since the only real work done is in scanning the cell lists and since the cell lists are disjoint, this implies an $O(n)$ running time overall. ■

Observe that the closest net point, for a point $p \in P$, must be in one of its neighborhood's grid cells. Since every grid cell can contain only a single net point, it follows that in constant time per point of P , one can compute each point's nearest net point. We thus have the following.

Corollary 35.5.4. *For a set $P \subseteq \mathbb{R}^d$ of n points, and a parameter $r > 0$, one can compute, in linear time, an r -net of P , and furthermore, for each net point the set of points of P for which it is the nearest net point.*

In the following, a **weighted point** is a point that is assigned a positive integer weight. For any subset S of a weighted point set P , let $|S|$ denote the number of points in S and let $\omega(S) = \sum_{p \in S} \omega(p)$ denote the total weight of S .

In particular, **Corollary 35.5.4** implies that for a weighted point set one can compute the following quantity in linear time.

Algorithm 35.5.5 (net). *Given a weighted point set $P \subseteq \mathbb{R}^d$, let $\mathcal{N}(r, P)$ denote an r -net of P , where the weight of each net point p is the total sum of the weights of the points assigned to it. We slightly abuse notation, and also use $\mathcal{N}(r, P)$ to designate the algorithm computing this net, which has linear running time.*

35.5.3. Computing an r -net in a sparse graph

Given a $G = (V, E)$ be an edge-weighted graph with n vertices and m edges, and let $r > 0$ be a parameter. We are interested in the problem of computing an r -net for G . That is, a set of vertices of G that complies with **Definition 35.5.2**_{p225}.

35.5.3.1. The algorithm

We compute an r -net in a sparse graph using a variant of Dijkstra's algorithm with the sequence of starting vertices chosen in a random permutation.

Let π_i be the i th vertex in a random permutation π of V . For each vertex v we initialize $\delta(v)$ to $+\infty$. In the i th iteration, we test whether $\delta(\pi_i) \geq r$, and if so we do the following steps:

- (A) Add π_i to the resulting net \mathcal{N} .
- (B) Set $\delta(\pi_i)$ to zero.
- (C) Perform Dijkstra's algorithm starting from π_i , modified to avoid adding a vertex u to the priority queue unless its tentative distance is smaller than the current value of $\delta(u)$. When such a vertex u is expanded, we set $\delta(u)$ to be its computed distance from π_i , and relax the edges adjacent to u in the graph.

35.5.3.2. Analysis

While the analysis here does not directly use backward analysis, it is inspired to a large extent by such an analysis as in [Section 35.4p224](#).

Lemma 35.5.6. *The set \mathcal{N} is an r -net in G .*

Proof: By the end of the algorithm, each $v \in V$ has $\delta(v) < r$, for $\delta(v)$ is monotonically decreasing, and if it were larger than r when v was visited then v would have been added to the net.

An induction shows that if $\ell = \delta(v)$, for some vertex v , then the distance of v to the set \mathcal{N} is at most ℓ . Indeed, for the sake of contradiction, let j be the (end of) the first iteration where this claim is false. It must be that $\pi_j \in \mathcal{N}$, and it is the nearest vertex in \mathcal{N} to v . But then, consider the shortest path between π_j and v . The modified Dijkstra must have visited all the vertices on this path, thus computing $\delta(v)$ correctly at this iteration, which is a contradiction.

Finally, observe that every two points in \mathcal{N} have distance $\geq r$. Indeed, when the algorithm handles vertex $v \in \mathcal{N}$, its distance from all the vertices currently in \mathcal{N} is $\geq r$, implying the claim. ■

Lemma 35.5.7. *Consider an execution of the algorithm, and any vertex $v \in V$. The expected number of times the algorithm updates the value of $\delta(v)$ during its execution is $O(\log n)$, and more strongly the number of updates is $O(\log n)$ with high probability.*

Proof: For simplicity of exposition, assume all distances in G are distinct. Let S_i be the set of all the vertices $x \in V$, such that the following two properties both hold:

- (A) $d_G(x, v) < d_G(v, \Pi_i)$, where $\Pi_i = \{\pi_1, \dots, \pi_i\}$.
- (B) If $\pi_{i+1} = x$ then $\delta(v)$ would change in the $(i + 1)$ th iteration.

Let $s_i = |S_i|$. Observe that $S_1 \supseteq S_2 \supseteq \dots \supseteq S_n$, and $|S_n| = 0$.

In particular, let \mathcal{E}_{i+1} be the event that $\delta(v)$ changed in iteration $(i + 1)$ – we will refer to such an iteration as being *active*. If iteration $(i + 1)$ is active then one of the points of S_i is π_{i+1} . However, π_{i+1} has a uniform distribution over the vertices of S_i , and in particular, if \mathcal{E}_{i+1} happens then $s_{i+1} \leq s_i/2$, with probability at least half, and we will refer to such an iteration as being *lucky*. (It is possible that $s_{i+1} < s_i$ even if \mathcal{E}_{i+1} does not happen, but this is only to our benefit.) After $O(\log n)$ lucky iterations the set S_i is empty, and we are done. Clearly, if both the i th and j th iteration are active, the events that they are each lucky are independent of each other. By the Chernoff inequality, after $c \log n$ active iterations, at least $\lceil \log_2 n \rceil$ iterations were lucky with high probability, implying the claim. Here c is a sufficiently large constant. ■

Interestingly, in the above proof, all we used was the monotonicity of the sets S_1, \dots, S_n , and that if $\delta(v)$ changes in an iteration then the size of the set S_i shrinks by a constant factor with good probability in this iteration. This implies that there is some flexibility in deciding whether or not to initiate Dijkstra's algorithm from each vertex of the permutation, without damaging the number of times of the values of $\delta(v)$ are updated.

Theorem 35.5.8. *Given a graph $G = (V, E)$, with n vertices and m edges, the above algorithm computes an r -net of G in $O((n \log n + m) \log n)$ expected time.*

Proof: By [Lemma 35.5.7](#), the two δ values associated with the endpoints of an edge get updated $O(\log n)$ times, in expectation, during the algorithm's execution. As such, a single edge creates $O(\log n)$ decrease-key operations in the heap maintained by the algorithm. Each such operation takes constant time if we use Fibonacci heaps to implement the algorithm. ■

35.6. Bibliographical notes

Backwards analysis was invented/discovered by Raimund Seidel, and the **QuickSort** example is taken from Seidel [Sei93]. The number of changes of the minimum result of **Section 35.1** is by now folklore.

The closet-pair result in **Section 35.3** follows Golin *et al.* [GRSS95]. This is in turn a simplification of a result of Rabin [Rab76]. Smid provides a survey of such algorithms [Smi00].

The good ordering of **Section 35.4** is probably also folklore, although a similar idea was used by Mendel and Schwob [MS09] for a different problem. Computing nets in \mathbb{R}^d , which has nothing to do with backwards analysis, **Section 35.5.2**, is from Har-Peled and Raichel [HR13].

Computing a net in a sparse graph, **Section 35.5.3**, is from [EHS14]. While backwards analysis fails to hold in this case, it provides a good intuition for the analysis, which is slightly more complicated and indirect.

Chapter 36

Linear time algorithms

36.1. The lowest point above a set of lines

Let L be a set of n lines in the plane. To simplify the exposition, assume the lines are in general position:

- (A) No two lines of L are parallel.
- (B) No line of L is vertical or horizontal.
- (C) No three lines of L meet in a point.

We are interested in the problem of computing the point with the minimum y coordinate that is above all the lines of L . We consider a point on a line to be above it.

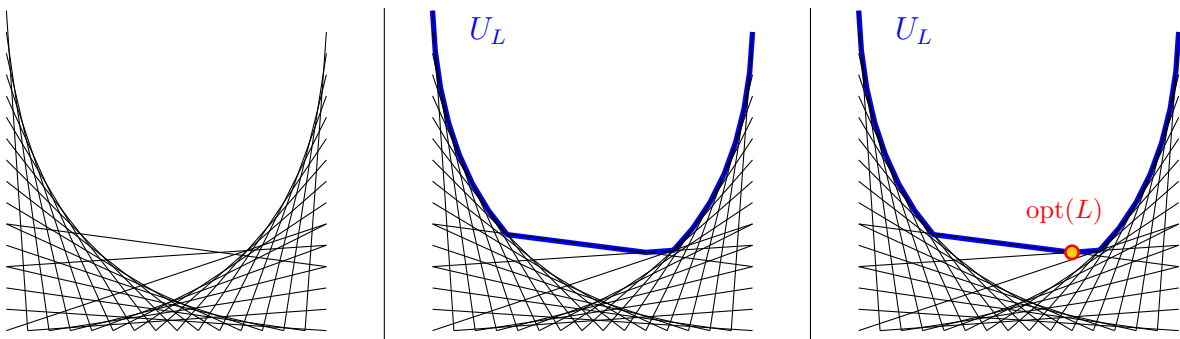


Figure 36.1: An input to the problem, the critical curve U_L , and the optimal solution – the point $\text{opt}(L)$.

For a line $\ell \in L$, and a value $\alpha \in \mathbb{R}$, let $\ell(x)$ be the value of ℓ at α . Formally, consider the intersection point of $p = \ell \cap (x = \alpha)$ (here, $x = \alpha$ is the vertical line passing through $(\alpha, 0)$). Then $\ell(x) = y(p)$.

Let $U_L(\alpha) = \max_{\ell \in L} \ell(\alpha)$ be the **upper envelope** of L . The function $U_L(\cdot)$ is convex, as one can easily verify. The problem asks to compute $y^* = \min_{x \in \mathbb{R}} U_L(x)$. Let x^* be the coordinate such that $y^* = U_L(x^*)$.

Definition 36.1.1. Let $\text{opt}(L) = (x^*, y^*)$ denote the optimal solution – that is, lowest point on $U_L(x)$.

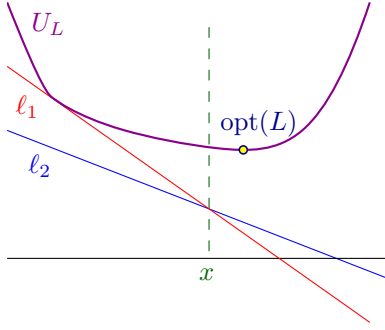


Figure 36.2: Illustration of the proof of [Lemma 36.1.4](#).

Remark 36.1.2. There are some uninteresting cases of this problem. For example, if all the lines of L have negative slope, then the solution is at $x^* = +\infty$. Similarly, if all the slopes are positive, then the solution is $x^* = -\infty$. We can easily check these cases in linear time. In the following, we assume that at least one line of L has positive slope, and at least one line has a negative slope.

Lemma 36.1.3. *Given a value x , and a set L of n lines, one can in linear time do the following:*

- (A) *Compute the value of $U_L(x)$.*
- (B) *Decide which one of the following happens: (I) $x = x^*$, (II) $x < x^*$, or (III) $x > x^*$.*

Proof: (A) Computing $\ell(x)$, for $x \in \mathbb{R}$, takes $O(1)$ time. Thus computing this value for all the lines of L takes $O(n)$ time, and the maximum can be computed in $O(n)$ time.

(B) For case (I) to happen, there must be two lines that realizes $U_L(x)$ – one of them has a positive slope, the other has negative slope. This clearly can be checked in linear time.

Otherwise, consider $U_L(x)$. If there is a single line that realizes the maximum for x , then its slope is the slope of $U_L(x)$ at x . If this slope is positive than $x^* < x$. If the slope is negative then $x < x^*$.

The slightly more challenging case is when two lines realizes the value of $U_L(x)$. That is $(x, U_L(x))$ is an intersection point of two lines of L (i.e., a **vertex**) on the upper envelope of the lines of L . Let ℓ_1, ℓ_2 be these two lines, and assume that $\text{slope}(\ell_1) < \text{slope}(\ell_2)$.

If $\text{slope}(\ell_2) < 0$, then both lines have negative slope, and $x^* > x$. If $\text{slope}(\ell_1) > 0$, then both lines have positive slope, and $x^* < x$. If $\text{slope}(\ell_1) < 0$, and $\text{slope}(\ell_1) > 0$, then this is case (I), and we are done. ■

Lemma 36.1.4. *Let (x, y) be the intersection point of two lines $\ell_1, \ell_2 \in L$, such that $\text{slope}(\ell_1) < \text{slope}(\ell_2)$, and $x < x^*$. Then $\text{opt}(L) = \text{opt}(L - \ell_1)$, where $L - \ell_1 = L \setminus \{\ell_1\}$*

Proof: See [Figure 36.2](#). Since $x < x^*$, it must be that $U_L(\cdot)$ has a negative slope at x (and also immediately to its right). In particular, for any $\alpha > x$, we have that $U_L(\alpha) \geq \ell_2(\alpha) > \ell_1(\alpha)$. That is, the line $\ell_1(x)$ is “buried” below ℓ_2 , and can not touch $U_L(\cdot)$ to the right of x . In particular, removing ℓ_1 from L can not change $U_L(\cdot)$ to the right of x . Furthermore, since $U_L(\cdot)$ has negative slope immediately after x , it implies that minimum point can not move by the deletion of ℓ_1 . Thus implying the claim. ■

Lemma 36.1.5. *Let (x, y) be the intersection point of two lines $\ell_1, \ell_2 \in L$, such that $\text{slope}(\ell_1) < \text{slope}(\ell_2)$, and $x^* < x$. Then $\text{opt}(L) = \text{opt}(L - \ell_2)$.*

Proof: Symmetric argument to the one used in the proof of [Lemma 36.1.4](#). ■

Observation 36.1.6. *The point $p = \text{opt}(L)$ is a vertex formed by the intersection of two lines of L . Indeed, since none of the lines of L are horizontal, if p was in the middle of a line, then we could move it and improve the value of the solution.*

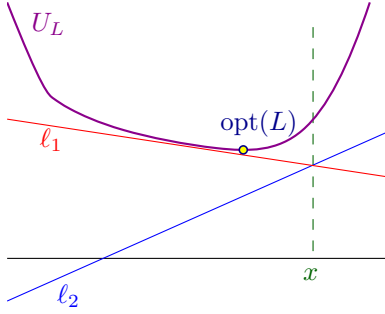


Figure 36.3: Illustration of the proof of [Lemma 36.1.5](#).

Lemma 36.1.7 (Prune). *Given a set L of n lines, one can compute, in linear time, either:*

- (A) *A set $L' \subseteq L$ such that $\text{opt}(L) = \text{opt}(L')$, and $|L'| \leq (7/8)|L|$.*
- (B) *A value x such that $x^*(L) = x$.*

Proof: If $|L| = n = O(1)$ then one can compute $\text{opt}(L)$ by brute force. Indeed, compute all the $\binom{n}{2}$ vertices induced by L , and for each one of them check if they define the optimal solution using the algorithm of [Lemma 36.1.3](#). This takes $O(1)$ time, as desired.

Otherwise, pair the lines of L in $N = \lfloor n/2 \rfloor$ pairs ℓ_i, ℓ'_i . For each pair, let x_i be the x -coordinate of the vertex $\ell_i \cap \ell'_i$. Compute, in linear time, using median selection, the median value z of x_1, \dots, x_N . For the sake of simplicity of exposition assume that $x_i < z$, for $i = 1, \dots, N/2 - 1$, and $x_i > z$, for $i = N/2 + 1, \dots, N$ (otherwise, reorder the lines and the values so that it happens).

Using the algorithm of [Lemma 36.1.3](#) decide which of the following happens:

- (I) $z = x^*$: we found the optimal solution, and we are done.
- (II) $z < x^*$. But then $x_i < z < x^*$, for $i = 1, \dots, N/2 - 1$. By [Lemma 36.1.4](#), either ℓ_i or ℓ'_i can be dropped without effecting the optimal solution, and which one can be dropped can be decided in $O(1)$ time. In particular, let L' be the set of lines after we drop a line from each such pair. We have that $\text{opt}(L') = \text{opt}(L)$, and $|L'| = n - (N/2 - 1) \leq (7/8)n$.
- (III) $z > x^*$. This case is handled symmetrically, using [Lemma 36.1.5](#). ■

Theorem 36.1.8. *Given a set L of n lines in the plane, one can compute the lowest point that is above all the lines of L (i.e., $\text{opt}(L)$) in linear time.*

Proof: The algorithm repeatedly apply the pruning algorithm of [Lemma 36.1.7](#). Clearly, by the above, this algorithm computes $\text{opt}(L)$ as desired.

In the i th iteration of this algorithm, if the set of lines has n_i lines, then this iteration takes $O(n_i)$ time. However, $n_i \leq (7/8)^i n$. In particular, the overall running time of the algorithm is

$$O\left(\sum_{i=0}^{\infty} (7/8)^i n\right) = O(n). \quad \blacksquare$$

36.2. Bibliographical notes

The algorithm presented in [Section 36.1](#) is a simplification of the work of Megiddo [[Meg84](#)]. Megiddo solved the much harder problem of solving linear programming in constant dimension in linear time, The algorithm presented is essentially the core of his basic algorithm.

Chapter 37

Streaming

I don't know why it should be, I am sure; but the sight of another man asleep in bed when I am up, maddens me. It seems to me so shocking to see the precious hours of a man's life - the priceless moments that will never come back to him again - being wasted in mere brutish sleep.

Jerome K. Jerome, Three men in a boat

37.1. How to sample a stream

Imagine that you are given a stream of elements s_1, s_2, \dots , and you need to sample k numbers from this stream (say, without repetition) – assume that you do not know the length of the stream in advance, and furthermore, you have only $O(k)$ space available for you. How to do that efficiently?

There are two natural schemes:

- (A) Whenever an element arrives, generate a random number for it in the range $[0, 1]$. Maintain a heap with the k elements with the lowest priority. Implemented naively this requires $O(\log k)$ comparisons after each insertion, but it is not difficult to improve this to $O(1)$ comparisons in the amortized sense per insertion. Clearly, the resulting set is the desired random sample
- (B) Let S_t be the random sample maintained in the t th iteration. When the i th element arrives, the algorithm flip a coin that is heads with probability $\min(1, k/i)$. If the coin is heads then it inserts s_i to S_{i-1} to get S_i . If S_{i-1} already have k elements, then first randomly delete one of the elements.

Theorem 37.1.1. *Given a stream of elements, one can uniformly sample k elements (without repetition), from the stream using $O(k)$ space, where $O(1)$ time is spent for handling each incoming element.*

Proof: We implement the scheme (B) above. We only need to argue that this is a uniform random sample. The claim trivially hold for $i = k$. So assume the claim holds for $i < t$, and we need to prove that the set after getting t th element is still a uniform random sample.

So, consider a specific set $K \subseteq \{s_1, \dots, s_t\}$ of k elements. The probability of K to be a random sample of size k from a set of t elements is $1/\binom{t}{k}$. We need to argue that this probability remains the same for this scheme.

So, if $s_t \notin K$, then we have

$$\mathbb{P}[K = S_t] = \mathbb{P}[K = S_{t-1} \text{ and } s_t \text{ was not inserted}] = \frac{1}{\binom{t-1}{k}} \left(1 - \frac{k}{t}\right) = \frac{k!(t-1-k)!(t-k)}{(t-1)!t} = \frac{1}{\binom{t}{k}}.$$

If $s_t \in K$, then

$$\mathbb{P}[K = S_t] = \mathbb{P} \left[\begin{array}{l} K \setminus \{s_t\} \subseteq S_{t-1}, \\ s_t \text{ was inserted} \\ \text{and } S_{t-1} \setminus K \text{ thrown out of } S_{t-1} \end{array} \right] = \frac{t-1-(k-1)}{\binom{t-1}{k}} \frac{1}{t} \frac{1}{k} = \frac{(t-k)k!(t-1-k)!}{(t-1)!t} = \frac{1}{\binom{t}{k}},$$

as desired. Indeed, there are $t-1-(k-1)$ subsets of size k of $\{s_1, \dots, s_{t-1}\}$ that contains $K \setminus \{s_t\}$ – since we fix $k-1$ of the $t-1$ elements. ■

37.2. Sampling and median selection

Let $B[1, \dots, n]$ be a set of n numbers. We would like to estimate the median, without computing it outright. A natural idea, would be to pick k elements e_1, \dots, e_k randomly from B , and return their median as the guess for the median of B .

In the following, let $R_B(t)$ be the t th smallest number in the array B .

Observation 37.2.1. For $\varepsilon \in (0, 1)$, we have that $\frac{1}{1-\varepsilon} \geq 1 + \varepsilon$.

Lemma 37.2.2. Let $\varepsilon \in (0, 1/2)$, and let $k = \lceil \frac{12}{\varepsilon^2} \ln \frac{2}{\delta} \rceil$. Let Z be the median of the random sample of B of size k . We have that

$$\mathbb{P}\left[R_B\left(\frac{1-\varepsilon}{2}n\right) \leq Z \leq R_B\left(\frac{1+\varepsilon}{2}n\right)\right] \geq 1 - \delta.$$

Namely, with probability at least $1 - \delta$, the returned value Z is $(\varepsilon/2)n$ positions away from the true median.

Proof: Let $L = R_B((1-\varepsilon)n/2)$. Let $X_i = 1$ if and only if $e_i \leq L$. We have that

$$\mathbb{P}[X_i = 1] = \frac{(1-\varepsilon)n/2}{n} = \frac{1-\varepsilon}{2}.$$

As such, setting $Y = \sum_{i=1}^k X_i$, we have

$$\mu = \mathbb{E}[Y] = \frac{1-\varepsilon}{2}k \geq \frac{k}{4} \geq \frac{3}{\varepsilon^2} \ln \frac{2}{\delta}.$$

One case is that the algorithm fails, if $Y \geq k/2$. We have that

$$\mathbb{P}[Y \geq k/2] = \mathbb{P}\left[Y \geq \frac{1/2}{(1-\varepsilon)/2} \cdot \frac{1-\varepsilon}{2}k\right] = \mathbb{P}[Y \geq (1+\varepsilon)\mu] \leq \exp(-\varepsilon^2\mu/3) \leq \exp\left(-\varepsilon^2 \cdot \frac{3}{\varepsilon^2} \ln \frac{2}{\delta}\right) \leq \frac{\delta}{2}.$$

by Chernoff's inequality (see [Theorem 37.5.1](#)).

This implies that $\mathbb{P}[R_B((1-\varepsilon)n/2) > Z] \leq \delta/2$.

The claim now follows by realizing that by symmetry (i.e., revering the order), we have that $\mathbb{P}[Z > R_B((1+\varepsilon)n/2)] \leq \delta/2$, and putting these two inequalities together. \blacksquare

The above already implies that we can get a good estimate for the median. We need something somewhat stronger – we state it without proof since it follows by similarly mucking around with Chernoff's inequality.

Lemma 37.2.3. Let $\varepsilon \in (0, 1/2)$, let B an array of n elements, and let $S = \{e_1, \dots, e_k\}$ be a set of k samples picked uniformly and randomly from B . Then, for some absolute constant c , and an integer k , such that $k \geq \lceil \frac{c}{\varepsilon^2} \ln \frac{1}{\delta} \rceil$, we have that

$$\mathbb{P}[R_S(k_-) \leq R_B(n/2) \leq R_S(k^+)] \geq 1 - \delta.$$

for $k_- = \lfloor (1-\varepsilon)k/2 \rfloor$, and $k^+ = \lfloor (1+\varepsilon)k/2 \rfloor$.

One can prove even a stronger statement:

$$\mathbb{P}[R_B((1-2\varepsilon)n/2) \leq R_S((1-\varepsilon)k/2) \leq R_B(n/2) \leq R_S((1+\varepsilon)k/2) \leq R_B((1+2\varepsilon)n/2)] \geq 1 - \delta$$

(the constant c would have to be slightly bigger).

37.2.1. A median selection with few comparisons

The above suggests a natural algorithm for computing the median (i.e., the element of rank $n/2$ in B). Pick a random sample S of $k = O(\sqrt{n} \log n)$ elements. Next, sort S , and pick the elements L and R of ranks $(1 - \varepsilon)k$ and $(1 + \varepsilon)k$ in S , respectively. Next, scan the elements, and compare them to L and R , and keep only the elements that are between. In the end of this process, we have computed:

(A) α : The rank of the number L in the set B .

(B) $T = \{x \in B \mid L \leq x \leq H\}$.

Compute, by brute force (i.e., sorting) the element of rank $n/2 - \alpha$ in T . Return it as the desired median. If $n/2 - \alpha$ is negative, then the algorithm failed, and it tries again.

Lemma 37.2.4. *The above algorithm performs $2n + O(n^{3/4} \log n)$ comparisons, and reports the median. This holds with high probability.*

Proof: Set $\varepsilon = 1/n^{1/4}$, and $\delta = 1/n^{O(1)}$, and observe that [Lemma 37.2.3](#) implies that with probability $\geq 1 - 1/\delta$, we have that the desired median is between L and H . In addition, [Lemma 37.2.3](#) also implies that $|T| \leq 4\varepsilon n \leq 4n^{3/4}$, which readily implies the correctness of the algorithm.

As for the bound on the number of comparisons, we have, with high probability, that the number of comparisons is

$$O(|S| \log |S| + |T| \log |T|) + 2n = O(\sqrt{n} \log^2 n + n^{3/4} \log n) + 2n,$$

since deciding if an element is between L and H requires two comparisons. ■

Lemma 37.2.5. *The above algorithm can be modified to perform $(3/2)n + O(n^{3/4} \log n)$ comparisons, and reports the median correctly. This holds with high probability.*

Proof: The trick is to randomly compare each element either first to L or first to H with equal probability. For elements that are either smaller than L or bigger than H , this requires $(3/2)n$ comparisons in expectation. Thus improving the bound from $2n$ to $(3/2)n$. ■

Remark 37.2.6. Note, that if we know, as in this case, that L and H are in the middle, than it is not needed to do the random comparisons trick used above – indeed, just regular algorithm would work. This trick makes sense only if do not know the rank of L and H in the real array, but only know that they are close together. Then, the random comparisons trick does work better than the deterministic approach.

Lemma 37.2.7. *Consider a stream B of n numbers, and assume we can make two passes over the data. Then, one can compute exactly the median of B using:*

(I) $O(n^{3/4})$ space.

(II) $1.5n + O(n^{3/4} \log n)$ comparisons.

The algorithm reports the median correctly, and it succeeds with high probability.

Proof: Implement the above algorithm, using the random sampling from [Theorem 37.1.1](#). ■

37.3. Big data and the streaming model

Here, we are interested in doing some computational tasks when the amount of data we have to handle is quite large (think terabytes or larger). The main challenge in many of these cases is that even reading the data once is expensive. Running times of $O(n \log n)$ might not be acceptable. Furthermore, in many cases, we can load all the data into memory.

In the *streaming* model, one reads the data as it comes in, but one can not afford to keep all the data. A natural example would be a internet router, which has gazillion of packets going through it every minute. We might still be interested in natural questions about these packets, but we want to do this without storing all the packets.

37.4. Heavy hitters

Imagine a stream s_1, \dots , where elements might repeat, and we would like to maintain a list of elements that appear at least εn times. We present a simple but clever scheme that maintains such a list.

The algorithm. To this end, let

$$k = \lceil 1/\varepsilon \rceil.$$

At each point in time, we maintain a set S of k elements, with a counter for each element. Let S_t be the version of S after t were inserted. When s_{t+1} arrives, we increase its counter if it is already in S_t . If $|S_t| < k$, then we just insert s_{t+1} to the set, and set its counter to 1. Otherwise, $|S_t| = k$ and $s_{t+1} \notin S_t$. We then decrease all the k counters of elements in S_t by 1. If a counter of an element in S_{t+1} is zero, then we delete it from the set.

37.5. Chernoff inequality

Proving the specific form of Chernoff's inequality we need is outside our scope. The interested reader is referred to notes here <https://sarielhp.org/p/notes/16/chernoff/chernoff.pdf>. We next state what we need:

Theorem 37.5.1. *Let X_1, \dots, X_n be n independent Bernoulli trials, where $\mathbb{P}[X_i = 1] = p_i$, and $\mathbb{P}[X_i = 0] = 1 - p_i$, for $i = 1, \dots, n$. Let $X = \sum_{i=1}^n X_i$, and $\mu = \mathbb{E}[X] = \sum_i p_i$. For $\delta \in (0, 1)$, we have*

$$\mathbb{P}[X > (1 + \delta)\mu] < \exp(-\mu\delta^2/3).$$

Part X

Exercises

Chapter 38

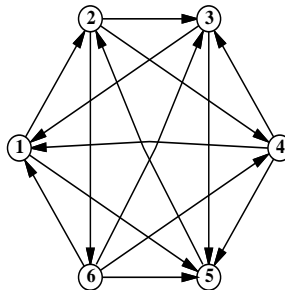
Exercises - Prerequisites

This chapter include problems that are perquisite. Their main purpose is to check whether you are read to take the 473 algorithms class. If you do not have the prerequisites it is *your responsibility* to fill in the missing gaps in your education.

38.1. Graph Problems

1 A trip through the graph. (20 PTS.)

A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once. Prove that every tournament contains at least one Hamiltonian path.



A six-vertex tournament containing the Hamiltonian path $6 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1$.

2 Graphs! Graphs! (20 PTS.)

A *coloring* of a graph G by α colors is an assignment to each vertex of G a color which is an integer between 1 and α , such that no two vertices that are connected by an edge have the same color.

- 2.A.** (5 PTS.) *Prove or disprove* that if in a graph G the maximum degree is k , then the vertices of the graph can be colored using $k + 1$ colors.

- 2.B.** (5 PTS.) Provide an efficient coloring algorithm for a graph G with n vertices and m edges that uses at most $k+1$ colors, where k is the maximum degree in G . What is the running time of your algorithm, if the graph is provided using adjacency lists. What is the running time of your algorithm if the graph is given with an adjacency matrix. (Note, that your algorithm should be as fast as possible.)
- 2.C.** (5 PTS.) A directed graph $G = (V, E)$ is a *neat* graph if there exist an ordering of the vertices of the graph $V(G) = \langle v_1, v_2, \dots, v_n \rangle$ such that if the edge (v_i, v_j) is in $E(G)$ then $i < j$. Prove (by induction) that a DAG (i.e., directed acyclic graph) is a neat graph.
- 2.D.** (5 PTS.) A cut (S, T) in a directed graph $G = (V, E)$ is a partition of V into two disjoint sets S and T . A cut is *mixed* if there exists $s, s' \in S$ and $t, t' \in T$ such that $(s, t) \in E$ and $(t', s') \in E$. Prove that if all the non-trivial cuts (i.e., neither S nor T are empty) are mixed then the graph is not a neat graph.

3 Mad Cow Disease (20 PTS.)

In a land far far away (i.e., Canada), a mad cow disease was spreading among cow farms. The cow farms were, naturally, organized as a $n \times n$ grid. The epidemic started when m contaminated cows were delivered to (some) of the farms. Once one cow in a farm has Mad Cow disease then all the cows in this farm get the disease. For a farm, if two or more of its neighboring farms have the disease then the cows in the farm would get the disease. A farm in the middle of the grid has four neighboring farms (two horizontally next to it, and two vertically next to it). We are interested in how the disease spread if we wait enough time.

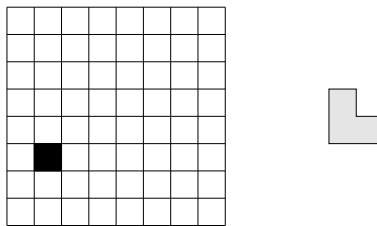
- (5 PTS.) Show that if $m = n$ then there is a scenario such that all the farms in the $n \times n$ grid get contaminated.
- (15 PTS.) Prove that if $m \leq n - 1$ then (always) not all the farms are contaminated.

4 Connectivity and walking. (10 PTS.)

- 4.A.** Use induction to prove that in a simple graph, every walk between a pair of vertices, u, v , contains a path between u and v . Recall that a walk is a list of the form $v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k$, in which e_i has endpoints v_{i-1} and v_i .
- 4.B.** Prove that a graph is connected if and only if for every partition of its vertices into two nonempty sets, there exists an edge that has endpoints in both sets.

5 Chessboard (10 PTS.)

Consider a $2^n \times 2^n$ chessboard with one (arbitrarily chosen) square removed, as in the following picture (for $n = 3$):



Prove that any such chessboard can be tiled without gaps or overlaps by L-shapes consisting of 3 squares each.

6 Coloring (10 PTS.)

- 6.A.** (5 PTS.) Let T_1, T_2 and T_3 be three trees defined over the set of vertices $\{v_1, \dots, v_n\}$. Prove that the graph $G = T_1 \cup T_2 \cup T_3$ is colorable using six colors (e is an edge of G if and only if it is an edge in one of trees T_1, T_2 and T_3).

6.B. (5 PTS.) Describe an efficient algorithm for computing this coloring. What is the running time of your algorithm?

7 **Binary trees and codes.** Professor George O’Jungle has a favorite 26-node binary tree, whose nodes are labeled by letters of the alphabet. The preorder and postorder sequences of nodes are as follows:

preorder: M N H C R S K W T G D X I Y A J P O E Z V B U L Q F
 postorder: C W T K S G R H D N A O E P J Y Z I B Q L F U V X M

Draw Professor O’Jungle’s binary tree, and give the in-order sequence of nodes.

38.2. Recurrences

8 **Recurrences.** (20 PTS.)

Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don’t* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied. More exact solutions are better.

- 8.A. (2 PTS.) $A(n) = A(\sqrt{n}/3 + \lfloor \log n \rfloor) + n$
- 8.B. (2 PTS.) $B(n) = \min_{0 < k < n} (3 + B(k) + B(n - k))$.
- 8.C. (2 PTS.) $C(n) = 3C(\lceil n/2 \rceil - 5) + n/\log n$
- 8.D. (2 PTS.) $D(n) = \frac{n}{n-3}D(n-1) + 1$
- 8.E. (2 PTS.) $E(n) = E(\lfloor 3n/4 \rfloor) + \sqrt{n}$
- 8.F. (2 PTS.) $F(n) = F(\lfloor \log n \rfloor) + \log n$ (**HARD**)
- 8.G. (2 PTS.) $G(n) = n + \lfloor \sqrt{n} \rfloor \cdot G(\lfloor \sqrt{n} \rfloor)$
- 8.H. (2 PTS.) $H(n) = \log(H(n-1)) + 1$
- 8.I. (2 PTS.) $I(n) = 5I(\lfloor \sqrt{n} \rfloor) + 1$
- 8.J. (2 PTS.) $J(n) = 3J(n/4) + 1$

9 **Recurrences II** (20 PTS.)

Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don’t* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied. More exact solutions are better.

- 9.A. (1 PTS.) $A(n) = A(n/3 + 5 + \lfloor \log n \rfloor) + n \log \log n$
- 9.B. (1 PTS.) $B(n) = \min_{0 < k < n} (3 + B(k) + B(n - k))$.
- 9.C. (1 PTS.) $C(n) = 3C(\lceil n/2 \rceil - 5) + n/\log n$
- 9.D. (1 PTS.) $D(n) = \frac{n}{n-5}D(n-1) + 1$
- 9.E. (1 PTS.) $E(n) = E(\lfloor 3n/4 \rfloor) + 1/\sqrt{n}$
- 9.F. (1 PTS.) $F(n) = F(\lfloor \log^2 n \rfloor) + \log n$ (**HARD**)
- 9.G. (1 PTS.) $G(n) = n + 7\sqrt{n} \cdot G(\lfloor \sqrt{n} \rfloor)$
- 9.H. (1 PTS.) $H(n) = \log^2(H(n-1)) + 1$
- 9.I. (1 PTS.) $I(n) = I(\lfloor n^{1/4} \rfloor) + 1$

9.J. (1 PTS.) $J(n) = J(n - \lfloor n/\log n \rfloor) + 1$

10 Recurrences III (20 PTS.)

Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied.

10.A. $A(n) = A(n/2) + n$

10.B. $B(n) = 2B(n/2) + n$

10.C. $C(n) = n + \frac{1}{2}(C(n-1) + C(3n/4))$

10.D. $D(n) = \max_{n/3 < k < 2n/3} (D(k) + D(n-k) + n)$ (**HARD**)

10.E. $E(n) = 2E(n/2) + n/\lg n$ (**HARD**)

10.F. $F(n) = \frac{F(n-1)}{F(n-2)}$, where $F(1) = 1$ and $F(2) = 2$. (**HARD**)

10.G. $G(n) = G(n/2) + G(n/4) + G(n/6) + G(n/12) + n$ [Hint: $\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} = 1$.] (**HARD**)

10.H. $H(n) = n + \sqrt{n} \cdot H(\sqrt{n})$ (**HARD**)

10.I. $I(n) = (n-1)(I(n-1) + I(n-2))$, where $F(0) = F(1) = 1$ (**HARD**)

10.J. $J(n) = 8J(n-1) - 15J(n-2) + 1$

11 Evaluate summations. (10 PTS.)

Evaluate the following summations; simplify your answers as much as possible. Significant partial credit will be given for answers in the form $\Theta(f(n))$ for some recognizable function $f(n)$.

11.A. (2 PTS.) $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{i}$
(**HARD**)

11.B. (2 PTS.) $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{j}$

11.C. (2 PTS.) $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{k}$

11.D. (2 PTS.) $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j \frac{1}{k}$

11.E. (2 PTS.) $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j \frac{1}{j \cdot k}$

12 Simplify binary formulas. This problem asks you to simplify some recursively defined boolean formulas as much as possible. In each case, prove that your answer is correct. Each proof can be just a few sentences long, but it must be a *proof*.

12.A. Suppose $\alpha_0 = p$, $\alpha_1 = q$, and $\alpha_n = (\alpha_{n-2} \wedge \alpha_{n-1})$ for all $n \geq 2$. Simplify α_n as much as possible. [Hint: What is α_5 ?]

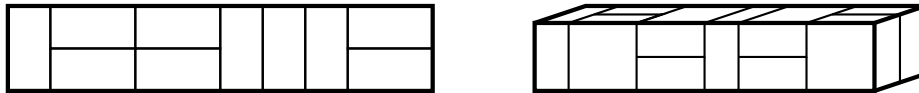
12.B. Suppose $\beta_0 = p$, $\beta_1 = q$, and $\beta_n = (\beta_{n-2} \leftrightarrow \beta_{n-1})$ for all $n \geq 2$. Simplify β_n as much as possible. [Hint: What is β_5 ?]

- 12.C. Suppose $\gamma_0 = p$, $\gamma_1 = q$, and $\gamma_n = (\gamma_{n-2} \Rightarrow \gamma_{n-1})$ for all $n \geq 2$. Simplify γ_n as much as possible. [Hint: What is γ_5 ?]
- 12.D. Suppose $\delta_0 = p$, $\delta_1 = q$, and $\delta_n = (\delta_{n-2} \bowtie \delta_{n-1})$ for all $n \geq 2$, where \bowtie is some boolean function with two arguments. Find a boolean function \bowtie such that $\delta_n = \delta_m$ if and only if $n - m$ is a multiple of 4. [Hint: There is only one such function.]

38.3. Counting

13 Counting dominos

- 13.A. A *domino* is a 2×1 or 1×2 rectangle. How many different ways are there to completely fill a $2 \times n$ rectangle with n dominos? Set up a recurrence relation and give an *exact* closed-form solution.
- 13.B. A *slab* is a three-dimensional box with dimensions $1 \times 2 \times 2$, $2 \times 1 \times 2$, or $2 \times 2 \times 1$. How many different ways are there to fill a $2 \times 2 \times n$ box with n slabs? Set up a recurrence relation and give an *exact* closed-form solution.



A 2×10 rectangle filled with ten dominos, and a $2 \times 2 \times 10$ box filled with ten slabs.

38.4. O notation and friends

14 Sorting functions (20 PTS.)

Sort the following 25 functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice.

$n^{4.5} - (n-1)^{4.5}$	n	$n^{2.1}$	$\lg^*(n/8)$	$1 + \lg \lg \lg n$
$\cos n + 2$	$\lg(\lg^* n)$	$(\lg n)!$	$(\lg^* n)^{\lg n}$	n^5
$\lg^* 2^{2^{2^n}}$	$2^{\lg n}$	$\sqrt[n]{n^e}$	$\sum_{i=1}^n i$	$\sum_{i=1}^n i^2$
$n^{7/(2n)}$	$n^{3/(2 \lg n)}$	$12 + \lfloor \lg \lg(n) \rfloor$	$(\lg(2+n))^{\lg n}$	$(1 + \frac{1}{154})^{15n}$
$n^{1/\lg \lg n}$	$n^{\lg \lg n}$	$\lg^{(201)} n$	$n^{1/125}$	$n(\lg n)^4$

To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2 , n , $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$. [Hint: When considering two functions $f(\cdot)$ and $g(\cdot)$ it is sometime useful to consider the functions $\ln f(\cdot)$ and $\ln g(\cdot)$.]

15 Sorting functions II (20 PTS.)

Sort the following 25 functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do

them anyway just for practice.

$n^{5.5} - (n-1)^{5.5}$	n	$n^{2.2}$	$\lg^*(n/7)$	$1 + \lg \lg n$
$\cos n + 2$	$\lg(\lg^* n)$	$\lg(n!)$	$(\lg^* n)^{\lg n}$	n^4
$\lg^* 2^{2^n}$	$2^{\lg^* n}$	$e^{\sqrt{n}}$	$\sum_{i=1}^n \frac{1}{i}$	$\sum_{i=1}^n \frac{1}{i^2}$
$n^{3/(2n)}$	$n^{3/(2 \lg n)}$	$\lfloor \lg \lg(n!) \rfloor$	$(\lg(7+n))^{\lg n}$	$(1 + \frac{1}{154})^{154n}$
$n^{1/\lg \lg n}$	$n^{\lg \lg n}$	$\lg^{(200)} n$	$n^{1/1234}$	$n(\lg n)^3$

To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2 , n , $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

16 *O* notation revisited. (10 PTS.)

- 16.A.** Let $f_i(n)$ be a sequence of functions, such that for every i , $f_i(n) = o(\sqrt{n})$ (namely, $\lim_{n \rightarrow \infty} \frac{f_i(n)}{\sqrt{n}} = 0$). Let $g(n) = \sum_{i=1}^n f_i(n)$. Prove or disprove: $g(n) = o(n^{3/2})$.
- 16.B.** If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Prove or disprove:
- $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$
 - $f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$
 - $f_1(n)^{f_2(n)} = O(g_1(n)^{g_2(n)})$

17 Some proofs required.

- 17.A.** Prove that $2^{\lceil \lg n \rceil + \lfloor \lg n \rfloor} = \Theta(n^2)$.
- 17.B.** Prove or disprove: $2^{\lfloor \lg n \rfloor} = \Theta(2^{\lceil \lg n \rceil})$.
- 17.C.** Prove or disprove: $2^{2^{\lfloor \lg \lg n \rfloor}} = \Theta(2^{2^{\lceil \lg \lg n \rceil}})$.
- 17.D.** Prove or disprove: If $f(n) = O(g(n))$, then $\log(f(n)) = O(\log(g(n)))$.
- 17.E.** Prove or disprove: If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.
(HARD)
- 17.F.** Prove that $\log^k n = o(n^{1/k})$ for any positive integer k .

38.5. Probability

18 Balls and boxes. (20 PTS.)

There are n balls (numbered from 1 to n) and n boxes (numbered from 1 to n). We put each ball in a randomly selected box.

- 18.A.** (4 PTS.) A box may contain more than one ball. Suppose X is the number on the box that has the smallest number among all nonempty boxes. What is the expectation of X ?
- 18.B.** (4 PTS.) What is the expected number of bins that have exactly one ball in them? (Hint: Compute the probability of a specific bin to contain exactly one ball and then use some properties of expectation.)
- 18.C.** (8 PTS.) We put the balls into the boxes in such a way that there is exactly one ball in each box. If the number written on a ball is the same as the number written on the box containing the ball, we say there is a match. What is the expected number of matches?
- 18.D.** (4 PTS.) What is the probability that there are exactly k matches? ($1 \leq k < n$)

[Hint: If you have to appeal to “intuition” or “common sense”, your answers are probably wrong!]

19 Idiotic Sort (20 PTS.)

There is an array A with n unsorted distinct numbers in it. $\text{IDIOTICSORT}(A)$ sorts the array using an iterative algorithm. In each iteration, it picks randomly (and uniformly) two indices i, j in the ranges $\{1, \dots, n\}$. Next, if $A[\min(i, j)] > A[\max(i, j)]$ it swaps $A[i]$ and $A[j]$. The algorithm magically stop once the array is sorted.

- 19.A. (5 PTS.) Prove that after (at most) $n!$ swaps performed by the algorithm, the array A is sorted.
- 19.B. (5 PTS.) Prove that after at most (say) $6n^3$ swaps performed by the algorithm, the array A is sorted. (There might be an easy solution, but I don't see it.)
- 19.C. (5 PTS.) Prove that if A is not sorted, than the probability for a swap in the next iteration is at least $\geq 2/n^2$.
- 19.D. (5 PTS.) Prove that if A is not sorted, then the expected number of iterations till the next swap is $\leq n^2/2$. [Hint: use geometric random variable.]
- 19.E. (5 PTS.) Prove that the expected number of iterations performed by the algorithm is $O(n^5)$. [Hint: Use linearity of expectation.]

20 Random walk. (10 PTS.)

A *random walk* is a walk on a graph G , generated by starting from a vertex $v_0 = v \in V(G)$, and in the i -th stage, for $i > 0$, randomly selecting one of the neighbors of v_{i-1} and setting v_i to be this vertex. A walk v_0, v_1, \dots, v_m is of length m .

- 20.A. For a vertex $u \in V(G)$, let $P_u(m, v)$ be the probability that a random walk of length m , starting from u , visits v (i.e., $v_i = v$ for some i).
Prove that a graph G with n vertices is connected, if and only if, for any two vertices $u, v \in V(G)$, we have $P_u(n-1, v) > 0$.
- 20.B. Prove that a graph G with n vertices is connected if and only if for any pair of vertices $u, v \in V(G)$, we have $\lim_{m \rightarrow \infty} P_u(m, v) = 1$.

21 Random Elections. (10 PTS.)

You are in a shop trying to buy green tea. There n different types of green tea that you are considering: T_1, \dots, T_n . You have a coin, and you decide to randomly choose one of them using random coin flips. Because of the different prices of the different teas, you decide that you want to choose the i th tea with probability p_i (of course, $\sum_{i=1}^n p_i = 1$).

Describe an algorithm that chooses a tea according to this distribution, using only coin flips. Compute the expected number of coin flips your algorithm uses. (Your algorithm should minimize the number of coin flips it uses, since if you flip coins too many times in the shop, you might be arrested.)

22 Runs? (10 PTS.)

We toss a fair coin n times. What is the expected number of “runs”? Runs are consecutive tosses with the same result. For example, the toss sequence HHHTTHTH has 5 runs.

23 A card game. Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, \dots , 52 of clubs. (They're big cards.) Penn shuffles the deck until each of the $52!$ possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the five of clubs.

- 23.A. On average, how many cards does Penn give Teller?
- 23.B. On average, what is the smallest-numbered card that Penn gives Teller? (**HARD**)

23.C. On average, what is the largest-numbered card that Penn gives Teller?

[Hint: Solve for an n -card deck and then set $n = 52$.] In each case, give *exact* answers and prove that they are correct. If you have to appeal to “intuition” or “common sense”, your answers are probably wrong!

24 Alice and Bob Alice and Bob each have a fair n -sided die. Alice rolls her die once. Bob then repeatedly throws his die until he rolls a number at least as big as the number Alice rolled. Each time Bob rolls, he pays Alice \$1. (For example, if Alice rolls a 5, and Bob rolls a 4, then a 3, then a 1, then a 5, the game ends and Alice gets \$4. If Alice rolls a 1, then no matter what Bob rolls, the game will end immediately, and Alice will get \$1.)

Exactly how much money does Alice expect to win at this game? Prove that your answer is correct. If you have to appeal to ‘intuition’ or ‘common sense’, your answer is probably wrong!

38.6. Basic data-structures and algorithms

25 Storing temperatures. (10 PTS.)

Describe a data structure that supports storing temperatures. The operations on the data structure are as follows:

Insert(t, d) — Insert the temperature t that was measured on date d . Each temperature is a real number between -100 and 150 . For example, `insert(22, "1/20/03")`.

Average(d_1, d_2) report what is the average of all temperatures that were measured between date d_1 and date d_2 .

Each operation should take time $O(\log n)$, where n is the number of dates stored in the data structure. You can assume that a date is just an integer which specifies the number of days since the first of January 1970.

26 Binary search tree modifications. (10 PTS.)

Suppose we have a binary search tree. You perform a long sequence of operations on the binary tree (insertion, deletions, searches, etc), and the maximum depth of the tree during those operations is at most h .

Modify the binary search tree T so that it supports the following operations. Implementing some of those operations would require you to modify the information stored in each node of the tree, and the way insertions/deletions are being handled in the tree. For each of the following, describe separately the changes made in detail, and the algorithms for answering those queries. (Note, that under the modified version of the binary search tree, insertion and deletion should still take $O(h)$ time, where h is the maximum height of the tree during all the execution of the algorithm.)

26.A. (2 PTS.) Find the smallest element stored in T in $O(h)$ time.

26.B. (2 PTS.) Given a query k , find the k -th smallest element stored in T in $O(h)$ time.

26.C. (3 PTS.) Given a query $[a, b]$, find the number of elements stored in T with their values being in the range $[a, b]$, in $O(h)$ time.

26.D. (3 PTS.) Given a query $[a, b]$, report (i.e., printout) all the elements stored in T in the range $[a, b]$, in $O(h + u)$ time, where u is the number of elements printed out.

27 Euclid revisited. (10 PTS.)

Prove that for any nonnegative parameters a and b , the following algorithms terminate and produce identical output. Also, provide bounds on the running times of those algorithms. Can you imagine any reason why WEIRDEUCLID would be preferable to FASTEUCLID?

```

SlowEuclid(a, b) :
  if b > a
    return SLOWEUCLID(b, a)
  else if b = 0
    return a
  else
    return SLOWEUCLID(b, a - b)

```

```

FastEuclid(a, b) :
  if b = 0
    return a
  else
    return FASTEUCLID(b, a mod b)

```

```

WeirdEuclid(a, b) :
  if b = 0
    return a
  if a = 0
    return b
  if a is even and b is even
    return 2*WEIRDEUCLID(a/2, b/2)
  if a is even and b is odd
    return WEIRDEUCLID(a/2, b)
  if a is odd and b is even
    return WEIRDEUCLID(a, b/2)
  if b > a
    return WEIRDEUCLID(b - a, a)
  else
    return WEIRDEUCLID(a - b, b)

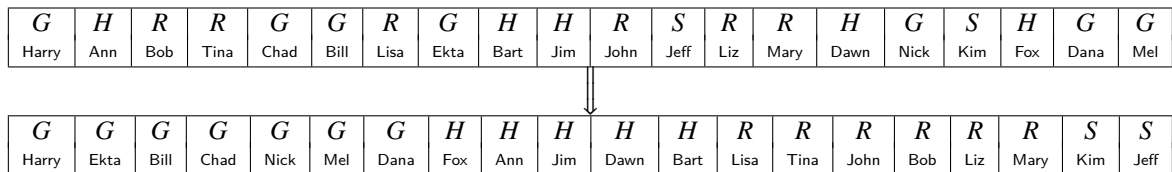
```

28 This despicable sorting hat trick.

Every year, upon their arrival at Hogwarts School of Witchcraft and Wizardry, new students are sorted into one of four houses (Gryffindor, Hufflepuff, Ravenclaw, or Slytherin) by the Hogwarts Sorting Hat. The student puts the Hat on their head, and the Hat tells the student which house they will join. This year, a failed experiment by Fred and George Weasley filled almost all of Hogwarts with sticky brown goo, mere moments before the annual Sorting. As a result, the Sorting had to take place in the basement hallways, where there was so little room to move that the students had to stand in a long line.

After everyone learned what house they were in, the students tried to group together by house, but there was too little room in the hallway for more than one student to move at a time. Fortunately, the Sorting Hat took CS Course many years ago, so it knew how to group the students as quickly as possible. What method did the Sorting Hat use?

- 28.A.** More formally, you are given an array of n items, where each item has one of four possible values, possibly with a pointer to some additional data. Describe an algorithm^① that rearranges the items into four clusters in $O(n)$ time using only $O(1)$ extra space.



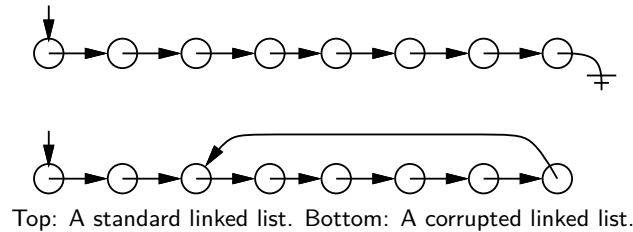
- 28.B.** Describe an algorithm for the case where there are k possible values (i.e., $1, 2, \dots, k$) that rearranges the items using only $O(\log k)$ extra space. How fast is your algorithm? (A faster algorithm would get more credit)

^①Since you've read the Homework Instructions, you know what the phrase 'describe an algorithm' means. Right?

- 28.C. Describe a faster algorithm (if possible) for the case when $O(k)$ extra space is allowed. How fast is your algorithm?
- 28.D. (HARD) Provide a fast algorithm that uses only $O(1)$ additional space for the case where there are k possible values.

29 Snake or shake?

Suppose you have a pointer to the head of singly linked list. Normally, each node in the list only has a pointer to the next element, and the last node's pointer is NULL. Unfortunately, your list might have been corrupted by a bug in somebody else's code², so that the last node has a pointer back to some other node in the list instead.



Describe an algorithm that determines whether the linked list is corrupted or not. Your algorithm must not modify the list. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the list, and use $O(1)$ extra space (not counting the list itself).

38.7. General proof thingies

30 Cornification (20 PTS.)

Cornification - Conversion into, or formation of, horn; a becoming like horn. Source: Webster's Revised Unabridged Dictionary.

During the sweetcorn festival in Urbana, you had been kidnapped by an extreme anti corn organization called Al Corona. To punish you, they give you several sacks with a total of $(n+1)n/2$ cobs of corn in them, and an infinite supply of empty sacks. Next, they ask you to play the following game: At every point in time, you take a cob from every non-empty sack, and you put this set of cobs into a new sack. The game terminates when you have n non-empty sacks, with the i th sack having i cobs in it, for $i = 1, \dots, n$.

For example, if we started with $\{1, 5\}$ (i.e., one sack has 1 cob, the other 5), we would have the following sequence of steps: $\{2, 4\}$, $\{1, 2, 3\}$ and the game ends.

- 30.A. (5 PTS.) Prove that the game terminates if you start from a configuration where all the cobs are in a single sack.
- 30.B. (5 PTS.) Provide a bound, as tight as possible, on the number of steps in the game till it terminates in the case where you start with a single sack.
- 30.C. (5 PTS.) (hard) Prove that the game terminates if you start from an arbitrary configuration where the cobs might be in several sacks.
- 30.D. (5 PTS.) Provide a bound, as tight as possible, on the number of steps in the game till it terminates in the general case.

31 Fibonacci numbers. Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m .

²After all, *your* code is always completely 100% bug-free. Isn't that right, Mr. Gates?

- 31.A. F_n is even if and only if n is divisible by 3.
- 31.B. $\sum_{i=0}^n F_i = F_{n+2} - 1$
- 31.C. $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$ (Really HARD)
- 31.D. If n is an integer multiple of m , then F_n is an integer multiple of F_m .

32 Some binomial identities.

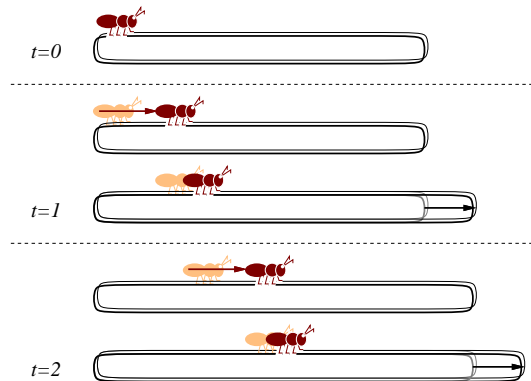
32.A. Prove the following identity by induction:

$$\binom{2n}{n} = \sum_{k=0}^n \binom{n}{k} \binom{n}{n-k}.$$

32.B. Give a non-inductive combinatorial proof of the same identity, by showing that the two sides of the equation count exactly the same thing in two different ways. There is a correct one-sentence proof.

38.8. Miscellaneous

33 **A walking ant.** (HARD) An ant is walking along a rubber band, starting at the left end. Once every second, the ant walks one inch to the right, and then you make the rubber band one inch longer by pulling on the right end. The rubber band stretches uniformly, so stretching the rubber band also pulls the ant to the right. The initial length of the rubber band is n inches, so after t seconds, the rubber band is $n + t$ inches long.



Every second, the ant walks an inch, and then the rubber band is stretched an inch longer.

- 33.A. How far has the ant moved after t seconds, as a function of n and t ? Set up a recurrence and (for full credit) give an *exact* closed-form solution. [Hint: What *fraction* of the rubber band's length has the ant walked?]
- 33.B. How long does it take the ant to get to the right end of the rubber band? For full credit, give an answer of the form $f(n) + \Theta(1)$ for some explicit function $f(n)$.

Chapter 39

Exercises - NP Completeness

39.1. Equivalence of optimization and decision problems

1 Beware of Greeks bearing gifts (The expression “beware of Greeks bearing gifts” is Based on Virgil’s Aeneid: “Quidquid id est, timeo Danaos et dona ferentes”, which means literally “Whatever it is, I fear Greeks even when they bring gifts.”)

The **reduction** faun, the brother of the **Partition** satyr, came to visit you on labor day, and left you with two black boxes.

1.A. (10 PTS.) The first black box, was a black box that can solves the following decision problem in polynomial time:

Minimum Test Collection

Instance: A finite set A of “possible diagnoses,” a collection C of subsets of A , representing binary “tests,” and a positive integer $J \leq |C|$.

Question: Is there a subcollection $C' \subseteq C$ with $|C'| \leq J$ such that, for every pair a_i, a_j of possible diagnoses from A , there is some test $c \in C'$ for which $|\{a_i, a_j\} \cap c| = 1$ (that is, a test c that “distinguishes” between a_i and a_j)?

Show how to use this black box, how to solve in polynomial time the optimization version of this problem (i.e., finding and outputting the smallest possible set C').

1.B. (10 PTS.)

The second box was a black box for solving

Subgraph Isomorphism.

Subgraph Isomorphism

Instance: Two graphs, $G = (V_1, E_1)$ and $H = (V_2, E_2)$.

Question: Does G contain a subgraph *isomorphic* to H , that is, a subset $V \subseteq V_1$ and a subset $E \subseteq E_1$ such that $|V| = |V_2|$, $|E| = |E_2|$, and there exists a one-to-one function $f : V_2 \rightarrow V$ satisfying $\{u, v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E$?

Show how to use this black box, to compute the subgraph isomorphism (i.e., you are given G and H , and you have to output f) in polynomial time.

2 Partition The **Partition** satyr, the uncle of the deduction fairy, had visited you on winter break and gave you, as a token of appreciation, a black-box that can solve **Partition** in polynomial time (note that this black box solves the decision problem). Let S be a given set of n integer numbers. Describe a polynomial time algorithm that computes, using the black box, a partition of S if such a solution exists. Namely, your algorithm should output a subset $T \subseteq S$, such that

$$\sum_{s \in T} s = \sum_{s \in S \setminus T} s.$$

39.2. Showing problems are NP-Complete

3 Graph Isomorphism

3.A. (5 PTS.) Show that the following problem is NP-COMPLETE.

SUBGRAPH ISOMORPHISM

Instance: Graphs $G = (V_1, E_1), H = (V_2, E_2)$.

Question: Does G contain a subgraph isomorphic to H , i.e., a subset $V \subseteq V_1$ and a subset $E \subseteq E_1$ such that $|V| = |V_2|$, $|E| = |E_2|$, and there exists a one-to-one function $f : V_2 \rightarrow V$ satisfying $\{u, v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E$?

3.B. (5 PTS.) Show that the following problem is NP-COMPLETE.

LARGEST COMMON SUBGRAPH

Instance: Graphs $G = (V_1, E_1), H = (V_2, E_2)$, positive integer K .

Question: Do there exist subsets $E'_1 \subseteq E_1$ and $E'_2 \subseteq E_2$ with $|E'_1| = |E'_2| \geq K$ such that the two subgraphs $G' = (V_1, E'_1)$ and $H' = (V_2, E'_2)$ are isomorphic?

4 NP Completeness collection

4.A. (5 PTS.)

MINIMUM SET COVER

Instance: Collection C of subsets of a finite set S and an integer k .

Question: Are there k sets S_1, \dots, S_k in C such that $S \subseteq \cup_{i=1}^k S_i$?

4.B. (5 PTS.)

BIN PACKING

Instance: Finite set U of items, a size $s(u) \in \mathbb{Z}^+$ for each $u \in U$, an integer bin capacity B , and a positive integer K .

Question: Is there a partition of U into disjoint sets U_1, \dots, U_K such that the sum of the sizes of the items inside each U_i is B or less?

4.C. (5 PTS.)

TILING

Instance: Finite set \mathcal{RECTS} of rectangles and a rectangle R in the plane.

Question: Is there a way of placing the rectangles of \mathcal{RECTS} inside R , so that no pair of the rectangles intersect, and all the rectangles have their edges parallel to the edges of R ?

4.D. (5 PTS.)

HITTING SET

Instance: A collection C of subsets of a set S , a positive integer K .

Question: Does S contain a *hitting set* for C of size K or less, that is, a subset $S' \subseteq S$ with $|S'| \leq K$ and such that S' contains at least one element from each subset in C .

- 5 LONGEST-PATH** Show that the problem of deciding whether an unweighted undirected graph has a path of length greater than k is **NP-COMplete**.
- 6 EXACT-COVER-BY-4-SETS** The **EXACT-COVER-BY-3-SETS** problem is defined as the following: given a finite set X with $|X| = 3q$ and a collection C of 3-element subsets of X , does C contain an *exact cover* for X , that is, a subcollection $C' \subseteq C$ such that every element of X occurs in exactly one member of C' ?

Given that EXACT-COVER-BY-3-SETS is **NP-COMplete**, show that EXACT-COVER-BY-4-SETS is also **NP-COMplete**.

39.3. Solving special subcases of **NP-Complete** problems in polynomial time

7 Subset Sum

Subset Sum

Instance: S - set of positive integers, t : - an integer number

Question: Is there a subset $X \subseteq S$ such that

$$\sum_{x \in X} x = t ?$$

Given an instance of **Subset Sum**, provide an algorithm that solves it in polynomial time in n , and M , where $M = \max_{s \in S} s$. Why this does not imply that $P = NP$?

- 8 2SAT** Given an instance of **2SAT** (this is a problem similar to **3SAT** where every clause has at most two variables), one can try to solve it by backtracking.

8.A. (1 PTS.) Prove that if a formula F' is not satisfiable, and F is formed by adding clauses to F' , then the formula F is not satisfiable. (Duh?)

We refer to F' as a *subformula* of F .

8.B. (3 PTS.) Given an assignment $x_i \leftarrow b$ to one of the variables of a **2SAT** instance F (where b is either 0 or 1), describe a polynomial time algorithm that computes a subformula F' of F , such that (i) F' does not have the variable x_i in it, (ii) F' is a **2SAT** formula, (iii) F' is satisfiable iff there is a satisfying assignment for F with $x_i = b$, and (iv) F' is a subformula of F .

How fast is your algorithm?

8.C. (6 PTS.) Describe a polynomial time algorithm that solves the **2SAT** problem (using (b)). How fast is your algorithm?

- 9 2-CNF-SAT** Prove that deciding satisfiability when all clauses have at most 2 literals is in **P**.

- 10 Hamiltonian Cycle Revisited** Let C_n denote the cycle graph over n vertices (i.e., $V(C_n) = \{1, \dots, n\}$, and $E(C_n) = \{\{1, 2\}, \{2, 3\}, \dots, \{n-1, n\}, \{n, 1\}\}$). Let C_n^k denote the graph where $\{i, j\} \in E(C_n^k)$ iff i and j are in distance at most k in C_n .

Let G be a graph, such that G is a subgraph of C_n^k , where k is a small constant. Describe a polynomial time algorithm (in n) that outputs a Hamiltonian cycle if such a cycle exists in G . How fast is your algorithm, as a function of n and k ?

11 Partition revisited Let S be an instance of partition, such that $n = |S|$, and $M = \max_{s \in S} s$. Show a polynomial time (in n and M) algorithm that solves partition.

12 Why Mike can not get it. (10 PTS.)

Not-3SAT

Instance: A 3CNF formula F

Question: Is F not satisfiable? (Namely, for all inputs for F , it evaluates to FALSE.)

12.A. Prove that **Not-3SAT** is *co-NP*.

12.B. Here is a proof that **Not-3SAT** is in *NP*: If the answer to the given instance is **Yes**, we provide the following proof to the verifier: We list every possible assignment, and for each assignment, we list the output (which is FALSE). Given this proof, of length L , the verifier can easily verify it in polynomial time in L . QED.

What is wrong with this proof?

12.C. Show that given a black-box that can solves **Not-3SAT**, one can find the satisfying assignment of a formula F in polynomial time, using polynomial number of calls to the black-box (if such an assignment exists).

13 NP-Completeness Collection (20 PTS.) Prove that the following problems are *NP-Complete*.

MINIMUM SET COVER

13.A.

Instance: Collection C of subsets of a finite set S and an integer k .

Question: Are there k sets S_1, \dots, S_k in C such that $S \subseteq \cup_{i=1}^k S_i$?

HITTING SET

13.B.

Instance: A collection C of subsets of a set S , a positive integer K .

Question: Does S contain a *hitting set* for C of size K or less, that is, a subset $S' \subseteq S$ with $|S'| \leq K$ and such that S' contains at least one element from each subset in C .

Hamiltonian Path

13.C.

Instance: Graph $G = (V, E)$

Question: Does G contains a Hamiltonian path? (Namely a path that visits all vertices of G .)

Max Degree Spanning Tree

13.D.

Instance: Graph $G = (V, E)$ and integer k

Question: Does G contains a spanning tree T where every node in T is of degree at most k ?

14 Independence (10 PTS.) Let $G = (V, E)$ be an undirected graph over n vertices. Assume that you are given a numbering $\pi : V \rightarrow \{1, \dots, n\}$ (i.e., every vertex have a unique number), such that for any edge $ij \in E$, we have $|\pi(i) - \pi(j)| \leq 20$.

Either prove that it is *NP-Hard* to find the largest independent set in G , or provide a polynomial time algorithm.

15 Partition We already know the following problem is **NP-COMplete**:

SUBSET SUM

Instance: A finite set A and a “size” $s(a) \in \mathbb{Z}^+$ for each $a \in A$, an integer B .

Question: Is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = B$?

Now let's consider the following problem:

PARTITION

Instance: A finite set A and a “size” $s(a) \in \mathbb{Z}^+$ for each $a \in A$.

Question: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)?$$

Show that PARTITION is NP-COMplete.

16 Minimum Set Cover (15 PTS.)

MINIMUM SET COVER

Instance: Collection C of subsets of a finite set S and an integer k .

Question: Are there k sets S_1, \dots, S_k in C such that $S \subseteq \cup_{i=1}^k S_i$?

16.A. (5 PTS.) Prove that MINIMUM SET COVER problem is NP-COMplete

16.B. (5 PTS.) Prove that the following problem is NP-COMplete.

HITTING SET

Instance: A collection C of subsets of a set S , a positive integer K .

Question: Does S contain a *hitting set* for C of size K or less, that is, a subset $S' \subseteq S$ with $|S'| \leq K$ and such that S' contains at least one element from each subset in C .

16.C. (5 PTS.) *Hitting set on the line*

Given a set \mathcal{I} of n intervals on the real line, show a $O(n \log n)$ time algorithm that computes the smallest set of points X on the real line, such that for every interval $I \in \mathcal{I}$ there is a point $p \in X$, such that $p \in I$.

17 Bin Packing

BIN PACKING

Instance: Finite set U of items, a size $s(u) \in \mathbb{Z}^+$ for each $u \in U$, an integer bin capacity B , and a positive integer K .

Question: Is there a partition of U into disjoint sets U_1, \dots, U_K such that the sum of the sizes of the items inside each U_i is B or less?

17.A. (5 PTS.) Show that the BIN PACKING problem is NP-COMplete

17.B. (5 PTS.) Show that the following problem is NP-COMplete.

TILING

Instance: Finite set \mathcal{RECTS} of rectangles and a rectangle R in the plane.

Question: Is there a way of placing all the rectangles of \mathcal{RECTS} inside R , so that no pair of the rectangles intersect in their interior, and all the rectangles have their edges parallel of the edges of R ?

18 Knapsack

18.A. (5 PTS.) Show that the following problem is NP-COMplete.

KNAPSACK

Instance: A finite set U , a "size" $s(u) \in \mathbb{Z}^+$ and a "value" $v(u) \in \mathbb{Z}^+$ for each $u \in U$, a size constraint $B \in \mathbb{Z}^+$, and a value goal $K \in \mathbb{Z}^+$.

Question: Is there a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq K$.

18.B. (5 PTS.) Show that the following problem is NP-COMplete.

MULTIPROCESSOR SCHEDULING

Instance: A finite set A of "tasks", a "length" $l(a) \in \mathbb{Z}^+$ for each $a \in A$, a number $m \in \mathbb{Z}^+$ of "processors", and a "deadline" $D \in \mathbb{Z}^+$.

Question: Is there a partition $A = A_1 \cup A_2 \cup \dots \cup A_m$ of A into m disjoint sets such that $\max\{\sum_{a \in A_i} l(a) : 1 \leq i \leq m\} \leq D$?

18.C. Scheduling with profits and deadlines

Suppose you have one machine and a set of n tasks a_1, a_2, \dots, a_n . Each task a_j has a processing time t_j , a profit p_j , and a deadline d_j . The machine can process only one task at a time, and task a_j must run uninterruptedly for t_j consecutive time units to complete. If you complete task a_j by its deadline d_j , you receive a profit p_j . But you receive no profit if you complete it after its deadline. As an optimization problem, you are given the processing times, profits and deadlines for a set of n tasks, and you wish to find a schedule that completes all the tasks and returns the greatest amount of profit.

18.C.i. (3 PTS.) State this problem as a decision problem.

18.C.ii. (2 PTS.) Show that the decision problem is NP-COMplete.

19 Vertex Cover

VERTEX COVER

Instance: A graph $G = (V, E)$ and a positive integer $K \leq |V|$.

Question: Is there a *vertex cover* of size K or less for G , that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and for each edge $\{u, v\} \in E$, at least one of u and v belongs to V' ?

19.A. Show that VERTEX COVER is NP-COMplete. Hint: Do a reduction from INDEPENDENT SET to VERTEX COVER.

19.B. Show a polynomial approximation algorithm to the VERTEX-COVER problem which is a factor 2 approximation of the optimal solution. Namely, your algorithm should output a set $X \subseteq V$, such that X is a vertex cover, and $|X| \leq 2K_{opt}$, where K_{opt} is the cardinality of the smallest vertex cover of G .^①

^①It was very recently shown (I. Dinur and S. Safra. On the importance of being biased. Manuscript. <http://www.math.ias.edu/~iritd/mypapers/vc.pdf>, 2001.) that doing better than 1.3600 approximation to VERTEX COVER is NP-Hard. In your free time you can try and improve this constant. Good luck.

- 19.C. Present a linear time algorithm that solves this problem for the case that G is a tree.
- 19.D. For a constant k , a graph G is k -separable, if there are k vertices of G , such that if we remove them from G , each one of the remaining connected components has at most $(2/3)n$ vertices, and furthermore each one of those connected components is also k -separable. (More formally, a graph $G = (V, E)$ is k -separable, if for any subset of vertices $S \subseteq V$, there exists a subset $M \subseteq S$, such that each connected component of $G_{S \setminus M}$ has at most $(2/3)|S|$ vertices, and $|M| \leq k$.)
Show that given a graph G which is k -separable, one can compute the optimal VERTEX COVER in $n^{O(k)}$ time.

20 Bin Packing

BIN PACKING

Instance: Finite set U of items, a size $s(u) \in \mathbb{Z}^+$ for each $u \in U$, an integer bin capacity B , and a positive integer K .

Question: Is there a partition of U into disjoint sets U_1, \dots, U_K such that the sum of the sizes of the items inside each U_i is B or less?

- 20.A. Show that the BIN PACKING problem is NP-COMplete.
- 20.B. In the optimization variant of BIN PACKING one has to find the minimum number of bins needed to contain all elements of U . Present an algorithm that is a factor two approximation to optimal solution. Namely, it outputs a partition of U into M bins, such that the total size of each bin is at most B , and $M \leq k_{opt}$, where k_{opt} is the minimum number of bins of size B needed to store all the given elements of U .
- 20.C. Assume that B is bounded by an integer constant m . Describe a polynomial algorithm that computes the solution that uses the minimum number of bins to store all the elements.
- 20.D. Show that the following problem is NP-COMplete.

TILING

Instance: Finite set \mathcal{RECTS} of rectangles and a rectangle R in the plane.

Question: Is there a way of placing the rectangles of \mathcal{RECTS} inside R , so that no pair of the rectangles intersect, and all the rectangles have their edges parallel of the edges of R ?

- 20.E. Assume that \mathcal{RECTS} is a set of squares that can be arranged as to tile R completely. Present a polynomial time algorithm that computes a subset $\mathcal{T} \subseteq \mathcal{RECTS}$, and a tiling of \mathcal{T} , so that this tiling of \mathcal{T} covers, say, 10% of the area of R .

21 Minimum Set Cover

MINIMUM SET COVER

Instance: Collection C of subsets of a finite set S and an integer k .

Question: Are there k sets S_1, \dots, S_k in C such that $S \subseteq \cup_{i=1}^k S_i$?

- 21.A. Prove that MINIMUM SET COVER problem is NP-COMplete.
- 21.B. The greedy approximation algorithm for MINIMUM SET COVER, works by taking the largest set in $X \in C$, remove all all the elements of X from S and also from each subset of C . The algorithm repeat this until all the elements of S are removed. Prove that the number of elements not covered after k_{opt} iterations is at most $n/2$, where k_{opt} is the smallest number of sets of C needed to cover S , and $n = |S|$.

- 21.C. Prove the greedy algorithm is $O(\log n)$ factor optimal approximation.
- 21.D. Prove that the following problem is NP-COMPLETE.

HITTING SET

Instance: A collection C of subsets of a set S , a positive integer K .

Question: Does S contain a *hitting set* for C of size K or less, that is, a subset $S' \subseteq S$ with $|S'| \leq K$ and such that S' contains at least one element from each subset in C .

- 21.E. Given a set \mathcal{I} of n intervals on the real line, show a $O(n \log n)$ time algorithm that computes the smallest set of points X on the real line, such that for every interval $I \in \mathcal{I}$ there is a point $p \in X$, such that $p \in I$.

22 k -Center

k -CENTER

Instance: A set P of n points in the plane, and an integer k and a radius r .

Question: Is there a cover of the points of P by k disks of radius (at most) r ?

- 22.A. Describe an $n^{O(k)}$ time algorithm that solves this problem.
- 22.B. There is a very simple and natural algorithm that achieves a 2-approximation for this cover: First it select an arbitrary point as a center (this point is going to be the center of one of the k covering disks). Then it computes the point that it furthest away from the current set of centers as the next center, and it continue in this fashion till it has k -points, which are the resulting centers. The smallest k equal radius disks centered at those points are the required k disks.
Show an implementation of this approximation algorithm in $O(nk)$ time.
- 22.C. Prove that that the above algorithm is a factor two approximation to the optimal cover. Namely, the radius of the disks output $\leq 2r_{opt}$, where r_{opt} is the smallest radius, so that we can find k -disks that cover the point-set.
- 22.D. Provide an ε -approximation algorithm for this problem. Namely, given k and a set of points P in the plane, your algorithm would output k -disks that cover the points and their radius is $\leq (1 + \varepsilon)r_{opt}$, where r_{opt} is the minimum radius of such a cover of P .
- 22.E. Prove that dual problem r -DISK-COVER problem is NP-Hard. In this problem, given P and a radius r , one should find the smallest number of disks of radius r that cover P .
- 22.F. Describe an approximation algorithm to the r -DISK COVER problem. Namely, given a point-set P and a radius r , outputs k disks, so that the k disks cover P and are of radius r , and $k = O(k_{opt})$, where k_{opt} is the minimal number of disks needed to cover P by disks of radius r .

23 MAX 3SAT Consider the Problem MAX SAT.

MAX SAT

Instance: Set U of variables, a collection C of disjunctive clauses of literals where a literal is a variable or a negated variable in U .

Question: Find an assignment that maximized the number of clauses of C that are being satisfied.

- 23.A. Prove that MAX SAT is NP-Hard.
- 23.B. Prove that if each clause has exactly three literals, and we randomly assign to the variables values 0 or 1, then the expected number of satisfied clauses is $(7/8)M$, where $M = |C|$.
- 23.C. Show that for any instance of MAX SAT, where each clause has exactly three different literals, there exists an assignment that satisfies at least $7/8$ of the clauses.
- 23.D. Let (U, C) be an instance of MAX SAT such that each clause has $\geq 10 \cdot \log n$ distinct variables, where n is the number of clauses. Prove that there exists a satisfying assignment. Namely, there exists an assignment that satisfy all the clauses of C .

24 Complexity

- 24.A. Prove that $P \subseteq \text{co-NP}$.
- 24.B. Show that if $\text{NP} \neq \text{co-NP}$, then every NP-COMplete problem is *not* a member of CO-NP.

25 **3SUM** Describe an algorithm that solves the following problem as quickly as possible: Given a set of n numbers, does it contain three elements whose sum is zero? For example, your algorithm should answer TRUE for the set $\{-5, -17, 7, -4, 3, -2, 4\}$, since $-5+7+(-2) = 0$, and FALSE for the set $\{-6, 7, -4, -13, -2, 5, 13\}$.

26 **Polynomially equivalent.** Consider the following pairs of problems:

- (A) MIN SPANNING TREE and MAX SPANNING TREE.
- (B) SHORTEST PATH and LONGEST PATH.
- (C) TRAVELING SALESMAN PROBLEM and VACATION TOUR PROBLEM (the longest tour is sought).
- (D) MIN CUT and MAX CUT (between s and t).
- (E) EDGE COVER and VERTEX COVER.
- (F) TRANSITIVE REDUCTION and MIN EQUIVALENT DIGRAPH.

(all of these seem dual or opposites, except the last, which are just two versions of minimal representation of a graph).

Which of these pairs are polynomial time equivalent and which are not? Why?

27 **PLANAR-3-COLOR** Using 3COLORABLE, and the ‘gadget’ in figure below, prove that the problem of deciding whether a planar graph can be 3-colored is NP-COMplete. Hint: show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.

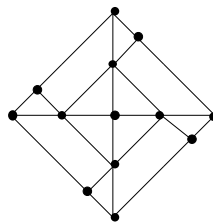


Figure 39.1: Gadget for PLANAR-3-COLOR.

28 **DEGREE-4-PLANAR-3-COLOR** Using the previous result, and the ‘gadget’ in the figure below, prove that the problem of deciding whether a planar graph with no vertex of degree greater than four can be 3-colored is NP-COMplete. Hint: show that you can replace any vertex with degree greater than 4 with a collection of gadgets connected in such a way that no degree is greater than four.

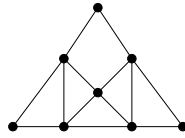


Figure 39.2: Gadget for DEGREE-4-PLANAR-3-COLOR.

29 Primality and Complexity Prove that **PRIMALITY** (Given n , is n prime?) is in $\text{NP} \cap \text{co-NP}$. Hint: **co-NP** is easy (what's a certificate for showing that a number is composite?). For **NP**, consider a certificate involving primitive roots and recursively their primitive roots. Show that knowing this tree of primitive roots can be checked to be correct and used to show that n is prime, and that this check takes poly time.

30 Poly time subroutines can lead to exponential algorithms Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

31 Polynomial time Hamiltonian path

- 31.A.** Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is non-hamiltonian. Give a polynomial time algorithm for finding a **hamiltonian cycle** in an undirected bipartite graph or establishing that it does not exist.
- 31.B.** Show that the **hamiltonian-path** problem can be solved in polynomial time on directed acyclic graphs by giving an efficient algorithm for the problem.
- 31.C.** Explain why the results in previous questions do not contradict the facts that both **HAM-CYCLE** and **HAM-PATH** are **NP-COMplete** problems.

32 (Really HARD)GRAPH-ISOMORPHISM Consider the problem of deciding whether one graph is isomorphic to another.

- 32.A.** Give a brute force algorithm to decide this.
- 32.B.** Give a dynamic programming algorithm to decide this.
- 32.C.** Give an efficient probabilistic algorithm to decide this.
- 32.D.** Either prove that this problem is **NP-COMplete**, give a poly time algorithm for it, or prove that neither case occurs.

33 (t, k) -grids. (20 PTS.)

A graph G is a (t, k) -grid if its vertices are

$$V(G) = \{(i, j) \mid i = 1, \dots, n/k, j = 1, \dots, k\},$$

and two vertices (x_1, x_2) and (y_1, y_2) can be connected only if $|x_1 - y_1| + |x_2 - y_2| \leq t$. Here n is the number of vertices of G .

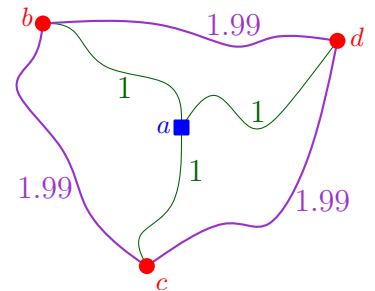
- 33.A.** (8 PTS.) Present an efficient algorithm that computes a **Vertex Cover** of minimum size in a given $(t, 1)$ -grid G (here you can assume that t is a constant).
- 33.B.** (12 PTS.) Let t and k be two constants. Provide an algorithm (as fast as possible) that in polynomial time computes the *maximum* size **Independent Set** for G . What is the running time of your algorithm (explicitly specify the dependency on t and k)?

34 Build the network. (20 PTS.)

You had decided to immigrate to Norstrilia (never heard of it? Google for it), and you had discovered to your horror that because of import laws the cities of Norstrilia are not even connected by a fast computer network. You join the Roderick company which decided to connect the k major cities by a network. To be as cheap as possible, your network is just going to be a spanning tree of these k cities, but you are allowed to put additional vertices in your network in some other cities. For every pair of cities, you know what is the price of laying a line connecting them. Your task is to compute the cheapest spanning tree for those k cities.

Formally, you are given a complete graph $G = (V, E)$ defined over n vertices. There is a (positive) weight $w(e)$ associated with each edges $e \in E(G)$. Furthermore, you can assume that $\forall i, j, k \in V$ you have $w(ik) \leq w(ij) + w(jk)$ (i.e., the triangle inequality). Finally, you are given a set $X \subseteq V$ of k vertices of G . You need to compute the cheapest tree T , such that $X \subseteq V(T)$, where the price of the tree T is $w(T) = \sum_{e \in E(T)} w(e)$.

To see why this problem is interesting, and inherently different from the minimum spanning tree problem, consider the graph on the right. The optimal solution, if we have to connect the three round vertices (i.e., b, c, d), is by taking the three middle edges ab, ad, ac (total price is 3). The naive solution, would be to take bc and cd , but its cost is 3.98. Note that the triangle inequality holds for the weights in this graph.



- 34.A. (5 PTS.) Provide a $n^{O(k)}$ time algorithm for this problem.
- 34.B. (15 PTS.) Provide an algorithm for this problem with running time $O(f(k) \cdot n^c)$, where $f(k)$ is a function of k , and c is a constant independent of the value of k .

(Comments: This problem is **NP-HARD**, although a 2-approximation is relatively easy. Problems that have running time like in (B) are referred to as **fixed parameter tractable**, since their running time is polynomial for a fixed value of the parameters.)

Chapter 40

Exercises - Network Flow

This chapter include problems that are related to network flow.

40.1. Network Flow

40.1.1. The good, the bad, and the middle.

(10 PTS.)

Suppose you're looking at a flow network G with source s and sink t , and you want to be able to express something like the following intuitive notion: Some nodes are clearly on the “source side” of the main bottlenecks; some nodes are clearly on the “sink side” of the main bottlenecks; and some nodes are in the middle. However, G can have many minimum cuts, so we have to be careful in how we try making this idea precise.

Here's one way to divide the nodes of G into three categories of this sort.

- We say a node v is *upstream* if, for all minimum s - t cuts (A, B) , we have $v \in A$ – that is, v lies on the source side of every minimum cut.
- We say a node v is *downstream* if, for all minimum s - t cuts (A, B) , we have $v \in B$ – that is, v lies on the sink side of every minimum cut.
- We say a node v is *central* if it is neither upstream nor downstream; there is at least one minimum s - t cut (A, B) for which $v \in A$, and at least one minimum s - t cut (A', B') for which $v \in B'$.

Give an algorithm that takes a flow network G and classifies each of its nodes as being upstream, downstream, or central. The running time of your algorithm should be within a constant factor of the time required to compute a *single* maximum flow.

40.1.2. Ad hoc networks

(20 PTS.)

Ad hoc networks are made up of low-powered wireless devices, have been proposed for situations like natural disasters in which the coordinators of a rescue effort might want to monitor conditions in a hard-to-reach area. The idea is that a large collection of these wireless devices could be dropped into such an area from an airplane and then configured into a functioning network.

Note that we're talking about (a) relatively inexpensive devices that are (b) being dropped from an airplane into (c) dangerous territory; and for the combination of reasons (a), (b), and (c), it becomes necessary to include provisions for dealing with the failure of a reasonable number of the nodes.

We'd like it to be the case that if one of the devices v detects that it is in danger of failing, it should transmit a representation of its current state to some other device in the network. Each device has a limited transmitting range – say it can communicate with other devices that lie within d meters of it. Moreover, since we don't want it to try transmitting its state to a device that has already failed, we should include some redundancy: A device v should have a set of k other devices that it can potentially contact, each within d meters of it. We'll call this a *back-up set* for device v .

1. Suppose you're given a set of n wireless devices, with positions represented by an (x, y) coordinate pair for each. Design an algorithm that determines whether it is possible to choose a back-up set for each device (i.e., k other devices, each within d meters), with the further property that, for some parameter b , no device appears in the back-up set of more than b other devices. The algorithm should output the back-up sets themselves, provided they can be found.
2. The idea that, for each pair of devices v and w , there's a strict dichotomy between being “in range” or “out of range” is a simplified abstraction. More accurately, there's a power decay function $f(\cdot)$ that specifies, for a pair of devices at distance δ , the signal strength $f(\delta)$ that they'll be able to achieve on their wireless connection. (We'll assume that $f(\delta)$ decreases with increasing δ .)

We might want to build this into our notion of back-up sets as follows: among the k devices in the back-up set of v , there should be at least one that can be reached with very high signal strength, at least one other that can be reached with moderately high signal strength, and so forth. More concretely, we have values $p_1 \geq p_2 \geq \dots \geq p_k$, so that if the back-up set for v consists of devices at distances $d_1 \leq d_2 \leq \dots \leq d_k$, then we should have $f(d_j) \geq p_j$ for each j .

Give an algorithm that determines whether it is possible to choose a back-up set for each device subject to this more detailed condition, still requiring that no device should appear in the back-up set of more than b other devices. Again, the algorithm should output the back-up sets themselves, provided they can be found.

40.1.3. Minimum Flow

(10 PTS.)

Give a polynomial-time algorithm for the following minimization analogue of the Maximum-Flow Problem. You are given a directed graph $G = (V, E)$, with a source $s \in V$ and sink $t \in V$, and numbers (capacities) $\ell(v, w)$ for each edge $(v, w) \in E$. We define a flow f , and the value of a flow, as usual, requiring that all nodes except s and t satisfy flow conservation. However, the given numbers are lower bounds on edge flow – that is, they require that $f(v, w) \geq \ell(v, w)$ for every edge $(v, w) \in E$, and there is no upper bound on flow values on edges.

1. Give a polynomial-time algorithm that finds a feasible flow of minimum possible values.
2. Prove an analogue of the Max-Flow Min-Cut Theorem for this problem (i.e., does min-flow = max-cut?).

40.1.4. Prove infeasibility.

You are trying to solve a circulation problem, but it is not feasible. The problem has demands, but no capacity limits on the edges. More formally, there is a graph $G = (V, E)$, and demands d_v for each node v (satisfying $\sum_{v \in V} d_v = 0$), and the problem is to decide if there is a flow f such that $f(e) \geq 0$ and $f^{in}(v) - f^{out}(v) = d_v$ for all nodes $v \in V$. Note that this problem can be solved via the circulation algorithm from Section 7.7 by setting $c_e = +\infty$ for all edges $e \in E$. (Alternately, it is enough to set c_e to be an extremely large number for each edge – say, larger than the total of all positive demands d_v in the graph.)

You want to fix up the graph to make the problem feasible, so it would be very useful to know why the problem is not feasible as it stands now. On a closer look, you see that there is a subset U of nodes such that there is no edge into U , and yet $\sum_{v \in U} d_v > 0$. You quickly realize that the existence of such a set immediately implies that the flow cannot exist: The set U has a positive total demand, and so needs incoming flow, and yet U has no edges into it. In trying to evaluate how far the problem is from being solvable, you wonder how big the demand of a set with no incoming edges can be.

Give a polynomial-time algorithm to find a subset $S \subset V$ of nodes such that there is no edge into S and for which $\sum_{v \in S} d_v$ is as large as possible subject to this condition.

40.1.5. Cellphones and services.

Consider an assignment problem where we have a set of n stations that can provide service, and there is a set of k requests for service. Say, for example, that the stations are cell towers and the requests are cell phones. Each request can be served by a given set of stations. The problem so far can be represented by a bipartite graph G : one side is the stations, the other the customers, and there is an edge (x, y) between customer x and station y if customer x can be served from station y . Assume that each station can serve at most one customer. Using a max-flow computation, we can decide whether or not all customers can be served, or can get an assignment of a subset of customers to stations maximizing the number of served customers.

Here we consider a version of the problem with an addition complication: Each customer offers a different amount of money for the service. Let U be the set of customers, and assume that customer $x \in U$ is willing to pay $v_x \geq 0$ for being served. Now the goal is to find a subset $X \subset U$ maximizing $\sum_{x \in X} v_x$ such that there is an assignment of the customers in X to stations.

Consider the following greedy approach. We process customers in order of decreasing value (breaking ties arbitrarily). When considering customer x the algorithm will either “promise” service to x or reject x in the

following greedy fashion. Let X be the set of customers that so far have been promised service. We add x to the set X if and only if there is a way to assign $X \cup \{x\}$ to servers, and we reject x otherwise. Note that rejected customers will not be considered later. (This is viewed as an advantage: If we need to reject a high-paying customer, at least we can tell him/her early.) However, we do not assign accepting customers to servers in a greedy fashion: we only fix the assignment after the set of accepted customers is fixed. Does this greedy approach produce an optimal set of customers? Prove that it does, or provide a counterexample.

40.1.6. Follow the stars

(20 PTS.)

Some friends of yours have grown tired of the game “Six Degrees of Kevin Bacon” (after all, they ask, isn’t it just breadth-first search?) and decide to invent a game with a little more punch, algorithmically speaking. Here’s how it works.

You start with a set X of n actresses and a set Y of n actors, and two players P_0 and P_1 . Player P_0 names an actress $x_1 \in X$, player P_1 names an actor y_1 who has appeared in a movie with x_1 , player P_0 names an actress x_2 who has appeared in a movie with y_1 , and so on. Thus, P_0 and P_1 collectively generate a sequence $x_1, y_1, x_2, y_2, \dots$ such that each actor/actress in the sequence has costarred with the actress/actor immediately preceding. A player P_i ($i = 0, 1$) loses when it is P_i ’s turn to move, and he/she cannot name a member of his/her set who hasn’t been named before.

Suppose you are given a specific pair of such sets X and Y , with complete information on who has appeared in a movie with whom. A *strategy* for P_i , in our setting, is an algorithm that takes a current sequence $x_1, y_1, x_2, y_2, \dots$ and generates a legal next move for P_i (assuming it’s P_i ’s turn to move). Give a polynomial-time algorithm that decides which of the two players can force a win, in a particular instance of this game.

40.1.7. Flooding

(10 PTS.)

Network flow issues come up in dealing with natural disasters and other crises, since major unexpected events often require the movement and evacuation of large numbers of people in a short amount of time.

Consider the following scenario. Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour’s driving time of their current location (so different people will have different options for hospitals, depending on where they are right now).

At the same time, one doesn’t want to overload any one of the hospitals by sending too many patients its way. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is *balanced*: Each hospital receives at most $\lceil n/k \rceil$ people.

Give a polynomial-time algorithm that takes the given information about the people’s locations and determines whether this is possible.

40.1.8. Capacitation, yeh, yeh, yeh

Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e , a designated source $s \in V$, and a designated sink $t \in V$. You are also given a maximum s - t flow in G , defined by a flow value f_e on each edge e . The flow $\{f_e\}$ is *acyclic*: There is no cycle in G on which all edges carry positive flow.

Now suppose we pick a specific edge $e^* \in E$ and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m + n)$, where m is the number of edges in G and n is the number of nodes.

40.1.9. Fast Friends

(20 PTS.)

Your friends have written a very fast piece of maximum-flow code based on repeatedly finding augmenting paths as in the course lecture notes. However, after you've looked at a bit of output from it, you realize that it's not always finding a flow of *maximum* value. The bug turns out to be pretty easy to find; your friends hadn't really gotten into the whole backward-edge thing when writing the code, and so their implementation builds a variant of the residual graph that *only includes the forwards edges*. In other words, it searches for s - t paths in a graph \tilde{G}_f consisting only of edges of e for which $f(e) < c_e$, and it terminates when there is no augmenting path consisting entirely of such edges. We'll call this the Forward-Edge-Only Algorithm. (Note that we do not try to prescribe how this algorithm chooses its forward-edge paths; it may choose them in any fashion it wants, provided that it terminates only when there are no forward-edge paths.)

It's hard to convince your friends they need to reimplement the code. In addition to its blazing speed, they claim, in fact, that it never returns a flow whose value is less than a fixed fraction of optimal. Do you believe this? The crux of their claim can be made precise in the following statement.

"There is an absolute constant $b > 1$ (independent of the particular input flow network), so that on every instance of the Maximum-Flow Problem, the Forward-Edge-Only Algorithm is guaranteed to find a flow of value at least $1/b$ times the maximum-flow value (regardless of how it chooses its forward-edge paths)."

Decide whether you think this statement is true or false, and give a proof of either the statement or its negation.

40.1.10. Even More Capacitation

(10 PTS.)

In a standard s - t Maximum-Flow Problem, we assume edges have capacities, and there is no limit on how much flow is allowed to pass through a node. In this problem, we consider the variant of the Maximum-Flow and Minimum-Cut problems with node capacities.

Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and nonnegative node capacities $\{c_v \geq 0\}$ for each $v \in V$. Given a flow f in this graph, the flow through a node v is defined as $f^{in}(v)$. We say that a flow is feasible if it satisfies the usual flow-conservation constraints and the node-capacity constraints: $f^{in}(v) \leq c_v$ for all nodes.

Give a polynomial-time algorithm to find an s - t maximum flow in such a node-capacitated network. Define an s - t cut for node-capacitated networks, and show that the analogue of the Max-Flow Min-Cut Theorem holds true.

40.1.11. Matrices

(10 PTS.)

Let M be an $n \times n$ matrix with each entry equal to either 0 or 1. Let m_{ij} denote the entry in row i and column j . A *diagonal entry* is one of the form m_{ii} for some i .

Swapping rows i and j of the matrix M denotes the following action: we swap the values of m_{ik} and m_{jk} , for $k = 1, \dots, n$. Swapping two columns is defined analogously.

We say that M is **rearrangeable** if it is possible to swap some of the pairs of rows and some of the pairs of columns (in any sequence) so that after all the swapping, all the diagonal entries of M are equal to 1.

- (2 PTS.) Give an example of a matrix M that is not rearrangeable, but for which at least one entry in each row and each column is equal to 1.
- (8 PTS.) Give a polynomial-time algorithm that determines whether a matrix M with 0-1 entries is rearrangeable.

40.1.12. Unique Cut

(10 PTS.)

Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and nonnegative edge capacities $\{c_e\}$. Give a polynomial-time algorithm to decide whether G has a *unique* minimum s - t cut (i.e., an s - t of capacity strictly less than that of all other s - t cuts).

40.1.13. Transitivity

(10 PTS.)

Given a graph $G = (V, E)$, and a natural number k , we can define a relation $\xrightarrow{G,k}$ on pairs of vertices of G as follows. If $x, y \in V$, we say that $x \xrightarrow{G,k} y$ if there exist k mutually edge-disjoint paths from x to y in G .

Is it true that for every G and every $k \geq 0$, the relation $\xrightarrow{G,k}$ is transitive? That is, is it always the case that if $x \xrightarrow{G,k} y$ and $y \xrightarrow{G,k} z$, then we have $x \xrightarrow{G,k} z$? Give a proof or a counterexample.

40.1.14. Census Rounding

(20 PTS.)

You are consulting for an environmental statistics firm. They collect statistics and publish the collected data in a book. The statistics are about populations of different regions in the world and are recorded in multiples of one million. Examples of such statistics would look like the following table.

Country	A	B	C	Total
grown-up men	11.998	9.083	2.919	24.000
grown-up women	12.983	10.872	3.145	27.000
children	1.019	2.045	0.936	4.000
total	26.000	22.000	7.000	55.000

We will assume here for simplicity that our data is such that all row and column sums are integers. The Census Rounding Problem is to round all data to integers without changing any row or column sum. Each fractional number can be rounded either up or down. For example, a good rounding for our table data would be as follows.

Country	A	B	C	Total
grown-up men	11.000	10.000	3.000	24.000
grown-up women	13.000	10.000	4.000	27.000
children	1.000	2.000	0.000	4.000
total	26.000	22.000	7.000	55.000

- (5 PTS.) Consider first the special case when all data are between 0 and 1. So you have a matrix of fractional numbers between 0 and 1, and your problem is to round each fraction that is between 0 and 1 to either 0 or 1 without changing the row or column sums. Use a flow computation to check if the desired rounding is possible.
- (5 PTS.) Consider the Census Rounding Problem as defined above, where row and column sums are integers, and you want to round each fractional number α to either $\lfloor \alpha \rfloor$ or $\lceil \alpha \rceil$. Use a flow computation to check if the desired rounding is possible.
- (10 PTS.) Prove that the rounding we are looking for in (a) and (b) always exists.

40.1.15. Edge Connectivity

(20 PTS.)

The *edge connectivity* of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how the edge connectivity of an undirected graph $G = (V, E)$ can be determined by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

40.1.16. Maximum Flow By Scaling

(20 PTS.)

Let $G = (V, E)$ be a flow network with source s , sink t , and an integer capacity $c(u, v)$ on each edge $(u, v) \in E$. Let $C = \max_{(u,v) \in E} c(u, v)$.

1. (2 PTS.) Argue that a minimum cut of G has capacity at most $C|E|$.
2. (5 PTS.) For a given number K , show that an augmenting path of capacity at least K can be found in $O(E)$ time, if such a path exists.

The following modification of FORD-FULKERSON-METHOD can be used to compute a maximum flow in G .

```
MAX-FLOW-BY-SCALING( $G, s, t$ )
1    $C \leftarrow \max_{(u,v) \in E} c(u, v)$ 
2   initialize flow  $f$  to 0
3    $K \leftarrow 2^{\lceil \lg C \rceil}$ 
4   while  $K \geq 1$  do {
5       while (there exists an augmenting path  $p$  of
               capacity at least  $K$ ) do {
6           augment flow  $f$  along  $p$ 
7       }
8        $K \leftarrow K/2$ 
9   }
```

3. (3 PTS.) Argue that MAX-FLOW-BY-SCALING returns a maximum flow.
4. (4 PTS.) Show that the capacity of a minimum cut of the residual graph G_f is at most $2K|E|$ each time line 4 is executed.
5. (4 PTS.) Argue that the inner **while** loop of lines 5-6 is executed $O(E)$ times for each value of K .
6. (2 PTS.) Conclude that MAX-FLOW-BY-SCALING can be implemented so that it runs in $O(E^2 \lg C)$ time.

40.1.17. Perfect Matching

(20 PTS.)

- (10 PTS.) A *perfect matching* is a matching in which every vertex is matched. Let $G = (V, E)$ be an undirected bipartite graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the *neighborhood* of X as

$$N(X) = \left\{ y \in V \mid (x, y) \in E \text{ for some } x \in X \right\},$$

that is, the set of vertices adjacent to some member of X . Prove *Hall's theorem*: there exists a perfect matching in G if and only if $|A| \leq |N(A)|$ for every subset $A \subseteq L$.

- (10 PTS.) We say that a bipartite graph $G = (V, E)$, where $V = L \cup R$, is *d-regular* if every vertex $v \in V$ has degree exactly d . Every d -regular bipartite graph has $|L| = |R|$. Prove that every d -regular bipartite graph has a matching of cardinality $|L|$ by arguing that a minimum cut of the corresponding flow network has capacity $|L|$.

40.1.18. Number of augmenting paths

- (10 PTS.) Show that a maximum flow in a network $G = (V, E)$ can always be found by a sequence of at most $|E|$ augmenting paths. [Hint: Determine the paths after finding the maximum flow.]
- (10 PTS.) Suppose that a flow network $G = (V, E)$ has symmetric edges, that is, $(u, v) \in E$ if and only if $(v, u) \in E$. Show that the Edmonds-Karp algorithm terminates after at most $|V||E|/4$ iterations. [Hint: For any edge (u, v) , consider how both $\delta(s, u)$ and $\delta(v, t)$ change between times at which (u, v) is critical.]

40.1.19. Minimum Cut Festival

(20 PTS.)

- Given a multigraph $G(V, E)$, show that an edge can be selected uniform at random from E in time $O(n)$, given access to a source of random bits.
- For any $\alpha \geq 1$, define an α approximate cut in a multigraph G as any cut whose cardinality is within a multiplicative factor α of the cardinality of the min-cut in G . Determine the probability that a single iteration of the randomized algorithm for cuts will produce as output some α -approximate cut in G .
- Using the analysis of the randomized min-cut algorithm, show that the number of distinct min-cuts in a multigraph G cannot exceed $n(n-1)/2$, where n is the number of vertices in G .
- Formulate and prove a similar result of the number of α -approximate cuts in a multigraph G .

40.1.20. Independence Matrix

(10 PTS.)

Consider a 0–1 matrix H with n_1 rows and n_2 columns. We refer to a row or a column of the matrix H as a line. We say that a set of 1's in the matrix H is *independent* if no two of them appear in the same line. We also say that a set of lines in the matrix is a *cover* of H if they include (i.e., “cover”) all the 1's in the matrix. Using the max-flow min-cut theorem on an appropriately defined network, show that the maximum number of independent 1's equals the minimum number of lines in the cover.

40.1.21. Scalar Flow Product

(10 PTS.)

Let f be a flow in a network, and let α be a real number. The *scalar flow product*, denoted by αf , is a function from $V \times V$ to \mathbb{R} defined by

$$(\alpha f)(u, v) = \alpha \cdot f(u, v).$$

Prove that the flows in a network form a *convex set*. That is, show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha)f_2$ for all α in the range $0 \leq \alpha \leq 1$.

40.1.22. Go to school!

Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining if both his children can go to the same school as a maximum-flow problem.

40.1.23. The Hopcroft-Karp Bipartite Matching Algorithm

(20 PTS.)

In this problem, we describe a faster algorithm, due to Hopcroft and Karp, for finding a maximum matching in a bipartite graph. The algorithm runs in $O(\sqrt{VE})$ time. Given an undirected, bipartite graph $G = (V, E)$, where $V = L \cup R$ and all edges have exactly one endpoint in L , let M be a matching in G . We say that a simple path P in G is an *augmenting path* with respect to M if it starts at an unmatched vertex in L , ends at an unmatched vertex in R , and its edges belong alternatively to M and $E - M$. (This definition of an augmenting path is related to, but different from, an augmenting path in a flow network.) In this problem, we treat a path as a sequence of edges, rather than as a sequence of vertices. A shortest augmenting path with respect to a matching M is an augmenting path with a minimum number of edges.

Given two sets A and B , the *symmetric difference* $A \oplus B$ is defined as $(A - B) \cup (B - A)$, that is, the elements that are in exactly one of the two sets.

1. (4 PTS.) Show that if M is a matching and P is an augmenting path with respect to M , then the symmetric difference $M \oplus P$ is a matching and $|M \oplus P| = |M| + 1$. Show that if P_1, P_2, \dots, P_k are vertex-disjoint augmenting paths with respect to M , then the symmetric difference $M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ is a matching with cardinality $|M| + k$.

The general structure of our algorithm is the following:

```
HOPCROFT-KARP( $G$ )
1   $M \leftarrow \emptyset$ 
2  repeat
3      let  $\mathcal{P} \leftarrow \{P_1, P_2, \dots, P_k\}$  be a maximum set of
          vertex-disjoint shortest augmenting paths
          with respect to  $M$ 
4       $M \leftarrow M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ 
5  until  $\mathcal{P} = \emptyset$ 
6  return  $M$ 
```

The remainder of this problem asks you to analyze the number of iterations in the algorithm (that is, the number of iterations in the **repeat** loop) and to describe an implementation of line 3.

2. (4 PTS.) Given two matchings M and M^* in G , show that every vertex in the graph $G' = (V, M \oplus M^*)$ has degree at most 2. Conclude that G' is a disjoint union of simple paths or cycles. Argue that edges in each such simple path or cycle belong alternatively to M or M^* . Prove that if $|M| \leq |M^*|$, then $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to M .

Let l be the length of a shortest augmenting path with respect to a matching M , and let P_1, P_2, \dots, P_k be a maximum set of vertex-disjoint augmenting paths of length l with respect to M . Let $M' = M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$, and suppose that P is a shortest augmenting path with respect to M' .

3. (2 PTS.) Show that if P is vertex-disjoint from P_1, P_2, \dots, P_k , then P has more than l edges.
4. (2 PTS.) Now suppose P is not vertex-disjoint from P_1, P_2, \dots, P_k . Let A be the set of edges $(M \oplus M') \oplus P$. Show that $A = (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P$ and that $|A| \geq (k + 1)l$. Conclude that P has more than l edges.
5. (2 PTS.) Prove that if a shortest augmenting path for M has length l , the size of the maximum matching is at most $|M| + |V|/l$.
6. (2 PTS.) Show that the number of **repeat** loop iterations in the algorithm is at most $2\sqrt{V}$. [Hint: By how much can M grow after iteration number \sqrt{V} ?]
7. (4 PTS.) Give an algorithm that runs in $O(E)$ time to find a maximum set of vertex-disjoint shortest augmenting paths P_1, P_2, \dots, P_k for a given matching M . Conclude that the total running time of HOPCROFT-KARP is $O(\sqrt{VE})$.

40.2. Min Cost Flow

40.2.1. Streaming TV.

(20 PTS.)

You are given a directed graph G , a source vertex s (i.e., a server in the internet), and a set T of vertices (i.e., consumers computers). We would like to broadcast as many TV programs from the server to the customers simultaneously. A single broadcast is a path from the server to one of the customers. The constraint is that no edge or vertex (except from the server) can have two streams going through them.

- 1** (10 PTS.) Provide a polynomial time algorithm that computes the largest number of paths that can be streamed from the server.
- 2** (10 PTS.) Let k be the number of paths computed in (A). Present an algorithm, that in polynomial time, computes a set of k such paths (one end point in the server, the other endpoint is in T) with minimum number of edges.

40.2.2. Transportation Problem.

(20 PTS.)

Let G be a digraph with n vertices and m edges.

In the transportation problem, you are given a set X of x vertices in a graph G , for every vertex $v \in X$ there is a quantity $q_x > 0$ of material available at v . Similarly, there is a set of vertices Y , with associated capacities c_y with each vertex $y \in Y$. Furthermore, every edge of G has an associated distance with it.

The work involved in transporting α units of material on an edge e of length ℓ is $\alpha * \ell$. The problem is to move all the material available in X to the vertices of Y , without violating the capacity constraints of the vertices, while minimizing the overall work involved.

Provide a polynomial time algorithm for this problem. How fast is your algorithm?

40.2.3. Edge Connectivity

(20 PTS.)

The *edge connectivity* of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how the edge connectivity of an undirected graph $G = (V, E)$ can be determined by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

40.2.4. Perfect Matching

(20 PTS.)

- (10 PTS.) A *perfect matching* is a matching in which every vertex is matched. Let $G = (V, E)$ be an undirected bipartite graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the *neighborhood* of X as

$$N(X) = \left\{ y \in V \mid (x, y) \in E \text{ for some } x \in X \right\},$$

that is, the set of vertices adjacent to some member of X . Prove *Hall's theorem*: there exists a perfect matching in G if and only if $|A| \leq |N(A)|$ for every subset $A \subseteq L$.

- (10 PTS.) We say that a bipartite graph $G = (V, E)$, where $V = L \cup R$, is *d-regular* if every vertex $v \in V$ has degree exactly d . Every d -regular bipartite graph has $|L| = |R|$. Prove that every d -regular bipartite graph has a matching of cardinality $|L|$ by arguing that a minimum cut of the corresponding flow network has capacity $|L|$.

40.2.5. Max flow by augmenting

- (10 PTS.) Show that a maximum flow in a network $G = (V, E)$ can always be found by a sequence of at most $|E|$ augmenting paths. [Hint: Determine the paths after finding the maximum flow.]
- (10 PTS.) Suppose that a flow network $G = (V, E)$ has symmetric edges, that is, $(u, v) \in E$ if and only if $(v, u) \in E$. Show that the Edmonds-Karp algorithm terminates after at most $|V||E|/4$ iterations. [Hint: For any edge (u, v) , consider how both $\delta(s, u)$ and $\delta(v, t)$ change between times at which (u, v) is critical.]

40.2.6. And now for something completely different.

(10 PTS.)

Prove that the following problems are NPC or provide a polynomial time algorithm to solve them:

- Given a directed graph G , and two vertices $u, v \in V(G)$, find the maximum number of edge disjoint paths between u and v .
- Given a directed graph G , and two vertices $u, v \in V(G)$, find the maximum number of *vertex* disjoint paths between u and v (the paths are disjoint in their vertices, except of course, for the vertices u and v).

40.2.7. Minimum Cut

(10 PTS.)

Present a *deterministic* algorithm, such that given an undirected graph G , it computes the minimum cut in G . How fast is your algorithm? How does your algorithm compare with the randomized algorithm shown in class?

Chapter 41

Exercises - Miscellaneous

41.1. Data structures

1 Furthest Neighbor (20 PTS.)

Let $P = \{p_1, \dots, p_n\}$ be a set of n points in the plane.

- 1.A. (10 PTS.) A *partition* $\mathcal{P} = (S, T)$ of P is a decomposition of P into two sets $S, T \subseteq P$, such that $P = S \cup T$, and $S \cap T = \emptyset$.

Describe a *deterministic*^① algorithm to compute $m = O(\log n)$ partitions $\mathcal{P}_1, \dots, \mathcal{P}_m$ of P , such that for any pair of distinct points $p, q \in P$, there exists a partition $\mathcal{P}_i = (S_i, T_i)$, where $1 \leq i \leq m$, such that $p \in S_i$ and $q \in T_i$ or vice versa (i.e., $p \in T_i$ and $q \in S_i$). The running time of your algorithm should be $O(n \log n)$.

- 1.B. (10 PTS.) Assume that you are given a black-box \mathcal{B} , such that given a set of points Q in the plane, one can compute in $O(|Q| \log |Q|)$ time, a data-structure \mathcal{X} , such that given any query point w in the plane, one can compute, in $O(\log |Q|)$ time, using the data-structure, the furthest point in Q from w (i.e., this is the point in Q with largest distance from w). To make things interesting, assume that if $w \in Q$, then the data-structure does not work.

Describe an algorithm that uses \mathcal{B} , and such that computes, in $O(n \log^2 n)$ time, for *every* point $p \in P$, its furthest neighbor f_p in $P \setminus \{p\}$.

2 Free lunch. (10 PTS.)

- 2.A. (3 PTS.) Provide a *detailed* description of the procedure that computes the *longest ascending subsequence* in a given sequence of n numbers. The procedure should use only arrays, and should output together with the length of the subsequence, the subsequence itself.
- 2.B. (4 PTS.) Provide a data-structure, that store pairs (a_i, b_i) of numbers, such that an insertion/deletion operation takes $O(\log n)$ time, where n is the total number of elements inserted. And furthermore, given a query interval $[\alpha, \beta]$, it can output in $O(\log n)$ time, the pair realizing

$$\max_{(a_i, b_i) \in S, a_i \in [\alpha, \beta]} b_i,$$

where S is the current set of pairs.

- 2.C. (3 PTS.) Using (b), describe an $O(n \log n)$ time algorithm for computing the longest ascending subsequence given a sequence of n numbers.

41.2. Divide and Conquer

3 Divide-and-Conquer Multiplication

^①There is a very nice and simple randomized algorithm for this problem, you can think about it if you are interested.

- 3.A.** (5 PTS.) Show how to multiply two linear polynomials $ax + b$ and $cx + d$ using only three multiplications. (Hint: One of the multiplications is $(a + b) \cdot (c + d)$.)
- 3.B.** (5 PTS.) Give two divide-and-conquer algorithms for multiplying two polynomials of degree-bound n that run in time $\Theta(n^{\lg 3})$. The first algorithm should divide the input polynomial coefficients into a high half and a low half, and the second algorithm should divide them according to whether their index is odd or even.
- 3.C.** (5 PTS.) Show that two n -bit integers can be multiplied in $O(n^{\lg 3})$ steps, where each step operates on at most a constant number of 1-bit values.

41.3. Fast Fourier Transform

- 4** **3sum** Consider two sets A and B , each having n integers in the range from 0 to $10n$. We wish to compute the *Cartesian sum* of A and B , defined by

$$C = \{x + y : x \in A \text{ and } y \in B\}.$$

Note that the integers in C are in the range from 0 to $20n$. We want to find the elements of C and the number of times each element of C is realized as a sum of elements in A and B . Show that the problem can be solved in $O(n \lg n)$ time. (Hint: Represent A and B as polynomials of degree at most $10n$.)

- 5** **Common subsequence** Given two sequences, a_1, \dots, a_n and b_1, \dots, b_m of real numbers, We want to determine whether there is an $i \geq 0$, such that $b_1 = a_{i+1}, b_2 = a_{i+2}, \dots, b_m = a_{i+m}$. Show how to solve this problem in $O(n \log n)$ time with high probability.

- 6** **Computing Polynomials Quickly** In the following, assume that given two polynomials $p(x), q(x)$ of degree at most n , one can compute the polynomial remainder of $p(x) \bmod q(x)$ in $O(n \log n)$ time. The *remainder* of $r(x) = p(x) \bmod q(x)$ is the unique polynomial of degree smaller than this of $q(x)$, such that $p(x) = q(x) * d(x) + r(x)$, where $d(x)$ is a polynomial.

Let $p(x) = \sum_{i=0}^{n-1} a_i x^i$ be a given polynomial.

- 6.A.** (4 PTS.) Prove that $p(x) \bmod (x - z) = p(z)$, for all z .
- 6.B.** (4 PTS.) We want to evaluate $p(\cdot)$ on the points x_0, x_1, \dots, x_{n-1} . Let

$$P_{ij}(x) = \prod_{k=i}^j (x - x_k)$$

and

$$Q_{ij}(x) = p(x) \bmod P_{ij}(x).$$

Observe that the degree of Q_{ij} is at most $j - i$.

Prove that, for all x , $Q_{kk}(x) = p(x_k)$ and $Q_{0,n-1}(x) = p(x)$.

- 6.C.** (4 PTS.) Prove that for $i \leq k \leq j$, we have

$$\forall x \quad Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$$

and

$$\forall x \quad Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x).$$

- 6.D.** (8 PTS.) Given an $O(n \log^2 n)$ time algorithm to evaluate $p(x_0), \dots, p(x_{n-1})$. Here x_0, \dots, x_{n-1} are n given real numbers.

41.4. Union-Find

7 Linear time Union-Find, (20 PTS.)

- 7.A. (2 PTS.) With path compression and union by rank, during the lifetime of a Union-Find data-structure, how many elements would have rank equal to $\lfloor \lg n - 5 \rfloor$, where there are n elements stored in the data-structure?
- 7.B. (2 PTS.) Same question, for rank $\lfloor (\lg n)/2 \rfloor$.
- 7.C. (4 PTS.) Prove that in a set of n elements, a sequence of n consecutive FIND operations take $O(n)$ time in total.
- 7.D. (2 PTS.)
Write a non-recursive version of FIND with path compression.
- 7.E. (6 PTS.) Show that any sequence of m MAKESET, FIND, and UNION operations, where all the UNION operations appear before any of the FIND operations, takes only $O(m)$ time if both path compression and union by rank are used.
- 7.F. (4 PTS.) What happens in the same situation if only the path compression is used?

8 Off-line Minimum (20 PTS.)

The *off-line minimum problem* asks us to maintain a dynamic set T of elements from the domain $\{1, 2, \dots, n\}$ under the operations INSERT and EXTRACT-MIN. We are given a sequence S of n INSERT and m EXTRACT-MIN calls, where each key in $\{1, 2, \dots, n\}$ is inserted exactly once. We wish to determine which key is returned by each EXTRACT-MIN call. Specifically, we wish to fill in an array *extracted*[1... m], where for $i = 1, 2, \dots, m$, *extracted*[i] is the key returned by the i th EXTRACT-MIN call. The problem is “off-line” in the sense that we are allowed to process the entire sequence S before determining any of the returned keys.

- 8.A. (4 PTS.)
In the following instance of the off-line minimum problem, each INSERT is represented by a number and each EXTRACT-MIN is represented by the letter E:

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5.

Fill in the correct values in the *extracted* array.

- 8.B. (8 PTS.)
To develop an algorithm for this problem, we break the sequence S into homogeneous subsequences. That is, we represent S by
 $I_1, E, I_2, E, I_3, \dots, I_m, E, I_{m+1}$,
where each E represents a single EXTRACT-MIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j , we initially place the keys inserted by these operations into a set K_j , which is empty if I_j is empty. We then do the following.

```
OFF-LINE-MINIMUM( $m, n$ )
1  for  $i \leftarrow 1$  to  $n$ 
2    do determine  $j$  such that  $i \in K_j$ 
3    if  $j \neq m + 1$ 
4      then  $extracted[j] \leftarrow i$ 
5         let  $l$  be the smallest value greater than  $j$  for which set  $K_l$  exists
6          $K_l \leftarrow K_j \cup K_l$ , destroying  $K_j$ 
7  return extracted
```

Argue that the array *extracted* returned by OFF-LINE-MINIMUM is correct.

8.C. (8 PTS.)

Describe how to implement OFF-LINE-MINIMUM efficiently with a disjoint-set data structure. Give a tight bound on the worst-case running time of your implementation.

9 Tarjan's Off-Line Least-Common-Ancestors Algorithm (20 PTS.)

The *least common ancestor* of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T . In the *off-line least-common-ancestors problem*, we are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T , and we wish to determine the least common ancestor of each pair in P .

To solve the off-line least-common-ancestors problem, the following procedure performs a tree walk of T with the initial call $LCA(\text{root}[T])$. Each node is assumed to be colored WHITE prior to the walk.

```

LCA( $u$ )
1  MAKESET( $u$ )
2   $\text{ancestor}[\text{FIND}(u)] \leftarrow u$ 
3  for each child  $v$  of  $u$  in  $T$ 
4      do LCA( $v$ )
5          UNION( $u, v$ )
6           $\text{ancestor}[\text{FIND}(u)] \leftarrow u$ 
7   $\text{color}[u] \leftarrow \text{BLACK}$ 
8  for each node  $v$  such that  $\{u, v\} \in P$ 
9      do if  $\text{color}[v] = \text{BLACK}$ 
10         then print "The least common ancestor of"  $u$  "and"  $v$  "is"  $\text{ancestor}[\text{FIND}(v)]$ 
    
```

9.A. (4 PTS.) Argue that line 10 is executed exactly once for each pair $\{u, v\} \in P$.

9.B. (4 PTS.) Argue that at the time of the call $LCA(u)$, the number of sets in the disjoint-set data structure is equal to the depth of u in T .

9.C. (6 PTS.) Prove that LCA correctly prints the least common ancestor of u and v for each pair $\{u, v\} \in P$.

9.D. (6 PTS.) Analyze the running time of LCA, assuming that we use the implementation of the disjoint-set data structure with path compression and union by rank.

10 Ackermann Function (20 PTS.)

The Ackermann's function $A_i(n)$ is defined as follows:

$$A_i(n) = \begin{cases} 4 & \text{if } n = 1 \\ 4n & \text{if } i = 1 \\ A_{i-1}(A_i(n-1)) & \text{otherwise} \end{cases}$$

Here we define $A(x) = A_x(x)$. And we define $\alpha(n)$ as a pseudo-inverse function of $A(x)$. That is, $\alpha(n)$ is the least x such that $n \leq A(x)$.

10.A. (4 PTS.) Give a precise description of what are the functions: $A_2(n)$, $A_3(n)$, and $A_4(n)$.

10.B. (4 PTS.) What is the number $A(4)$?

10.C. (4 PTS.) **Prove** that $\lim_{n \rightarrow \infty} \frac{\alpha(n)}{\log^*(n)} = 0$.

10.D. (4 PTS.) We define

$$\log^{**} n = \min \left\{ i \geq 1 \mid \underbrace{\log^* \dots \log^* n}_{i \text{ times}} \leq 2 \right\}$$

(i.e., how many times do you have to take \log^* of a number before you get a number smaller than

2). **Prove** that $\lim_{n \rightarrow \infty} \frac{\sqrt{\alpha(n)}}{\log^{**}(n)} = 0$.

10.E. (4 PTS.) **Prove** that $\log(\alpha(n)) \leq \alpha(\log^{**} n)$ for n large enough.

41.5. Lower bounds

11 Sort them Up (20 PTS.)

A sequence of real numbers x_1, \dots, x_n is k -mixed, if there exists a permutation π , such that $x_{\pi(i)} \leq x_{\pi(i+1)}$ and $|\pi(i) - i| \leq k$, for $i = 1, \dots, n-1$.

11.A. (10 PTS.) Give a fast algorithm for sorting x_1, \dots, x_n .

11.B. (10 PTS.) Prove a lower bound in the comparison model on the running time of your algorithm.

12 Another Lower Bound (20 PTS.)

Let $b_1 \leq b_2 \leq b_3 \leq \dots \leq b_k$ be k given sorted numbers, and let A be a set of n arbitrary numbers $A = \{a_1, \dots, a_n\}$, such that $b_1 < a_i < b_k$, for $i = 1, \dots, n$

The rank $v = r(a_i)$ of a_i is the index, such that $b_v < a_i < b_{v+1}$.

Prove, that in the comparison model, any algorithm that outputs the ranks $r(a_1), \dots, r(a_n)$ must take $\Omega(n \log k)$ running time in the worst case.

41.6. Number theory

13 Some number theory. (10 PTS.)

13.A. (5 PTS.) Prove that if $\gcd(m, n) = 1$, then $m^{\phi(n)} + n^{\phi(m)} \equiv 1 \pmod{mn}$.

13.B. (5 PTS.) Give two distinct proofs that there are an infinite number of prime numbers.

14 Even More Number Theory (10 PTS.)

Prove that $|P(n)| = \Omega(n^2)$, where $P(n) = \left\{ (a, b) \mid a, b \in \mathbb{Z}, 0 < a < b \leq n, \gcd(a, b) = 1 \right\}$.

15 Yet Another Number Theory Question (20 PTS.)

15.A. (2 PTS.) Prove that the product of all primes p , for $m < p \leq 2m$ is at most $\binom{2m}{m}$.

15.B. (4 PTS.) **Using (a)**, prove that the number of all primes between m and $2m$ is $O(m/\ln m)$.

15.C. (3 PTS.) **Using (b)**, prove that the number of primes smaller than n is $O(n/\ln n)$.

15.D. (2 PTS.) Prove that if 2^k divides $\binom{2m}{m}$ then $2^k \leq 2m$.

15.E. (5 PTS.) (Hard) Prove that for a prime p , if p^k divides $\binom{2m}{m}$ then $p^k \leq 2m$.

15.F. (4 PTS.) **Using (e)**, prove that the number of primes between 1 and n is $\Omega(n/\ln n)$. (Hint: use the fact that $\binom{2m}{m} \geq 2^{2m}/(2m)$.)

41.7. Sorting networks

16 Lower bound on sorting network (10 PTS.)

Prove that an n -input sorting network must contain at least one comparator between the i th and $(i + 1)$ st lines for all $i = 1, 2, \dots, n - 1$.

17 First sort, then partition

Suppose that we have $2n$ elements $\langle a_1, a_2, \dots, a_{2n} \rangle$ and wish to partition them into the n smallest and the n largest. Prove that we can do this in constant additional depth after separately sorting $\langle a_1, a_2, \dots, a_n \rangle$ and $\langle a_{n+1}, a_{n+2}, \dots, a_{2n} \rangle$.

18 Easy points. (20 PTS.)

Let $S(k)$ be the depth of a sorting network with k inputs, and let $M(k)$ be the depth of a merging network with $2k$ inputs. Suppose that we have a sequence of n numbers to be sorted and we know that every number is within k positions of its correct position in the sorted order, which means that we need to move each number at most $(k - 1)$ positions to sort the inputs. For example, in the sequence 3 2 1 4 5 8 7 6 9, every number is within 3 positions of its correct position. But in sequence 3 2 1 4 5 9 8 7 6, the number 9 and 6 are outside 3 positions of its correct position.

Show that we can sort the n numbers in depth $S(k) + 2M(k)$. (You need to prove your answer is correct.)

19 Matrix Madness (20 PTS.)

We can sort the entries of an $m \times m$ matrix by repeating the following procedure k times:

- (I) Sort each odd-numbered row into monotonically increasing order.
- (II) Sort each even-numbered row into monotonically decreasing order.
- (III) Sort each column into monotonically increasing order.

- 19.A. (8 PTS.) Suppose the matrix contains only 0's and 1's. We repeat the above procedure again and again until no changes occur. In what order should we read the matrix to obtain the sorted output ($m \times m$ numbers in increasing order)? Prove that any $m \times m$ matrix of 0's and 1's will be finally sorted.
- 19.B. (8 PTS.) Prove that by repeating the above procedure, any matrix of real numbers can be sorted. [Hint: Refer to the proof of the zero-one principle.]
- 19.C. (4 PTS.) Suppose k iterations are required for this procedure to sort the $m \times m$ numbers. Give an upper bound for k . The tighter your upper bound the better (prove your bound).

41.8. Max Cut

20 Splitting and splicing

Let $G = (V, E)$ be a graph with n vertices and m edges. A *splitting* of G is a partition of V into two sets V_1, V_2 , such that $V = V_1 \cup V_2$, and $V_1 \cap V_2 = \emptyset$. The cardinality of the split (V_1, V_2) , denoted by $m(V_1, V_2)$, is the number of edges in G that has one vertex in V_1 , and one vertex in V_2 . Namely,

$$m(V_1, V_2) = \left| \left\{ e \mid e = \{uv\} \in E(G), u \in V_1, v \in V_2 \right\} \right|.$$

Let $f(G) = \max_{V_1} m(V_1, V_2)$ be the maximum cardinality of such a split. Describe a deterministic polynomial time algorithm that computes a splitting (V_1, V_2) of G , such that $m(V_1, V_2) \geq f(G)/2$. (Hint: Start from an arbitrary split, and continue in a greedy fashion to improve it.)

Chapter 42

Exercises - Approximation Algorithms

This chapter include problems that are related to approximation algorithms.

42.1. Greedy algorithms as approximation algorithms

42.1.1. Greedy algorithm does not work for TSP with the triangle inequality.

(20 PTS.)

In the greedy Traveling Salesman algorithm, the algorithm starts from a starting vertex $v_1 = s$, and in i -th stage, it goes to the closest vertex to v_i that was not visited yet.

1. (10 PTS.) Show an example that prove that the greedy traveling salesman does not provide any constant factor approximation to the TSP.

Formally, for any constant $c > 0$, provide a complete graph G and positive weights on its edges, such that the length of the greedy TSP is by a factor of (at least) c longer than the length of the shortest TSP of G .

2. (10 PTS.) Show an example, that prove that the greedy traveling salesman does not provide any constant factor approximation to the TSP with *triangle inequality*.

Formally, for any constant $c > 0$, provide a complete graph G , and positive weights on its edges, such that the weights obey the triangle inequality, and the length of the greedy TSP is by a factor of (at least) c longer than the length of the shortest TSP of G . (In particular, *prove* that the triangle inequality holds for the weights you assign to the edges of G .)

42.1.2. Greedy algorithm does not work for **VertexCover**.

(10 PTS.)

Extend the example shown in class for the greedy algorithm for **Vertex Cover**. Namely, for any n , show a graph G_n , with n vertices, for which the greedy **Vertex Cover** algorithm, outputs a vertex cover which is of size $\Omega(\text{Opt}(G_n) \log n)$, where $\text{Opt}(G_n)$ is the cardinality of the smallest **Vertex Cover** of G_n .

42.1.3. Greedy algorithm does not work for independent set.

(20 PTS.)

A natural algorithm, GREEDYINDEPENDENT, for computing maximum independent set in a graph, is to repeatedly remove the vertex of lowest degree in the graph, and add it to the independent set, and remove all its neighbors.

1. (5 PTS.) Show an example, where this algorithm fails to output the optimal solution.
2. (5 PTS.) Let G be a $(k, k + 1)$ -uniform graph (this is a graph where every vertex has degree either k or $k + 1$). Show that the above algorithm outputs an independent set of size $\Omega(n/k)$, where n is the number of vertices in G .

- (5 PTS.) Let G be a graph with average degree δ (i.e., $\delta = 2|E(G)|/|V(G)|$). Prove that the above algorithm outputs an independent set of size $\Omega(n/\delta)$.
- (5 PTS.) For any integer k , present an example of a graph G_k , such that GREEDYINDEPENDENT outputs an independent set of size $\leq |OPT(G_k)|/k$, where $OPT(G_k)$ is the largest independent set in G_k . How many vertices and edges does G_k has? What is the average degree of G_k ?

42.1.4. Greedy algorithm does not work for coloring. Really.

(20 PTS.)

Let G be a graph defined over n vertices, and let the vertices be ordered: v_1, \dots, v_n . Let G_i be the induced subgraph of G on v_1, \dots, v_i . Formally, $G_i = (V_i, E_i)$, where $V_i = \{v_1, \dots, v_i\}$ and

$$E_i = \left\{ uv \in E \mid u, v \in V_i \text{ and } uv \in E(G) \right\}.$$

The greedy coloring algorithm, colors the vertices, one by one, according to their ordering. Let k_i denote the number of colors the algorithm uses to color the first i vertices.

In the i -th iteration, the algorithm considers v_i in the graph G_i . If all the neighbors of v_i in G_i are using all the k_{i-1} colors used to color G_{i-1} , the algorithm introduces a new color (i.e., $k_i = k_{i-1} + 1$) and assigns it to v_i . Otherwise, it assigns v_i one of the colors $1, \dots, k_{i-1}$ (i.e., $k_i = k_{i-1}$).

Give an example of a graph G with n vertices, and an ordering of its vertices, such that even if G can be colored using $O(1)$ (in fact, it is possible to do this with two) colors, the greedy algorithm would color it with $\Omega(n)$ colors. (Hint: consider an ordering where the first two vertices are not connected.)

42.1.5. Greedy coloring does not work even if you do it in the right order.

(20 PTS.)

Given a graph G , with n vertices, let us define an ordering on the vertices of G where the min degree vertex in the graph is last. Formally, we set v_n to be a vertex of minimum degree in G (breaking ties arbitrarily), define the ordering recursively, over the graph $G \setminus v_n$, which is the graph resulting from removing v_n from G . Let v_1, \dots, v_n be the resulting ordering, which is known as MIN LAST ORDERING.

- (10 PTS.) Prove that the greedy coloring algorithm, if applied to a planar graph G , which uses the min last ordering, outputs a coloring that uses 6 colors.^①
- (10 PTS.) Give an example of a graph G_n with $O(n)$ vertices which is 3-colorable, but nevertheless, when colored by the greedy algorithm using min last ordering, the number of colors output is n .

42.2. Approximation for hard problems

42.2.1. Even More on Vertex Cover

- (3 PTS.) Give an example of a graph for which APPROX-VERTEX-COVER always yields a suboptimal solution.
- (2 PTS.) Give an efficient algorithm that finds an optimal vertex cover for a tree in linear time.

^①There is a quadratic time algorithm for coloring planar graphs using 4 colors (i.e., follows from a constructive proof of the four color theorem). Coloring with 5 colors requires slightly more cleverness.

3. (5 PTS.) (Based on CLRS 35.1-3)

Professor Nixon proposes the following heuristic to solve the vertex-cover problem. Repeatedly select a vertex of highest degree, and remove all of its incident edges. Give an example to show that the professor's heuristic does not have an approximation ratio of 2. [Hint: Try a bipartite graph with vertices of uniform degree on the left and vertices of varying degree on the right.]

42.2.2. Maximum Clique

(10 PTS.)

Let $G = (V, E)$ be an undirected graph. For any $k \geq 1$, define $G^{(k)}$ to be the undirected graph $(V^{(k)}, E^{(k)})$, where $V^{(k)}$ is the set of all ordered k -tuples of vertices from V and $E^{(k)}$ is defined so that (v_1, v_2, \dots, v_k) is adjacent to (w_1, w_2, \dots, w_k) if and only if for each i ($1 \leq i \leq k$) either vertex v_i is adjacent to w_i in G , or else $v_i = w_i$.

1. (5 PTS.) Prove that the size of the maximum clique in $G^{(k)}$ is equal to the k -th power of the size of the maximum clique in G .
2. (5 PTS.) Argue that if there is an approximation algorithm that has a constant approximation ratio for finding a maximum-size clique, then there is a fully polynomial time approximation scheme for the problem.

42.2.3. Pack these squares.

(10 PTS.)

Let R be a set of squares. You need to pack them inside the unit square in the plane (i.e., place them inside the square), such that all the squares are interior disjoint. Provide a polynomial time algorithm that outputs a packing that covers at least $OPT/4$ fraction of the unit square, where OPT is the fraction of the unit square covered by the optimal solution.

42.2.4. Smallest Interval

(20 PTS.)

Given a set X of n real numbers x_1, \dots, x_n (no necessarily given in sorted order), and $k > 0$ a parameter (which is not necessarily small). Let $I_k = [a, b]$ be the shortest interval that contains k numbers of X .

1. (5 PTS.) Give a $O(n \log n)$ time algorithm that outputs I_k .
2. (5 PTS.) An interval J is called 2-cover, if it contains at least k points of X , and $|J| \leq 2|I_k|$, where $|J|$ denote the length of J . Give a $O(n \log(n/k))$ expected time algorithm that computes a 2-cover.
3. (10 PTS.) (hard) Give an expected linear time algorithm that outputs a 2-cover of X with high probability.

42.2.5. Rectangles are Forever.

(20 PTS.)

A rectangle in the plane r is called *neat*, if the ratio between its longest edge and shortest edge is bounded by a constant α . Given a set of rectangles R , the induced graph G_R , has the rectangles of R as vertices, and it connect two rectangles if their intersection is not empty.

1. (5 PTS.) (hard?) Given a set R of n neat rectangles in the plane (not necessarily axis parallel), describe a polynomial time algorithm for computing an independent set I in the graph G_R , such that $|I| \geq \beta|X|$, where X is the largest independent set in G_R , and β is a constant that depends only on α . Give an explicit formula for the dependency of β on α . What is the running time of your algorithm?

- (5 PTS.) Let R be a set of rectangles which are axis parallel. Show a polynomial time algorithm for finding the largest independent set in G_R if all the rectangles of R intersects the y -axis.
- (10 PTS.) Let R be a set of axis parallel rectangles. Using (b), show to compute in polynomial time an independent set of rectangles of size $\Omega(k^c)$, where k is the size of the largest independent set in G_R and c is an absolute constant. (Hint: Consider all vertical lines through vertical edges of rectangles of R . Next, show that by picking one of them “cleverly” and using (b), one can perform a divide and conquer to find a large independent set. Define a recurrence on the size of the independent set, and prove a lower bound on the solution of the recurrence.)

42.2.6. Graph coloring revisited

- (5 PTS.) Prove that a graph G with a chromatic number k (i.e., k is the minimal number of colors needed to color G), must have $\Omega(k^2)$ edges.
- (5 PTS.) Prove that a graph G with m edges can be colored using $4\sqrt{m}$ colors.
- (10 PTS.) Describe a polynomial time algorithm that given a graph G , which is 3-colorable, it computes a coloring of G using, say, at most $O(\sqrt{n})$ colors.

Chapter 43

Randomized Algorithms

This chapter include problems on randomized algorithms

43.1. Randomized algorithms

43.1.1. Find k th smallest number.

(20 PTS.)

This question asks you to design and analyze a *randomized incremental* algorithm to select the k th smallest element from a given set of n elements (from a universe with a linear order).

In an *incremental* algorithm, the input consists of a sequence of elements x_1, x_2, \dots, x_n . After any prefix x_1, \dots, x_{i-1} has been considered, the algorithm has computed the k th smallest element in x_1, \dots, x_{i-1} (which is undefined if $i \leq k$), or if appropriate, some other invariant from which the k th smallest element could be determined. This invariant is updated as the next element x_i is considered.

Any incremental algorithm can be *randomized* by first randomly permuting the input sequence, with each permutation equally likely.

- (5 PTS.) Describe an incremental algorithm for computing the k th smallest element.
- (5 PTS.) How many comparisons does your algorithm perform in the worst case?

- (10 PTS.) What is the expected number (over all permutations) of comparisons performed by the randomized version of your algorithm? (Hint: When considering x_i , what is the probability that x_i is smaller than the k th smallest so far?) You should aim for a bound of at most $n + O(k \log(n/k))$. Revise (A) if necessary in order to achieve this.

43.1.2. Minimum Cut Festival

(20 PTS.)

- Given a multigraph $G(V, E)$, show that an edge can be selected uniform at random from E in time $O(n)$, given access to a source of random bits.
- For any $\alpha \geq 1$, define an α approximate cut in a multigraph G as any cut whose cardinality is within a multiplicative factor α of the cardinality of the min-cut in G . Determine the probability that a single iteration of the randomized algorithm for cuts will produce as output some α -approximate cut in G .
- Using the analysis of the randomized min-cut algorithm, show that the number of distinct min-cuts in a multigraph G cannot exceed $n(n-1)/2$, where n is the number of vertices in G .
- Formulate and prove a similar result of the number of α -approximate cuts in a multigraph G .

43.1.3. Adapt min-cut

(20 PTS.)

Consider adapting the min-cut algorithm to the problem of finding an s - t *min-cut* in an undirected graph. In this problem, we are given an undirected graph G together with two distinguished vertices s and t . An s - t min-cut is a set of edges whose removal disconnects s from t ; we seek an edge set of minimum cardinality. As the algorithm proceeds, the vertex s may get amalgamated into a new vertex as the result of an edge being contracted; we call this vertex the s -vertex (initially s itself). Similarly, we have a t -vertex. As we run the contraction algorithm, we ensure that we never contract an edge between the s -vertex and the t -vertex.

- (10 PTS.) Show that there are graphs in which the probability that this algorithm finds an s - t min-cut is exponentially small.
- (10 PTS.) How large can the number of s - t min-cuts in an instance be?

43.1.4. Majority tree

(20 PTS.)

Consider a uniform rooted tree of height h (every leaf is at distance h from the root). The root, as well as any internal node, has 3 children. Each leaf has a boolean value associated with it. Each internal node returns the value returned by the majority of its children. The evaluation problem consists of determining the value of the root; at each step, an algorithm can choose one leaf whose value it wishes to read.

- 1 Show that for any deterministic algorithm, there is an instance (a set of boolean values for the leaves) that forces it to read all $n = 3^h$ leaves. (hint: Consider an adversary argument, where you provide the algorithm with the minimal amount of information as it request bits from you. In particular, one can devise such an adversary algorithm.)
- 2 Consider the recursive randomized algorithm that evaluates two subtrees of the root chosen at random. If the values returned disagree, it proceeds to evaluate the third sub-tree. If they agree, it returns the value they agree on. Show the expected number of leaves read by the algorithm on any instance is at most $n^{0.9}$.

43.1.5. Hashing to Victory

(20 PTS.)

In this question we will investigate the construction of hash table for a set W , where W is static, provided in advance, and we care only for search operations.

1. (2 PTS.) Let $U = \{1, \dots, m\}$, and $p = m + 1$ is a prime.

Let $W \subseteq U$, such that $n = |W|$, and s an integer number larger than n . Let $g_k(x, s) = (kx \bmod p) \bmod s$.

Let $\beta(k, j, s) = \left| \left\{ x \mid x \in W, g_k(x, s) = j \right\} \right|$. Prove that

$$\sum_{k=1}^{p-1} \sum_{j=1}^s \binom{\beta(k, j, s)}{2} < \frac{(p-1)n^2}{s}.$$

2. (2 PTS.) Prove that there exists $k \in U$, such that

$$\sum_{j=1}^s \binom{\beta(k, j, s)}{2} < \frac{n^2}{s}.$$

3. (2 PTS.) Prove that $\sum_{j=1}^n \beta(k, j, n) = |W| = n$.

4. (3 PTS.) Prove that there exists a $k \in U$ such that $\sum_{j=1}^n (\beta(k, j, n))^2 < 3n$.

5. (3 PTS.) Prove that there exists a $k' \in U$, such that the function $h(x) = (k'x \bmod p) \bmod n^2$ is one-to-one when restricted to W .

6. (3 PTS.) Conclude, that one can construct a hash-table for W , of $O(n^2)$, such that there are no collisions, and a search operation can be performed in $O(1)$ time (note that the time here is worst case, also note that the construction time here is quite bad - ignore it).

7. (3 PTS.) Using (d) and (f), conclude that one can build a two-level hash-table that uses $O(n)$ space, and perform a lookup operation in $O(1)$ time (worst case).

43.1.6. Sorting Random Numbers

(20 PTS.)

Suppose we pick a real number x_i at random (uniformly) from the unit interval, for $i = 1, \dots, n$.

1. (5 PTS.) Describe an algorithm with an expected linear running time that sorts x_1, \dots, x_n .

To make this question more interesting, assume that we are going to use some standard sorting algorithm instead (say merge sort), which compares the numbers directly. The binary representation of each x_i can be generated as a potentially infinite series of bits that are the outcome of unbiased coin flips. The idea is to generate only as many bits in this sequence as is necessary for resolving comparisons between different numbers as we sort them. Suppose we have only generated some prefixes of the binary representations of the numbers. Now, when comparing two numbers x_i and x_j , if their current partial binary representation can resolve the comparison, then we are done. Otherwise, they have the same partial binary representations (upto the length of the shorter of the two) and we keep generating more bits for each until they first differ.

1. (10 PTS.) Compute a tight upper bound on the expected number of coin flips or random bits needed for a single comparison.
2. (5 PTS.) Generating bits one at a time like this is probably a bad idea in practice. Give a more practical scheme that generates the numbers in advance, using a small number of random bits, given an upper bound n on the input size. Describe a scheme that works correctly with probability $\geq 1 - n^{-c}$, where c is a prespecified constant.

Chapter 44

Exercises - Linear Programming

This chapter include problems that are related to linear programming.

44.1. Miscellaneous

44.1.1. Slack form

(10 PTS.)

Let L be a linear program given in slack form, with n nonbasic variables N , and m basic variables B . Let N' and B' be a different partition of $N \cup B$, such that $|N' \cup B'| = |N \cup B|$. Show a polynomial time algorithm that computes an equivalent slack form that has N' as the nonbasic variables and B' as the basic variables. How fast is your algorithm?

44.2. Tedious

44.2.1. Tedious Computations

(20 PTS.)

Provide *detailed* solutions for the following problems, showing each pivoting stage separately.

1. (5 PTS.)

$$\begin{aligned} &\text{maximize } 6x_1 + 8x_2 + 5x_3 + 9x_4 \\ &\text{subject to} \\ &2x_1 + x_2 + x_3 + 3x_4 \leq 5 \\ &x_1 + 3x_2 + x_3 + 2x_4 \leq 3 \\ &x_1, x_2, x_3, x_4 \geq 0. \end{aligned}$$

2. (5 PTS.)

$$\begin{aligned} &\text{maximize } 2x_1 + x_2 \\ &\text{subject to} \\ &2x_1 + x_2 \leq 4 \\ &2x_1 + 3x_2 \leq 3 \\ &4x_1 + x_2 \leq 5 \\ &x_1 + 5x_2 \leq 1 \\ &x_1, x_2 \geq 0. \end{aligned}$$

3. (5 PTS.)

$$\begin{aligned} &\text{maximize } 6x_1 + 8x_2 + 5x_3 + 9x_4 \\ &\text{subject to} \\ &x_1 + x_2 + x_3 + x_4 = 1 \\ &x_1, x_2, x_3, x_4 \geq 0. \end{aligned}$$

4. (5 PTS.)

$$\text{minimize } x_{12} + 8x_{13} + 9x_{14} + 2x_{23} + 7x_{24} + 3x_{34}$$

subject to

$$\begin{aligned}x_{12} + x_{13} + x_{14} &\geq 1 \\ -x_{12} + x_{23} + x_{24} &= 0 \\ -x_{13} - x_{23} + x_{34} &= 0 \\ x_{14} + x_{24} + x_{34} &\leq 1 \\ x_{12}, x_{13}, \dots, x_{34} &\geq 0.\end{aligned}$$

44.2.2. Linear Programming for a Graph

1. (3 PTS.) Given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow \mathcal{R}$ mapping edges to real-valued weights, a source vertex s , and a destination vertex t . Show how to compute the value $d[t]$, which is the weight of a shortest path from s to t , by linear programming.
2. (4 PTS.) Given a graph G as in (a), write a linear program to compute $d[v]$, which is the shortest-path weight from s to v , for each vertex $v \in V$.

3. (4 PTS.) In the *minimum-cost multicommodity-flow problem*, we are given a directed graph $G = (V, E)$, in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$ and a cost $\alpha(u, v)$. As in the multicommodity-flow problem (Chapter 29.2, CLRS), we are given k different commodities, K_1, K_2, \dots, K_k , where commodity i is specified by the triple $K_i = (s_i, t_i, d_i)$. Here s_i is the source of commodity i , t_i is the sink of commodity i , and d_i is the demand, which is the desired flow value for commodity i from s_i to t_i . We define a flow for commodity i , denoted by f_i , (so that $f_i(u, v)$ is the flow of commodity i from vertex u to vertex v) to be a real-valued function that satisfies the flow-conservation, skew-symmetry, and capacity constraints. We now define $f(u, v)$, the *aggregate flow*, to be sum of the various commodity flows, so that $f(u, v) = \sum_{i=1}^k f_i(u, v)$. The aggregate flow on edge (u, v) must be no more than the capacity of edge (u, v) .

The cost of a flow is $\sum_{u,v \in V} f(u, v)$, and the goal is to find the feasible flow of minimum cost. Express this problem as a linear program.

44.2.3. Linear programming

(20 PTS.)

1. (10 PTS.) Show the following problem in NP-hard.

Integer Linear Programming

Instance: A linear program in standard form, in which A and B contain only integers.

Question: Is there a solution for the linear program, in which the x must take integer values?

2. (5 PTS.) A steel company must decide how to allocate next week's time on a rolling mill, which is a machine that takes unfinished slabs of steel as input and produce either of two semi-finished products: bands and coils. The mill's two products come off the rolling line at different rates:

Bands 200 tons/hr
Coils 140 tons/hr.

They also produce different profits:

Bands \$ 25/ton
Coils \$ 30/ton.

Based on current booked orders, the following upper bounds are placed on the amount of each product to produce:

Bands 6000 tons
Coils 4000 tons.

Given that there are 40 hours of production time available this week, the problem is to decide how many tons of bands and how many tons of coils should be produced to yield the greatest profit. Formulate this problem as a linear programming problem. Can you solve this problem by inspection?

3. (5 PTS.) A small airline, Ivy Air, flies between three cities: Ithaca (a small town in upstate New York), Newark (an eyesore in beautiful New Jersey), and Boston (a yuppie town in Massachusetts). They offer several flights but, for this problem, let us focus on the Friday afternoon flight that departs from Ithaca, stops in Newark, and continues to Boston. There are three types of passengers:
- (a) Those traveling from Ithaca to Newark (god only knows why).
 - (b) Those traveling from Newark to Boston (a very good idea).
 - (c) Those traveling from Ithaca to Boston (it depends on who you know).

The aircraft is a small commuter plane that seats 30 passengers. The airline offers three fare classes:

- (a) Y class: full coach.
- (b) B class: nonrefundable.
- (c) M class: nonrefundable, 3-week advanced purchase.

Ticket prices, which are largely determined by external influences (i.e., competitors), have been set and advertised as follows:

	Ithaca-Newark	Newark-Boston	Ithaca-Boston
Y	300	160	360
B	220	130	280
M	100	80	140

Based on past experience, demand forecasters at Ivy Air have determined the following upper bounds on the number of potential customers in each of the 9 possible origin-destination/fare-class combinations:

	Ithaca-Newark	Newark-Boston	Ithaca-Boston
Y	4	8	3
B	8	13	10
M	22	20	18

The goal is to decide how many tickets from each of the 9 origin/destination/fare-class combinations to sell. The constraints are that the plane cannot be overbooked on either the two legs of the flight and that the number of tickets made available cannot exceed the forecasted maximum demand. The objective is to maximize the revenue. Formulate this problem as a linear programming problem.

44.2.4. Distinguishing between probabilities

(5 PTS.) Suppose that Y is a random variable taking on one of the n known values:

$$a_1, a_2, \dots, a_n.$$

Suppose we know that Y either has distribution p given by

$$\mathcal{P}(Y = a_j) = p_j$$

or it has distribution q given by

$$\mathcal{P}(Y = a_j) = q_j.$$

Of course, the numbers $p_j, j = 1, 2, \dots, n$ are nonnegative and sum to one. The same is true for the q_j 's. Based on a single observation of Y , we wish to guess whether it has distribution p or distribution q . That is, for each possible outcome a_j , we will assert with probability x_j that the distribution is p and with probability $1 - x_j$ that the distribution is q . We wish to determine the probabilities $x_j, j = 1, 2, \dots, n$, such that the probability of saying the distribution is p when in fact it is q has probability no larger than β , where β is some small positive value (such as 0.05). Furthermore, given this constraint, we wish to maximize the probability that we say the distribution is p when in fact it is p . Formulate this maximization problem as a linear programming problem.

44.2.5. Strong duality.

(20 PTS.)

Consider a directed graph G with source vertex s and target vertex t and associated costs $\text{cost}(\cdot) \geq 0$ on the edges. Let \mathcal{P} denote the set of all the directed (simple) paths from s to t in G .

Consider the following (very large) integer program:

$$\begin{aligned} & \text{minimize} && \sum_{e \in E(G)} \text{cost}(e)x_e \\ & \text{subject to} && x_e \in \{0, 1\} \quad \forall e \in E(G) \\ & && \sum_{e \in \pi} x_e \geq 1 \quad \forall \pi \in \mathcal{P}. \end{aligned}$$

- 1 (5 PTS.) What does this IP compute?
- 2 (5 PTS.) Write down the relaxation of this IP into a linear program.
- 3 (5 PTS.) Write down the dual of the LP from (B). What is the interpretation of this new LP? What is it computing for the graph G (prove your answer)?
- 4 (5 PTS.) The strong duality theorem states the following.

Theorem 44.2.1. *If the primal LP problem has an optimal solution $x^* = (x_1^*, \dots, x_n^*)$ then the dual also has an optimal solution, $y^* = (y_1^*, \dots, y_m^*)$, such that*

$$\sum_j c_j x_j^* = \sum_i b_i y_i^*.$$

In the context of (A)-(C) what result is implied by this theorem if we apply it to the primal LP and its dual above? (For this, you can assume that the optimal solution to the LP of (B) is integral – which is not quite true – things are slightly more complicated than that.)

Chapter 45

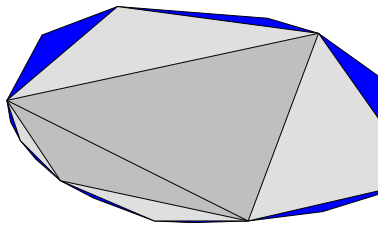
Exercises - Computational Geometry

This chapter include problems that are related to computational geometry.

45.1. Misc

1 Nearest Point to a Polygon (20 PTS.)

Given a convex polygon P , its balanced triangulation is created by recursively triangulating the convex polygon P' defined by its even vertices, and finally adding consecutive diagonals between even points. For example:



Alternative interpretation of this construction, is that we create a sequence of polygons where P_0 is the highest level polygon (a quadrangle in the above example), and P_i is the refinement of P_{i-1} till $P_{\lceil \log n \rceil} = P$.

1. (5 PTS.) Given a polygon P , show how to compute its balanced triangulation in linear time.
2. (15 PTS.) Let T be the dual tree to the balanced triangulation. Show how to use T and the balanced triangulation to answer a query to decide whether point q is inside P or outside it. The query time should be $O(\log n)$, where n is the number of vertices of P . (Hint: use T to maintain the closest point in P_i to q , and use this to decide in constant time what is the closest point in P_{i+1} to q .)

2 Sweeping (20 PTS.)

- 2.A. (5 PTS.) Given two x -monotone polygons, show how to compute their intersection polygon (which might be made out of several connected components) in $O(n)$ time, where n is the total number of vertices of P and X .
- 2.B. (5 PTS.) You are given a set \mathcal{H} of n half-planes (a half plane is the region defined by a line - it is either all the points above a given line, or below it). Show an algorithm to compute the *convex* polygon $\cap_{h \in \mathcal{H}} h$ in $O(n \log n)$ time. (Hint: use (a).)
- 2.C. (10 PTS.) Given two simple polygons P and Q , show how to compute their intersection polygon. How fast is your algorithm?
What the maximum number of connected components of the polygon $P \cap Q$ (provide a tight upper and lower bounds)?

45.1.1. Robot Navigation

(20 PTS.)

Given a set S of m simple polygons in the plane (called obstacles), with total complexity n , and start point s and end point t , find the shortest path between s and t (this is the path that a robot would take to move from s to t).

1. (5 PTS.) For a point $q \in \mathbb{R}^2$, which is not contained in any of the obstacles, the *visibility polygon* of q , is the set of all the points in the plane that are visible from q . Show how to compute this visibility polygon in $O(n \log n)$ time.
2. (5 PTS.) Show a $O(n^3)$ time algorithm for this problem. (Hint: Consider the shortest path, and analyze its structure. Build an appropriate graph, and do a Dijkstra in this graph.)
3. (10 PTS.) Show a $O(n^2 \log n)$ time algorithm for this problem.

45.1.2. Point-Location

Given a x -monotone polygonal chain C with n vertices, show how to preprocess it in linear time, such that given a query point q , one can decide, in $O(\log n)$ time, whether q is below and above C , and what is the segment of C that intersects the vertical line that passes through q . Show how to use this to decide, in $O(\log n)$ whether a point p is inside a x -monotone polygon P with n vertices. Why would this method be preferable to the balanced triangulation used in the previous question (when used on a convex polygon)?

45.1.3. Convexity revisited.

- 2.A. Prove that for any set S of four points in the plane, there exists a partition of S into two subsets S_1, S_2 , such that $\text{CH}(S_1) \cap \text{CH}(S_2) \neq \emptyset$.
- 2.B. Prove that any point x which is a convex combination of n points p_1, \dots, p_n in the plane, can be defined as a convex combination of three points of p_1, \dots, p_n . (Hint: use (a) and induction on the number of points.)
- 2.C. Prove that for any set S of $d + 2$ points in \mathbb{R}^d , there exists a partition of S into two subsets S_1, S_2 , such that $\text{CH}(S_1) \cap \text{CH}(S_2) \neq \emptyset$, $S = S_1 \cup S_2$, and $S_1 \cap S_2 = \emptyset$. (Hint: Use (a) and induction on the dimension.)

3 **Covered by triangles** You are given a set of n triangles in the plane, show an algorithm, as fast as possible, that decides whether the square $[0, 1] \times [0, 1]$ is completely covered by the triangles.

45.1.4. Nearest Neighbor

Let P be a set of n points in the plane. For a point $p \in P$, its nearest neighbor in P , is the point in $P \setminus \{p\}$ which has the smallest distance to p . Show how to compute for every point in P its nearest neighbor in $O(n \log n)$ time.

Chapter 46

Exercises - Entropy

46.0.1. Compress a sequence.

We wish to compress a sequence of independent, identically distributed random variables X_1, X_2, \dots . Each X_j takes on one of n values. The i th value occurs with probability p_i , where $p_1 \geq p_2 \geq \dots \geq p_n$. The result is compressed as follows. Set

$$T_i = \sum_{j=1}^{i-1} p_j,$$

and let the i th codeword be the first $\lceil \lg(1/p_i) \rceil$ bits of T_i . Start with an empty string, and consider X_j in order. If X_j takes on the i th value, append the i th codeword to the end of the string.

- 1 Show that no codeword is the prefix of any other codeword.
- 2 Let Z be the average number of bits appended for each random variable X_j . Show that

$$\mathbb{H}(X_j) \leq z \leq \mathbb{H}(X_j) + 1.$$

46.0.2. Arithmetic coding

Arithmetic coding is a standard compression method. In the case when the string to be compressed is a sequence of biased coin flips, it can be described as follows. Suppose that we have a sequence of bits $X = (X_1, X_2, \dots, X_n)$, where each X_i is independently 0 with probability p and 1 with probability $1 - p$. The sequences can be ordered lexicographically, so for $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_n)$, we say that $x < y$ if $x_i = 0$ and $y_i = 1$ in the first coordinate i such that $x_i \neq y_i$. If $z(x)$ is the number of zeroes in the string x , then define $p(x) = p^{z(x)}(1 - p)^{n - z(x)}$ and

$$q(x) = \sum_{y < x} p(y).$$

- 1 Suppose we are given $X = (X_1, X_2, \dots, X_n)$. Explain how to compute $q(X)$ in time $O(n)$ (assume that any reasonable operation on real numbers takes constant time).
- 2 Argue that the intervals $[q(x), q(x) + p(x))$ are disjoint subintervals of $[0, 1)$.
- 3 Given (A) and (B), the sequence X can be represented by any point in the interval $I(X) = [q(X), q(X) + p(X))$. Show that we can choose a codeword in $I(X)$ with $\lceil \lg(1/p(X)) \rceil + 1$ binary decimal digits to represent X in such a way that no codeword is the prefix of any other codeword.
- 4 Given a codeword chosen as in (C), explain how to decompress it to determine the corresponding sequence (X_1, X_2, \dots, X_n) .
- 5 Using the Chernoff inequality, argue that $\lg(1/p(X))$ is close to $n\mathbb{H}(p)$ with high probability. Thus, this approach yields an effective compression scheme.

46.0.3. Computing entropy.

1. Let $S = \sum_{i=1}^{10} 1/i^2$. Consider a random variable X such that $\mathbb{P}[X = i] = 1/(Si^2)$, for $i = 1, \dots, 10$. Compute $\mathbb{H}(X)$.
2. Let $S = \sum_{i=1}^{10} 1/i^3$. Consider a random variable X such that $\mathbb{P}[X = i] = 1/(Si^3)$, for $i = 1, \dots, 10$. Compute $\mathbb{H}(X)$.
3. Let $S(\alpha) = \sum_{i=1}^{10} 1/i^\alpha$, for $\alpha > 1$. Consider a random variable X such that $\mathbb{P}[X = i] = 1/(S(\alpha)i^\alpha)$, for $i = 1, \dots, 10$. Prove that $\mathbb{H}(X)$ is either increasing or decreasing as a function of α (you can assume that α is an integer).

46.0.4. When is entropy maximized?

Consider an n -sided die, where the i th face comes up with probability p_i . Show that the entropy of a die roll is maximized when each face comes up with equal probability $1/n$.

46.0.5. Condition entropy.

The *conditional entropy* $\mathbb{H}(Y|X)$ is defined by

$$\mathbb{H}(Y|X) = \sum_{x,y} \mathbb{P}[(X = x) \cap (Y = y)] \lg \frac{1}{\mathbb{P}[Y = y|X = x]}.$$

If $Z = (X, Y)$, prove that

$$\mathbb{H}(Z) = \mathbb{H}(X) + \mathbb{H}(Y|X).$$

46.0.6. Improved randomness extraction.

We have shown that we can extract, on average, at least $\lfloor \lg m \rfloor - 1$ independent, unbiased bits from a number chosen uniformly at random from $\{0, \dots, m-1\}$. It follows that if we have k numbers chosen independently and uniformly at random from $\{0, \dots, m-1\}$ then we can extract, on average, at least $k \lfloor \lg m \rfloor - k$ independent, unbiased bits from them. Give a better procedure that extracts, on average, at least $k \lfloor \lg m \rfloor - 1$ independent, unbiased bits from these numbers.

46.0.7. Kraft inequality.

Assume you have a (valid) prefix code with n codewords, where the i th codeword is made out of ℓ_i bits. Prove that

$$\sum_{i=1}^n \frac{1}{2^{\ell_i}} \leq 1.$$

Part XI

Homeworks/midterm/final

Chapter 47

Fall 2001

47.1. Homeworks

47.1.1. Homework 0

- 1** 1. Prove that any positive integer can be written as the sum of distinct powers of 2. For example: $42 = 2^5 + 2^3 + 2^1$, $25 = 2^4 + 2^3 + 2^0$, $17 = 2^4 + 2^0$. [Hint: ‘Write the number in binary’ is not a proof; it just restates the problem.]
2. Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers—if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, $17 = F_7 + F_4 + F_2$.
3. Prove that *any* integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example: $42 = 3^4 - 3^3 - 3^2 - 3^1$, $25 = 3^3 - 3^1 + 3^0$, $17 = 3^3 - 3^2 - 3^0$.
- 2** Sort the following 26 functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice.

1	n	n^2	$\lg n$	$\lg^* n$
$2^{2^{\lg n+1}}$	$\lg^* 2^n$	$2^{\lg^* n}$	$2^{\sqrt{\lg n}}$	$\lfloor \lg(n!) \rfloor$
$\lfloor \lg n \rfloor!$	$n^{\lg n}$	$(\lg n)^n$	n^n	$(\lg n)^{\lg n}$
$n^{1/\lg n}$	$n^{\lg \lg n}$	$\log_{1000} n$	$\lg^{1000} n$	$\lg^{(1000)} n$
$(1 + \frac{1}{1000})^n$	$n^{1/1000}$	$n^{1/\lg n}$	$(1 + 1/n)^n$	$(n + 1)^3 - n^3$
$n^{2.5}$				

To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2 , n , $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

3 Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied. Extra credit will be given for more exact solutions.

1. $A(n) = 7A(n/3) + n \log \log n$
2. $B(n) = \min_{0 < k < n} (B(k) + B(n - k) + 7)$.
3. $C(n) = 4C(\lfloor n/2 \rfloor + 5) + n^2$
4. $D(n) = D(n - 1) + 1/n$
5. $E(n) = -E(n - 1) + 1/n$
6. $F(n) = 2F(n/\log n) + 6$ (**HARD**)
7. $G(n) = n + 2\sqrt{n} \cdot G(\sqrt{n})$

4 This problem asks you to simplify some recursively defined boolean formulas as much as possible. In each case, prove that your answer is correct. Each proof can be just a few sentences long, but it must be a *proof*.

1. Suppose $\alpha_0 = p$, $\alpha_1 = q$, and $\alpha_n = (\alpha_{n-2} \wedge \alpha_{n-1})$ for all $n \geq 2$. Simplify α_n as much as possible. [*Hint: What is α_5 ?*]
2. Suppose $\beta_0 = p$, $\beta_1 = q$, and $\beta_n = (\beta_{n-2} \Leftrightarrow \beta_{n-1})$ for all $n \geq 2$. Simplify β_n as much as possible. [*Hint: What is β_5 ?*]
3. Suppose $\gamma_0 = p$, $\gamma_1 = q$, and $\gamma_n = (\gamma_{n-2} \Rightarrow \gamma_{n-1})$ for all $n \geq 2$. Simplify γ_n as much as possible. [*Hint: What is γ_5 ?*]
4. Suppose $\delta_0 = p$, $\delta_1 = q$, and $\delta_n = (\delta_{n-2} \bowtie \delta_{n-1})$ for all $n \geq 2$, where \bowtie is some boolean function with two arguments. Find a boolean function \bowtie such that $\delta_n = \delta_m$ if and only if $n - m$ is a multiple of 4. [*Hint: There is only one such function.*]

5 Every year, upon their arrival at Hogwarts School of Witchcraft and Wizardry, new students are sorted into one of four houses (Gryffindor, Hufflepuff, Ravenclaw, or Slytherin) by the Hogwarts Sorting Hat. The student puts the Hat on their head, and the Hat tells the student which house they will join. This year, a failed experiment by Fred and George Weasley filled almost all of Hogwarts with sticky brown goo, mere moments before the annual Sorting. As a result, the Sorting had to take place in the basement hallways, where there was so little room to move that the students had to stand in a long line.

After everyone learned what house they were in, the students tried to group together by house, but there was too little room in the hallway for more than one student to move at a time. Fortunately, the Sorting Hat took CS 373 many years ago, so it knew how to group the students as quickly as possible. What method did the Sorting Hat use?

1. More formally, you are given an array of n items, where each item has one of four possible values, possibly with a pointer to some additional data. Describe an algorithm^① that rearranges the items into four clusters in $O(n)$ time using only $O(1)$ extra space.

^①Since you've read the Homework Instructions, you know what the phrase 'describe an algorithm' means. Right?

<i>G</i>	<i>H</i>	<i>R</i>	<i>R</i>	<i>G</i>	<i>G</i>	<i>R</i>	<i>G</i>	<i>H</i>	<i>H</i>	<i>R</i>	<i>S</i>	<i>R</i>	<i>R</i>	<i>H</i>	<i>G</i>	<i>S</i>	<i>H</i>	<i>G</i>	<i>G</i>
Harry	Ann	Bob	Tina	Chad	Bill	Lisa	Ekta	Bart	Jim	John	Jeff	Liz	Mary	Dawn	Nick	Kim	Fox	Dana	Mel

<i>G</i>	<i>G</i>	<i>G</i>	<i>G</i>	<i>G</i>	<i>G</i>	<i>G</i>	<i>H</i>	<i>H</i>	<i>H</i>	<i>H</i>	<i>H</i>	<i>R</i>	<i>R</i>	<i>R</i>	<i>R</i>	<i>R</i>	<i>R</i>	<i>S</i>	<i>S</i>
Harry	Ekta	Bill	Chad	Nick	Mel	Dana	Fox	Ann	Jim	Dawn	Bart	Lisa	Tina	John	Bob	Liz	Mary	Kim	Jeff

- Describe an algorithm for the case where there are k possible values (i.e., $1, 2, \dots, k$) that rearranges the items using only $O(\log k)$ extra space. How fast is your algorithm? (A faster algorithm would get more credit)
- Describe a faster algorithm (if possible) for the case when $O(k)$ extra space is allowed. How fast is your algorithm?
(HARD)
- Optional practice exercise - *no credit*: Provide a fast algorithm that uses only $O(1)$ additional space for the case where there are k possible values.

6 [This problem is required only for graduate students; undergrads can submit a solution for extra credit.]

Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, \dots , 52 of clubs. (They’re big cards.) Penn shuffles the deck until each of the $52!$ possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the five of clubs.

- On average, how many cards does Penn give Teller?
- On average, what is the smallest-numbered card that Penn gives Teller? **(HARD)**
- On average, what is the largest-numbered card that Penn gives Teller?

[Hint: Solve for an n -card deck and then set $n = 52$.] In each case, give *exact* answers and prove that they are correct. If you have to appeal to “intuition” or “common sense”, your answers are probably wrong!

Practice Problems

7 Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m .

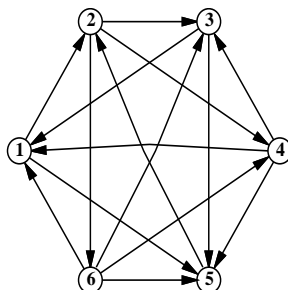
- F_n is even if and only if n is divisible by 3.
- $\sum_{i=0}^n F_i = F_{n+2} - 1$
- $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$ **(Really HARD)**
- If n is an integer multiple of m , then F_n is an integer multiple of F_m .

8 1. Prove the following identity by induction:

$$\binom{2n}{n} = \sum_{k=0}^n \binom{n}{k} \binom{n}{n-k}.$$

- Give a non-inductive combinatorial proof of the same identity, by showing that the two sides of the equation count exactly the same thing in two different ways. There is a correct one-sentence proof.

- 9** A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once. Prove that every tournament contains at least one Hamiltonian path.



A six-vertex tournament containing the Hamiltonian path $6 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1$.

- 10** Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied. Extra credit will be given for more exact solutions.

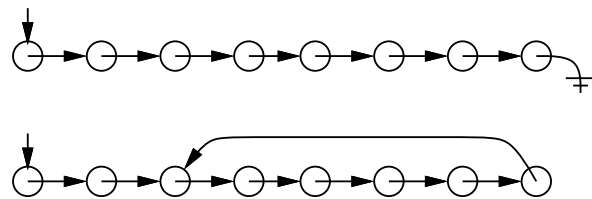
1. $A(n) = A(n/2) + n$
2. $B(n) = 2B(n/2) + n$ (**Really HARD**)
3. $C(n) = n + \frac{1}{2}(C(n-1) + C(3n/4))$
4. $D(n) = \max_{n/3 < k < 2n/3} (D(k) + D(n-k) + n)$ (**HARD**)
5. $E(n) = 2E(n/2) + n/\lg n$ (**HARD**)
6. $F(n) = \frac{F(n-1)}{F(n-2)}$, where $F(1) = 1$ and $F(2) = 2$. (**HARD**)
7. $G(n) = G(n/2) + G(n/4) + G(n/6) + G(n/12) + n$ [Hint: $\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} = 1$.] (**HARD**)
8. $H(n) = n + \sqrt{n} \cdot H(\sqrt{n})$ (**HARD**)
9. $I(n) = (n-1)(I(n-1) + I(n-2))$, where $F(0) = F(1) = 1$ (**HARD**)
10. $J(n) = 8J(n-1) - 15J(n-2) + 1$

- 11**
1. Prove that $2^{\lceil \lg n \rceil + \lfloor \lg n \rfloor} = \Theta(n^2)$.
 2. Prove or disprove: $2^{\lfloor \lg n \rfloor} = \Theta(2^{\lceil \lg n \rceil})$.
 3. Prove or disprove: $2^{2^{\lfloor \lg \lg n \rfloor}} = \Theta(2^{2^{\lceil \lg \lg n \rceil}})$.
 4. Prove or disprove: If $f(n) = O(g(n))$, then $\log(f(n)) = O(\log(g(n)))$.
 5. Prove or disprove: If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.
(**HARD**)
 6. Prove that $\log^k n = o(n^{1/k})$ for any positive integer k .

- 12** Evaluate the following summations; simplify your answers as much as possible. Significant partial credit will be given for answers in the form $\Theta(f(n))$ for some recognizable function $f(n)$.

1. $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{i}$
(HARD)
2. $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{j}$
3. $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{k}$

13 Suppose you have a pointer to the head of singly linked list. Normally, each node in the list only has a pointer to the next element, and the last node's pointer is NULL. Unfortunately, your list might have been corrupted by a bug in somebody else's code², so that the last node has a pointer back to some other node in the list instead.

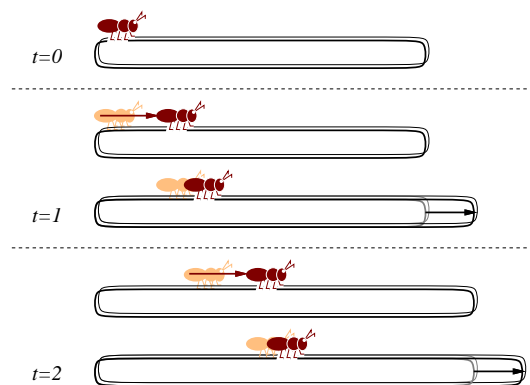


Top: A standard linked list. Bottom: A corrupted linked list.

Describe an algorithm that determines whether the linked list is corrupted or not. Your algorithm must not modify the list. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the list, and use $O(1)$ extra space (not counting the list itself).

(HARD)

14 An ant is walking along a rubber band, starting at the left end. Once every second, the ant walks one inch to the right, and then you make the rubber band one inch longer by pulling on the right end. The rubber band stretches uniformly, so stretching the rubber band also pulls the ant to the right. The initial length of the rubber band is n inches, so after t seconds, the rubber band is $n + t$ inches long.



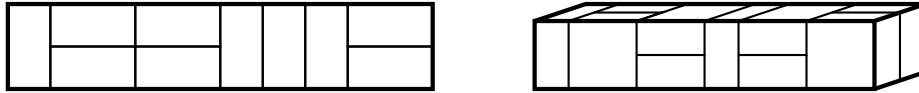
Every second, the ant walks an inch, and then the rubber band is stretched an inch longer.

1. How far has the ant moved after t seconds, as a function of n and t ? Set up a recurrence and (for full credit) give an *exact* closed-form solution. [Hint: What *fraction* of the rubber band's length has the ant walked?]

²After all, *your* code is always completely 100% bug-free. Isn't that right, Mr. Gates?

2. How long does it take the ant to get to the right end of the rubber band? For full credit, give an answer of the form $f(n) + \Theta(1)$ for some explicit function $f(n)$.

- 15**
1. A *domino* is a 2×1 or 1×2 rectangle. How many different ways are there to completely fill a $2 \times n$ rectangle with n dominos? Set up a recurrence relation and give an *exact* closed-form solution.
 2. A *slab* is a three-dimensional box with dimensions $1 \times 2 \times 2$, $2 \times 1 \times 2$, or $2 \times 2 \times 1$. How many different ways are there to fill a $2 \times 2 \times n$ box with n slabs? Set up a recurrence relation and give an *exact* closed-form solution.



A 2×10 rectangle filled with ten dominos, and a $2 \times 2 \times 10$ box filled with ten slabs.

- 16** Professor George O’Jungle has a favorite 26-node binary tree, whose nodes are labeled by letters of the alphabet. The preorder and postorder sequences of nodes are as follows:

preorder: M N H C R S K W T G D X I Y A J P O E Z V B U L Q F

postorder: C W T K S G R H D N A O E P J Y Z I B Q L F U V X M

Draw Professor O’Jungle’s binary tree, and give the inorder sequence of nodes.

- 17** Alice and Bob each have a fair n -sided die. Alice rolls her die once. Bob then repeatedly throws his die until he rolls a number at least as big as the number Alice rolled. Each time Bob rolls, he pays Alice \$1. (For example, if Alice rolls a 5, and Bob rolls a 4, then a 3, then a 1, then a 5, the game ends and Alice gets \$4. If Alice rolls a 1, then no matter what Bob rolls, the game will end immediately, and Alice will get \$1.)

Exactly how much money does Alice expect to win at this game? Prove that your answer is correct. If you have to appeal to ‘intuition’ or ‘common sense’, your answer is probably wrong!

- 18** Prove that for any nonnegative parameters a and b , the following algorithms terminate and produce identical output. Also, provide bounds on the running times of those algorithms. Can you imagine any reason why WEIRDEUCLID would be preferable to FASTEUCLID?

```
SLOWEUCLID( $a, b$ ) :
  if  $b > a$ 
    return SLOWEUCLID( $b, a$ )
  else if  $b = 0$ 
    return  $a$ 
  else
    return SLOWEUCLID( $b, a - b$ )
```

```
FASTEUCLID( $a, b$ ) :
  if  $b = 0$ 
    return  $a$ 
  else
    return FASTEUCLID( $b, a \bmod b$ )
```

```

WEIRDEUCLID( $a, b$ ) :
  if  $b = 0$ 
    return  $a$ 
  if  $a$  is even and  $b$  is even
    return  $2 * \text{WEIRDEUCLID}(a/2, b/2)$ 
  if  $a$  is even and  $b$  is odd
    return  $\text{WEIRDEUCLID}(a/2, b)$ 
  if  $a$  is odd and  $b$  is even
    return  $\text{WEIRDEUCLID}(a, b/2)$ 
  if  $b > a$ 
    return  $\text{WEIRDEUCLID}(b - a, a)$ 
  else
    return  $\text{WEIRDEUCLID}(a - b, b)$ 

```

47.1.2. Homework 1

Required Problems

- 1** Consider the following strange sorting algorithm. For simplicity we will assume that n is always some positive power of 2 (i.e. $n = 2^i$, for some positive integer $i > 0$).

```

4-STUPIDSORT( $A[0..n-1]$ ) :
  if  $n \leq 8$ 
    INSERTIONSORT( $A[0..n-1]$ )
  else /*  $n > 8$  */
    for  $i \leftarrow 0$  to 2
      for  $j \leftarrow 2$  to  $i$ 
        4-STUPIDSORT( $A[jn/4..(j+2)n/4-1]$ )

```

1. Prove that 4-STUPIDSORT actually sorts its input.
2. State a recurrence (including the base case(s)) for the number of comparisons executed by 4-STUPIDSORT.
3. Solve the recurrence, and prove that your solution is correct. Does the algorithm deserve its name?
4. Show that the number of *swaps* executed by 4-STUPIDSORT is at most $\binom{n}{2}$.

(HARD)

- 2** The following randomized algorithm selects the r th smallest element in an unsorted array $A[1..n]$. For example, to find the smallest element, you would call $\text{RANDOMSELECT}(A, 1)$; to find the median element, you would call $\text{RANDOMSELECT}(A, \lfloor n/2 \rfloor)$. Recall from lecture that PARTITION splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n - 1$ comparisons altogether, and returns the new index of the pivot element.

```

RANDOMSELECT( $A[1..n], r$ ) :
   $p \leftarrow \text{Random}(1, n)$ 
   $k \leftarrow \text{Partition}(A[1..n], p)$ 
  if  $r < k$ 
    return RANDOMSELECT( $A[1..k-1], r$ )
  else if  $r > k$ 
    return RANDOMSELECT( $A[k+1..n], r-k$ )
  else
    return  $A[k]$ 

```

1. State a recurrence for the expected running time of RANDOMSELECT, as a function of n and r .
2. What is the *exact* probability that RANDOMSELECT compares the i -th smallest and j -th smallest elements in the input array? The correct answer is a simple function of i , j , and r . [Hint: Check your answer by trying a few small examples.]
3. Show that for any n and r , the expected running time of RANDOMSELECT is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b). For extra credit, find the exact expected number of comparisons, as a function of n and r .
4. What is the expected number of times that RANDOMSELECT calls itself recursively?

3 Solve the following randomization problems:

1. Given a function RANDBIT() that returns 0 or 1 with equal probability, write a function RAND(n) that returns a random integer between 0 and $n-1$ (inclusive) with uniform distribution over those possible values.
2. Given a function RANDREAL() that returns a random real number between 0 and 1, describe a procedure that generates a random permutation of the integer set $\{1, \dots, n\}$ where every possible permutation has the same probability of being returned. You can assume that RANDREAL() never returns the same number twice.
3. Given a function RAND(n) as describe above, describe a linear-time algorithm that outputs a permutation of the integer set $\{1, \dots, n\}$ where all permutations have equal probability. You can assume that RAND(n) takes constant time.

4 [This problem is required only for graduate students. Undergraduates can submit a solution for extra credit.]

You are watching a stream of packets go by one at a time, and want to take a random sample of k distinct packets from the stream. You face several problems:

- You only have room to save k packets at any one time.
- You do not know the total number of packets in the stream.
- If you choose not to save a packet as it goes by, it is gone forever.

In *each* of the three scenarios below, devise a scheme so that whenever the packet stream terminates you are left holding a subset of k packets chosen uniformly at random from the entire packet stream. If the total number of packets in the stream is less than k , you should hold all of these packets. (hint: To verify your solution, imagine that you now repeat the same experiment with the same stream sent in the *reverse* order. The probability to get the same output in the two experiments should be the same.)

1. Prior to watching the stream you know that the total number of packets is some number n . Also, you have room to save all n packets (i.e. $k = n$).
2. In this scenario you the know the values of both n and k in advance.

(HARD)

3. Here you still have room to hold k packets. However, you have no idea how many packets will flow through in the stream (i.e., n is unknown in advance).

5 Recall from lecture the notion of an *edit distance* between two words (Lecture 2, Section 2.7). There we derived an algorithm EDITDISTANCE to compute the minimum number of legal transformations required to convert one string into another (i.e. the edit distance). Let us now consider the problem assuming we have some additional information. Namely, suppose we are given a parameter k that is an upper limit on the edit distance of the two string parameters (k is some positive finite integer). Write a new procedure K-EDITDISTANCE to achieve the same results as EDITDISTANCE, this time taking advantage of the additional k parameter. What are the new and improved space and time complexities?

6 Finding maximal longest monotonically increasing subsequence

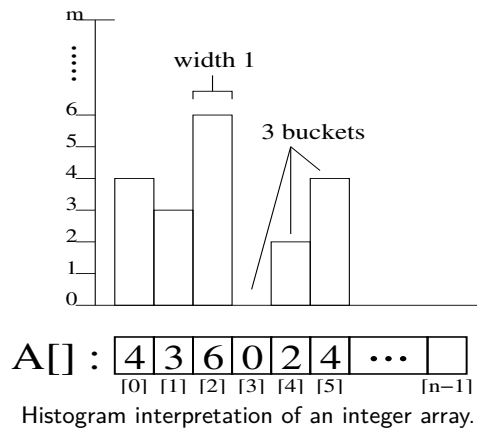
1. Give an $O(n^2)$ -time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. Recall that a monotonically increasing sequence is a sequence s_1, \dots, s_n where $i \leq j \Rightarrow s_i \leq s_j$ for all i and j in $\{1, \dots, n\}$.

2. [This problem is required only for graduate students. Undergraduates can submit a solution for extra credit.]

Give an $O(n \lg n)$ -time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (*Hint:* Observe that the last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i - 1$. Maintain candidate subsequences by linking them through the input sequence.)

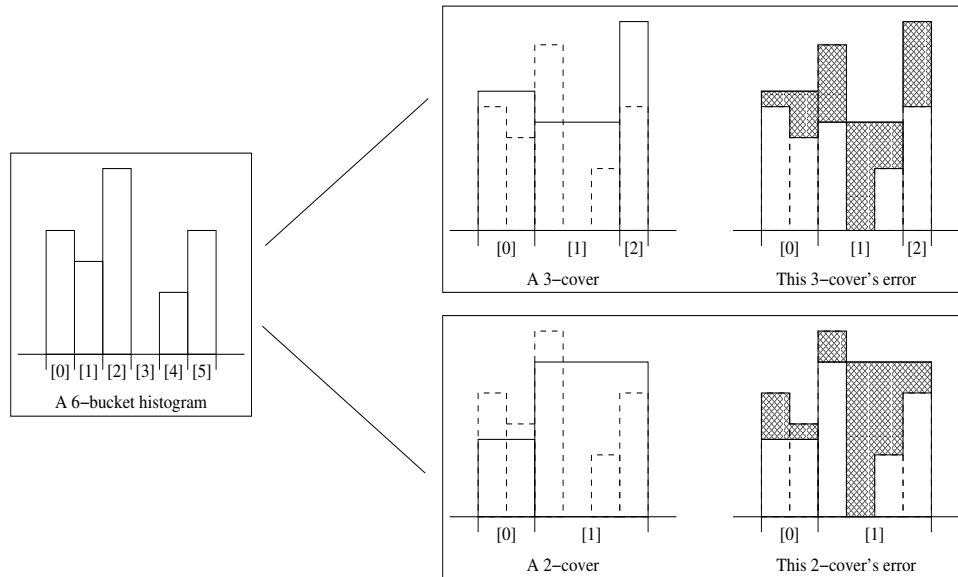
7 [This problem is required only for graduate students. Undergraduates can submit a solution for extra credit.]

Let an integer array $A[0..n - 1]$ represent a histogram with n buckets, each of width 1 (see the figure below). Informally, a *bucket* is a single-valued bar in the histogram while *width* is measured by the number of array cells beneath an object (i.e. a single bucket can have width greater than one). For clarity, assume throughout this entire problem that $n \geq 1$ and that each of the n array entries are integers in some predetermined finite range $[0..M]$.



A k -cover of a histogram is an approximation in the form of a new histogram of the same total width, but with k buckets (possibly of k different widths) instead of the n buckets in the original histogram (see the figures below). Each of the new k buckets have a width which is a non-negative integer number.^① The height of a new bucket is an integer number in the range $[0..M]$.

^①A new bucket must cover all or none of any given original bucket (i.e. cannot split an original bucket between two or more new buckets).



A k -cover and its associated error.

The error of a k -cover is the area of the geometric difference between the original histogram and its cover (i.e. the total area between the original histogram graph and the covering histogram graph). The illustrations above should clarify this quantity. A *best* k -cover of a given histogram is a k -cover whose error is no larger than any other k -cover of that same histogram (for the same value of k). The covers shown above are definitely not optimal.

1. How many bits are required to store a histogram?
2. How many bits are required to store a k -cover of a histogram?
3. Write an algorithm $1\text{-COVER}(A[l..r])$ that determines the best 1-cover of the histogram $A[l..r]$ and returns the value (height) of that histogram (*NOTE*: we are only covering A from index l through index r). You can assume that $0 \leq l \leq r < n$.
4. Using 1-COVER , write an algorithm $K\text{-COVER}(A[0..n-1], k, W[0..k-1], H[0, \dots, k-1])$ that determines the best k -cover of $A[0..n-1]$ and assigns the corresponding bucket widths of the k -cover into the array $W[0..k-1]$ and their heights into $H[0, \dots, k-1]$.
5. What are the space and running time complexities of $K\text{-COVER}$?
(Really HARD)
6. **Practice problem - no credit:** Describe an algorithm that receives as an input an histogram and constructs a data-structure in $O(n \log n)$ time, so that given query interval $[l..r]$, it computes $1\text{-COVER}(A[l..r])$ in $O(\log M \log^2 n)$ time. (It is possible to do even better than that.)

Practice Problems

8 Suppose you are a simple shopkeeper living in a country with n different types of coins, with values $1 = c[1] < c[2] < \dots < c[n]$. (In the U.S., for example, $n = 6$ and the values are 1, 5, 10, 25, 50 and 100 cents.) Your beloved and belevolent dictator, El Generalissimo, has decreed that whenever you give a customer change, you must use the smallest possible number of coins, so as not to wear out the image of El Generalissimo lovingly engraved on each coin by servants of the Royal Treasury.

1. In the United States, there is a simple greedy algorithm that always results in the smallest number of coins: subtract the largest coin and recursively give change for the remainder. El Generalissimo

does not approve of American capitalist greed. Show that there is a set of coin values for which the greedy algorithm does *not* always give the smallest possible of coins.

- Describe and analyze a dynamic programming algorithm to determine, given a target amount A and a sorted array $c[1..n]$ of coin values, the smallest number of coins needed to make A cents in change. You can assume that $c[1] = 1$, so that it is possible to make change for any amount A .

9 What excitement! The Champaign Spinners and the Urbana Dreamweavers have advanced to meet each other in the World Series of Basketweaving! The World Champions will be decided by a best-of- $2n - 1$ series of head-to-head weaving matches, and the first to win n matches will take home the coveted Golden Basket (for example, a best-of-7 series requiring four match wins, but we will keep the generalized case). We know that for any given match there is a constant probability p that Champaign will win, and a subsequent probability $q = 1 - p$ that Urbana will win.

Let $P(i, j)$ be the probability that Champaign will win the series given that they still need i more victories, whereas Urbana needs j more victories for the championship. $P(0, j) = 1$, $1 \leq j \leq n$, because Champaign needs no more victories to win. $P(i, 0) = 0$, $1 \leq i \leq n$, as Champaign cannot possibly win if Urbana already has. $P(0, 0)$ is meaningless. Champaign wins any particular match with probability p and loses with probability q , so

$$P(i, j) = p \cdot P(i - 1, j) + q \cdot P(i, j - 1)$$

for any $i \geq 1$ and $j \geq 1$.

Create and analyze an $O(n^2)$ -time dynamic programming algorithm that takes the parameters n , p and q and returns the probability that Champaign will win the series (that is, calculate $P(n, n)$).

10 The traditional Devonian/Cornish drinking song “The Barley Mow” has the following pseudolyrics^②, where $\text{container}[i]$ is the name of a container that holds 2^i ounces of beer.^③

```

BarleyMow(n):
    “Here’s a health to the barley-mow, my brave boys,”
    “Here’s a health to the barley-mow!”

    “We’ll drink it out of the jolly brown bowl,”
    “Here’s a health to the barley-mow!”
    “Here’s a health to the barley-mow, my brave boys,”
    “Here’s a health to the barley-mow!”

    for i ← 1 to n
        “We’ll drink it out of the container[i], boys,”
        “Here’s a health to the barley-mow!”
        for j ← i downto 1
            “The textcontainer[j],”
            “And the jolly brown bowl!”
            “Here’s a health to the barley-mow!”
            “Here’s a health to the barley-mow, my brave boys,”
            “Here’s a health to the barley-mow!”

```

- Suppose each container name $\text{textcontainer}[i]$ is a single word, and you can sing four words a second. How long would it take you to sing $\text{BarleyMow}(n)$? (Give a tight asymptotic bound.)

^②Pseudolyrics are to lyrics as pseudocode is to code.

^③One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.

- If you want to sing this song for $n > 20$, you'll have to make up your own container names, and to avoid repetition, these names will get progressively longer as n increases^④. Suppose $\text{textcontainer}[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing $\text{BarleyMow}(n)$? (Give a tight asymptotic bound.)
- Suppose each time you mention the name of a container, you drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each $\text{textcontainer}[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang $\text{BarleyMow}(n)$? (Give an *exact* answer, not just an asymptotic bound.)

11 Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the i th word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of whitespace between any two words on the same line.

Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line (not counting the one pixel required between every adjacent pair of words). Specifically, if a line contains words i through j , then the amount of extra white space on that line is $P - j + i - \sum_{k=i}^j p_k$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

12 You are at a political convention with n delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party a delegate belongs to. However, you can check whether any two delegates are in the *same* party or not by introducing them to each other. (Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.)

- Suppose a majority (more than half) of the delegates are from the same political party. Give an efficient algorithm that identifies a member of the majority party.
- Suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Present a practical procedure to pick a person from the plurality party as parsimoniously as possible. (Please.)

13 Give an algorithm that finds the *second* smallest of n elements in at most $n + \lceil \lg n \rceil - 2$ comparisons. [Hint: divide and conquer to find the smallest; where is the second smallest?]

14 A company is planning a party for its employees. The employees in the company are organized into a strict hierarchy, that is, a tree with the company president at the root. The organizers of the party have assigned a real number to each employee measuring how 'fun' the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the 'fun' ratings of the guests.

15 Suppose you have a subroutine that can find the median of a set of n items (i.e., the $\lfloor n/2 \rfloor$ smallest) in $O(n)$ time. Give an algorithm to find the k th biggest element (for arbitrary k) in $O(n)$ time.

16 You're walking along the beach and you stub your toe on something in the sand. You dig around it and find that it is a treasure chest full of gold bricks of different (integral) weight. Your knapsack can only

^④"We'll drink it out of the hemisemidemiyottapint, boys!"

carry up to weight n before it breaks apart. You want to put as much in it as possible without going over, but you *cannot* break the gold bricks up.

1. Suppose that the gold bricks have the weights $1, 2, 4, 8, \dots, 2^k$, $k \geq 1$. Describe and prove correct a greedy algorithm that fills the knapsack as much as possible without going over.
2. Give a set of 3 weight values for which the greedy algorithm does *not* yield an optimal solution and show why.
3. Give a dynamic programming algorithm that yields an optimal solution for an arbitrary set of gold brick values.

47.1.3. Homework 2

CS 373: Combinatorial Algorithms, Fall 2001

Homework 2 (due Tue. October 2, 2001 at 23:59)

Required Problems

- 1** [20 points] Let's analyze the number of random bits needed to implement the operations of a treap. Suppose we pick a priority p_i at random from the unit interval. Then the binary representation of each p_i can be generated as a potentially infinite series of bits that are the outcome of unbiased coin flips. The idea is to generate only as many bits in this sequence as is necessary for resolving comparisons between different priorities. Suppose we have only generated some prefixes of the binary representations of the priorities of the elements in the treap T . Now, while inserting an item y , we compare its priority p_y to other's priorities to determine how y should be rotated. While comparing p_y to some p_i , if their current partial binary representation can resolve the comparison, then we are done. Otherwise, they have the same partial binary representations (up to the length of the shorter of the two) and we keep generating more bits for each until they first differ.
- 1.A.** Compute a tight upper bound on the expected number of coin flips or random bits needed for a single priority comparison. (Note that during insertion every time we decide whether or not to perform a rotation, we perform a priority comparison. We are interested in the number of bits generated in such a single comparison.)
 - 1.B.** Generating bits one at a time like this is probably a bad idea in practice. Give a more practical scheme that generates the priorities in advance, using a small number of random bits, given an upper bound n on the treap size. Describe a scheme that works correctly with probability $\geq 1 - n^{-c}$, where c is a prespecified constant.
- 2** [20 points] Consider a uniform rooted tree of height h (every leaf is at distance h from the root). The root, as well as any internal node, has 3 children. Each leaf has a boolean value associated with it. Each internal node returns the value returned by the majority of its children. The evaluation problem consists of determining the value of the root; at each step, an algorithm can choose one leaf whose value it wishes to
- 2.A.** (5 PTS.) Describe a deterministic algorithm that runs in $O(n)$ time, that computes the value of the tree, where $n = 3^h$.

2.B. (10 PTS.) Consider the recursive randomized algorithm that evaluates two subtrees of the root chosen at random. If the values returned disagree, it proceeds to evaluate the third sub-tree. Show the expected number of leaves read by the algorithm on any instance is at most $n^{0.9}$.

(HARD)

2.C. (5 PTS.) Show that for any deterministic algorithm, there is an instance (a set of boolean values for the leaves) that forces it to read all $n = 3^h$ leaves. (hint: Consider an adversary argument, where you provide the algorithm with the minimal amount of information as it request bits from you. In particular, one can devise such an adversary algorithm.).

3 [20 points] Assume that you are given a data-structure DS (i.e., a black box) so that given n points p_1, \dots, p_n in \mathbb{R}^d one can build it in $T(n)$ time a data-structure that supports nearest-neighbor search in $Q(n)$ time. Namely, given a query point $q \in \mathbb{R}^d$, one can compute the point p_i which is nearest to q among all the points p_1, \dots, p_n . Formally, $dist(p_i, q) \leq dist(p_j, q)$ for $j = 1, \dots, n$.

Furthermore, assume that one can delete a point from DS in $D(n)$ time (namely, this point would no longer be considered in answering nearest-neighbor queries). Deleting a point that does not exist in the data-structure is allowed, and does not do anything.

Describe how to construct a data-structure that support insertions (i.e., you are allowed to use the black-box described above as a building block in the new data-structure). The new data-structure should have the following performance:

- Build an empty data-structure in $O(1)$ time.
- Insertion takes $O((T(n)/n) \log n)$ amortized time.
- Deletion takes $O(D(n) \log n)$ time.
- Nearest neighbor query takes $O(Q(n) \log n)$ time.

Here n is the overall number of insertions/deletions performed. (Hint: Maintain several data-structures DS_1, \dots, DS_k and simulate insertions by performing rebuilds.). Show that if $T(n) = n^2$ then the amortized insertion time is in fact $O(T(n)/n)$.

4 [20 points] You are given two sorted arrays of real numbers $A[1..n], B[1..n]$ (say, sorted in increasing order). Consider the set $C = \{C_{ij}\}$ of n^2 numbers that can be represented as $C_{ij} = A[i] + B[j]$ (to make things simple, assume that all those n^2 numbers are different), for $i = 1, \dots, n$ and $j = 1, \dots, n$.

4.A. (3 point) Describe an $O(n^2)$ time algorithm that receives k , and return the k th smallest element in C .

4.B. (15 points) Describe an $O(n \log^2 n)$ expected time algorithm that receives k , and return the k th smallest element in C . (Hint: consider C to be written as an implicit matrix of size $n \times n$, and observe that this matrix has a lot of useful properties).

4.C. (2 points) Describe an $O(n \log n)$ expected time algorithm that receives k , and return the k th smallest element in C .

5 [10 points]

5.A. Given a binary search tree T with n nodes with values stored in each node, describe an algorithm that prints all the values of T from smallest to largest in $O(n)$ time.

5.B. Describe such an algorithm that uses only $O(1)$ space, assuming that each node in T has a pointer to its parent.

5.C. Describe an algorithm that receives a tree T and outputs a balanced binary tree T' with the same values stored in it. The algorithm should work in linear time.

6 [10 points] Show how to implement a queue with two ordinary stacks so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$

7 [10 points] [This problem is required only for graduate students. No extra credit would be given for undergraduates submitting a solution for this question] Design a data structure to support the following two operations for a set S of integers:

INSERT(S, x) inserts x into S .

DELETE-LARGER-HALF(S) deletes the largest $\lceil S/2 \rceil$

Explain how to implement this data structure so that any sequence of m operations runs in $O(m)$ time.

8 [10 points] [This problem is required only for graduate students. No extra credit would be given for undergraduates submitting a solution for this question]

You have to provide a data-structure that support the following operations:

1. Create an empty set.
2. Insert an integer number into a set.
3. Delete an integer number from a set.
4. Search - given an integer number decide if the number is inside the set.
5. Merge two sets into a new set (the two old sets are destroyed, and can not be used any more).

Describe how to implement such a data-structure so that the price of each operation is $O(\log n)$ amortized. Namely, after performing a sequence of n operations, the overall running time is $O(n \log n)$. (Remember to prove everything - bounds, correctness, etc.)

9 [10 points] [This problem is required only for graduate students. No extra credit would be given for undergraduates submitting a solution for this question] Consider an ordinary binary search tree augmented by adding to each node x the field $size[x]$ giving the number of keys stored in the subtree rooted at x . Let α be a constant in the range $1/2 < \alpha < 1$. We say that a given node x is α -balanced if

$$size[left[x]] \leq \alpha * size[x]$$

and

$$size[right[x]] \leq \alpha * size[x]$$

The tree as a whole is α -balanced if every node in the tree is α -balanced.

Suppose that INSERT and DELETE are implemented as usual for an n -node binary search tree, except that after every such operation, if any node in the tree is no longer α -balanced, then the subtree rooted at the highest such node in the tree is rebuilt so that it becomes 1/2-balanced. i.e., as balanced as it can be.

We shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T , we define

$$\Delta(x) = |size[left[x]] - size[right[x]]|$$

and we define the potential of T as

$$\Phi(T) = c * \sum \Delta(x) \text{ for all } x \text{ in } T \text{ such that } \Delta(x) \geq 2$$

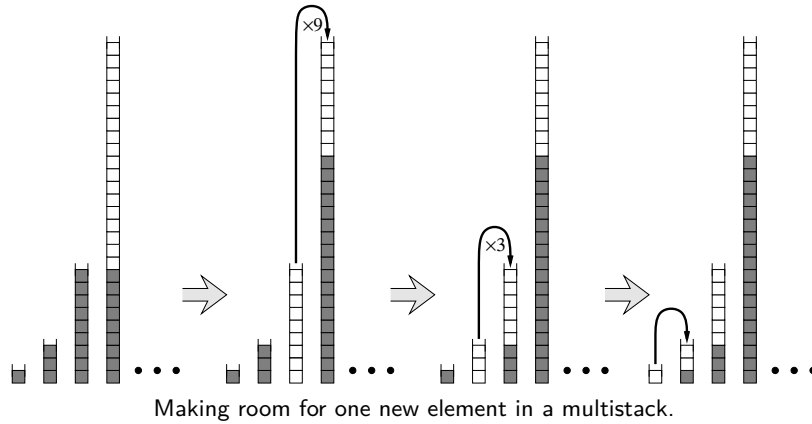
where c is a sufficiently large constant that depends on α

9.A. Argue that any binary search tree has non-negative potential and that a 1/2-balanced tree has potential 0.

- 9.B. Suppose that m units of potential can pay for rebuilding an m -node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α -balanced?
- 9.C. Show that inserting a node into or deleting a node from an n -node α -balanced tree costs $O(\log n)$ amortized time

Practice Problems

- 10 Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k . Describe an algorithm to find the k th smallest element in the union of A and B . (For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, our algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicates. For full credit, your algorithm should run in $\Theta(\log n)$ time. [Hint: First try to solve the special case $k = n$.]
- 11 Say that a binary search tree is *augmented* if every node v also stores $|v|$, the size of its subtree.
- 11.A. Show that a rotation in an augmented binary tree can be performed in constant time.
- 11.B. Describe an algorithm $\text{SCAPEGOATSELECT}(k)$ that selects the k th smallest item in an augmented scapegoat tree in $O(\log n)$ *worst-case* time.
- 11.C. Describe an algorithm $\text{SPRAYSELECT}(k)$ that selects the k th smallest item in an augmented splay tree in $O(\log n)$ *amortized* time.
- 11.D. Describe an algorithm $\text{TREAPSELECT}(k)$ that selects the k th smallest item in an augmented treap in $O(\log n)$ *expected* time.
- 12 Do the following:
- 12.A. Prove that only one subtree gets rebalanced in a scapegoat tree insertion.
- 12.B. Prove that $I(v) = 0$ in every node of a perfectly balanced tree. (Recall that $I(v) = \max\{0, |T| - |s| - 1\}$, where T is the child of greater height and s the child of lesser height, and $|v|$ is the number of nodes in subtree v . A perfectly balanced tree has two perfectly balanced subtrees, each with as close to half the nodes as possible.)
(HARD)
- 12.C. Show that you can rebuild a fully balanced binary tree from an unbalanced tree in $O(n)$ time using only $O(\log n)$ additional memory.
- 13 Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:
- After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
 - After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.
- Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do *not* use the potential method (it makes it much more difficult).
- 14 A *multistack* consists of an infinite series of stacks S_0, S_1, S_2, \dots , where the i th stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i , we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2} , and so on. Moving a single element from one stack to the next takes $O(1)$ time.

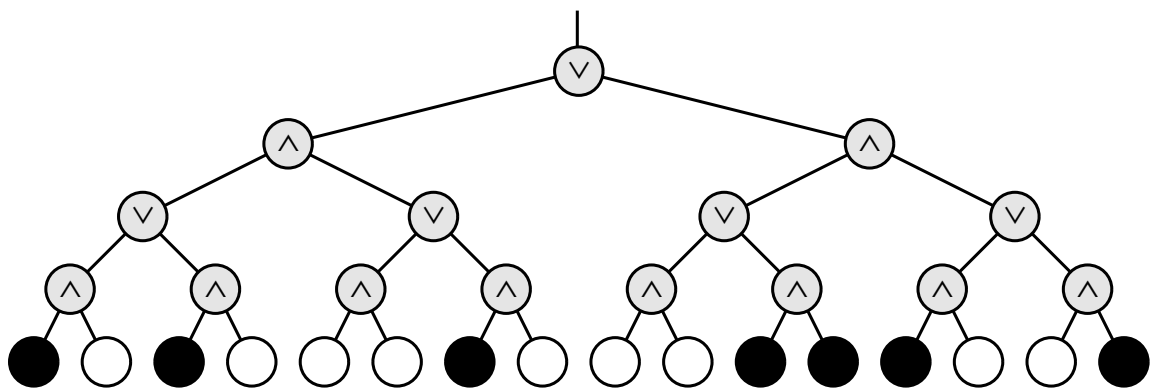


1. [1 point] In the worst case, how long does it take to push one more element onto a multistack containing n elements?
2. [9 points] Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack. You can use any method you like.

15 Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.

You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are *and* gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

- 15.A. (2 PTS.) Describe and analyze a deterministic algorithm to determine whether or not you can win. [Hint: This is easy!]
- 15.B. (8 PTS.) Unfortunately, Death won't let you even look at every node in the tree. Describe a randomized algorithm that determines whether you can win in $\Theta(3^n)$ expected time. [Hint: Consider the case $n = 1$.]



- 16 1. Show that it is possible to transform any n -node binary search tree into any other n -node binary search tree using at most $2n - 2$ rotations. (HARD)

- Use fewer than $2n - 2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n - 6$ rotations, and there are pairs of trees that are $2n - 10$ rotations apart. These are the best bounds known.

17 Faster Longest Increasing Subsequence(LIS)

Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. [Hint: In the dynamic programming solution, you don't really have to look back at all previous items. There was a practice problem on HW 1 that asked for an $O(n^2)$ algorithm for this. If you are having difficulty, look at the solution provided in the HW 1 solutions.]

18 Amortization

- Modify the binary double-counter (see class notes Sept 12) to support a new operation SIGN, which determines whether the number being stored is positive, negative, or zero, in constant time. The amortized time to increment or decrement the counter should still be a constant.

[Hint: Suppose p is the number of significant bits in P , and n is the number of significant bits in N . For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update p and n in $O(1)$ time.]

(HARD)

- Do the same but now you can't assume that p and n can be updated in $O(1)$ time.

(HARD)

19 Amortization

Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of 'fits', where the i th least significant fit indicates whether the sum includes the i th Fibonacci number F_i . For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string. This is *not* the same representation as on Homework 0.]

20 Detecting overlap

- You are given a list of ranges represented by min and max (e.g., [1,3], [4,5], [4,9], [6,8], [7,10]). Give an $O(n \log n)$ -time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.
- You are given a list of rectangles represented by min and max x - and y -coordinates. Give an $O(n \log n)$ -time algorithm that decides whether or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). [Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.]

21 Comparison of Amortized Analysis Methods

A sequence of n operations is performed on a data structure. The i th operation costs i if i is an exact power of 2, and 1 otherwise. That is operation i costs $f(i)$, where:

$$f(i) = \begin{cases} i, & i = 2^k, \\ 1, & \text{otherwise} \end{cases}$$

Determine the amortized cost per operation using the following methods of analysis:

- Aggregate method

2. Accounting method (**HARD**)
 3. Potential method
-

47.1.4. Homework 3

CS 373: Combinatorial Algorithms, Fall 2001

Homework 3 (due Thursday, October 18, 2001 at 11:59.99 p.m.)

Required Problems

1 *Treaps revisited* (10 PTS.)

Modify a treap data-structure T so that it supports the following operations quickly. Implementing some of those operations would require you to modify the information stored in each node of the treap, and how insertions/deletions are being handled in the treap. For each of the following, describe separately the changes made in detail, and the algorithms for answering those queries. (Note, that under the modified version of the treap, insertion and deletion should still take $O(\log n)$ expected time.)

1. (2 PTS.) Find the smallest element stored in T in $O(\log n)$ expected time.
2. (2 PTS.) Find the largest element stored in T in $O(\log n)$ expected time.
3. (2 PTS.) Given a query k , find the k -th smallest element stored in T in $O(\log n)$ expected time.
4. (2 PTS.) Given a query $[a, b]$, find the number of elements stored in T with their values being in the range $[a, b]$, in $O(\log n)$ expected time.
5. (2 PTS.) Given a query $[a, b]$, report (i.e., printout) all the elements stored in T in the range $[a, b]$, in $O(\log(n) + u)$ expected time, where u is the number of elements printed out.

2 *The Pub Inspector Problem* (10 PTS.)

Jane is a Champaign pub inspector, and she have to visit all the pubs in Champaign starting from her home and then coming back to her home. She knows the exact distances between any two pubs, and also from any pub to her home. Help her find such a cycle that is no longer than twice of the optimal shortest cycle.

Formally, you are given a undirected graph $G = (V, E)$, where any two vertices a and b have an edge ab connecting them (such a graph is called a clique), and for every edge e of G there is an associated non-negative length $d(e) \geq 0$. Furthermore, the triangle inequality holds for those distance; namely, for any $a, b, c \in V(G)$, we have $d(ab) + d(bc) \geq d(ac)$.

Show a polynomial time algorithm that computes a cycle which visits all the vertices (visiting every vertex exactly once) and has a length at most twice of the optimal shortest cycle that visits all the vertices.

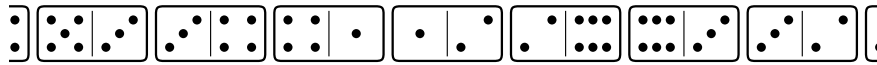
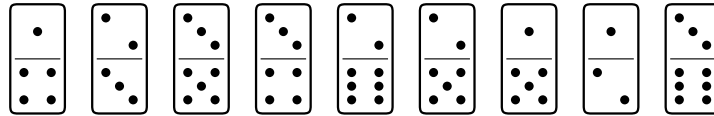
Hint: Use the minimal spanning tree and the fact that the triangle inequality holds for the distances in the graph.

3 *The domino effect* (20 PTS.)

The graph $G = (V, E)$ is a connected undirected graph with all vertices having even degrees.

1. (5 PTS.) Show a linear time algorithm to find a set of cycles $U = \{c : c \text{ is a cycle of } G\}$, so that all the cycles are disjoint and their union cover all the edges of G . A cycle might use an edge at most once (however, a vertex might be used several time in the same cycle). Namely, if E_c is the set of edges in the cycle c , then then $\bigcup_{c \in U} E_c = E$, and $E_c \cap E_d = \emptyset$ for $c, d \in U, c \neq d$.
2. (5 PTS.) Show a linear time algorithm that computes a cycle c covering all the edges of G . Namely, $E_c = E$.
3. (5 PTS.) If there are exactly two vertices which have odd degree, and all the other vertices have even degree, show a linear algorithm to find a single path π that covers all the edges in the graph. Namely, $E_\pi = E$.
4. (5 PTS.) A domino is a 2×1 rectangle divided into two squares, with a certain number of pips (dots) in each square. In most domino games, the players lay down dominos at either end of a single chain. Adjacent dominos in the chain must have matching numbers. (See the figure below.)

Describe and analyze an efficient algorithm to determine whether a given set of n dominos can be lined up in a single chain. For example, for the set of dominos shown below, the correct output is TRUE.



Top: A set of nine dominos

Bottom: The entire set lined up in a single chain

4 *Going home* (20 PTS.)

Every evening, Jane walks from her office to her home. There are several buildings (i.e., obstacles) between her home and the office. For simplicity, we assume that all the obstacles are triangles. Figure 2(a) is one of the routes Jane may take, and Figure 2(b) is seemingly a shorter route from Jane's home to her office. Describe an algorithm for computing the shortest path from Jane home to her office.

Formally, you are given a set of n disjoint triangles $\Delta_1, \dots, \Delta_n$ in the plane, a starting point s , and ending point t . Describe an algorithm that computes the Euclidean shortest path from s to t that avoids the interior of triangles. Your algorithm should be no slower than $O(n^3)$.

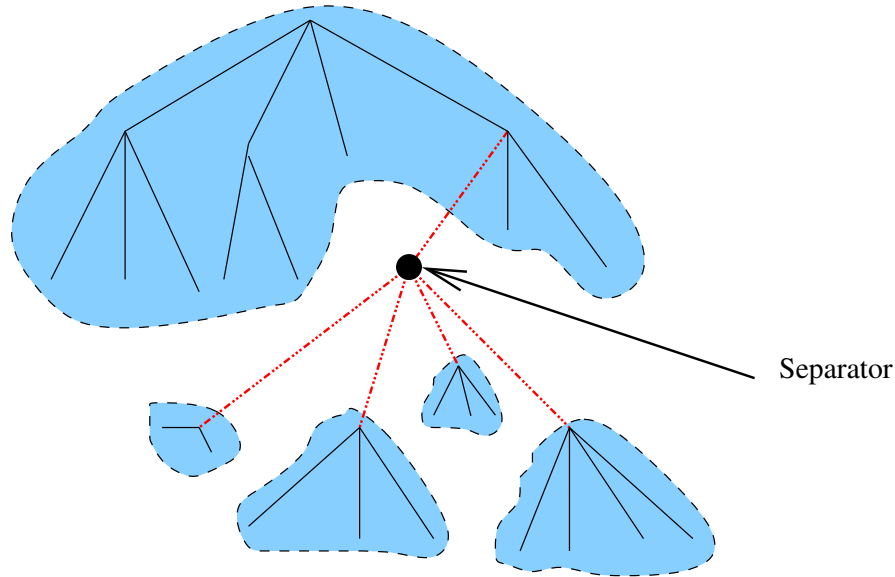


Figure 47.1: A tree T , a separator in T , and the forest of trees generated by the removal of the separator, and all the edges adjacent to it.

5 *Separators in trees* (20 PTS.)

$T = (V, E)$ is a tree, a node $v \in T$ is a separator of T , if v satisfies the following condition:

$$\forall f \in F \quad |f| \leq \frac{2}{3}n,$$

where $F = T \setminus \{v\}$ is the forest resulting from T by removing v and all the edges attached to v from T , $n = |T|$, and $|f|$ denotes the number of nodes in a tree f . Intuitively, a separator is a good vertex to use in breaking a tree into subtrees. Figure 47.1 shows a tree and its separator.

1. (4 PTS.) Prove that there exists a separator in any tree.
2. (8 PTS.) Show a linear time algorithm to find a separator of a tree. (Hint: It is sometimes useful to pick an arbitrary vertex of a tree, and designate it as a root, thus turning T into a rooted tree, and thus oriented.)
3. (8 PTS.) We assign a weight $w(e)$ to each edge e of the tree. Design a data structure so that the tree built using the data structure can answer the query "Which is the edge with the smallest weight on the unique path between the nodes a and b ?" in $O(\log n)$ time. The data-structure should use $O(n \log n)$ space and preprocessing.
(HARD)
4. (5 PTS.) [Extra credit question for graduate and undergraduates.] Show a data-structure to answer the query in $O(\log \log n)$ time, that uses $O(n \log \log n)$ space and preprocessing.
(Really HARD)
5. (5 PTS.) [Extra credit question for graduate and undergraduates.] Show that one can build a data-structure that answer the query in $O(1)$ time with $O(n \log n)$ space.

6 Slot-size bound for chaining (20 PTS.)

[This problem is required only for graduate students. Undergraduates can submit a solution for extra credit.]

Suppose that we have a hash table with n slots, with collisions resolved by chaining, and suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have been inserted. Your mission is to prove an $O(\lg n / \lg \lg n)$ upper bound on $E[M]$, the expected value of M .

1. (4 PTS.) Argue that the probability Q_k that exactly k keys hash to a particular slot is given by

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

2. (4 PTS.) Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Show that $P_k \leq nQ_k$.
3. (4 PTS.) Use Stirling's approximation^① to show that $Q_k < e^k / k^k$. Hint: Use the inequality $1 + x \leq e^x$, for $x \geq 0$.
4. (4 PTS.) Show that there exists a constant $c > 1$ such that $Q_{k_0} < 1/n^3$, for $k_0 = c \lg n / \lg \lg n$. Conclude that $P_k < 1/n^2$ for $k \geq k_0 = c \lg n / \lg \lg n$.
5. (4 PTS.) Argue that

$$E[m] \leq Pr \left\{ M > \frac{c \lg n}{\lg \lg n} \right\} * n + Pr \left\{ M \leq \frac{c \lg n}{\lg \lg n} \right\} * \frac{c \lg n}{\lg \lg n}$$

Conclude that $E[M] = O(\lg n / \lg \lg n)$

7 Slightly Faster MST (10 PTS.)

[This problem is required only for graduate students. Undergraduates can submit a solution for extra credit.]

1. (4 PTS.) Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W ?
2. (3 PTS.) Suppose that the edge weights in a graph are uniformly distributed over the half-open interval $[0,1)$. Which algorithm, Kruskal's or Prim's, can you make run faster?
3. (3 PTS.) Suppose that a graph G has a minimum spanning tree already computed. How quickly can the minimum spanning tree be updated if a new vertex and incident edges are added to G ?

8 The Dominator (10 PTS.)

[This problem is required only for graduate students. Undergraduates can submit a solution for extra credit.]

Let $G = (V, E)$ be a *directed* graph represented by an adjacency list. So each node $G[i]$ has list of all nodes reachable in 1 step from i . Each node v of G also has a non-negative value associated with it $w(v)$. Given an $O(|E| + |V| \log |V|)$ time algorithm that computes, for every node, the highest value reachable from that node. A node w is reachable from v , if there is a directed path in G starting at v and ending in w .

For example, if there is a directed path in G from any vertex to any vertex (i.e., G is strongly-connected), then the value computed for all the nodes of G would be the same in the entire graph.

^①Stirling's approximation $n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$

Practice Problems

1 Hashing:

A hash table of size m is used to store n items with $n \leq m/2$. Open addressing is used for collision resolution.

1. Assuming uniform hashing, show that for $i = 1, 2, \dots, n$, the probability that the i^{th} insertion requires strictly more than k probes is at most 2^{-k} .
2. Show that for $i = 1, 2, \dots, n$, the probability that the i^{th} insertion requires more than $2 \lg n$ probes is at most $1/n^2$.

Let the random variable X_i denote the number of probes required by the i^{th} insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

- (c) Show that $\Pr\{X > 2 \lg n\} \leq 1/n$.
- (d) Show that the expected length of the longest probe sequence is $E[X] = O(\lg n)$.

2 Reliable Network:

Suppose you are given a graph of a computer network $G = (V, E)$ and a function $r(u, v)$ that gives a reliability value for every edge $(u, v) \in E$ such that $0 \leq r(u, v) \leq 1$. The reliability value gives the probability that the network connection corresponding to that edge will *not* fail. Describe and analyze an algorithm to find the most reliable path from a given source vertex s to a given target vertex t .

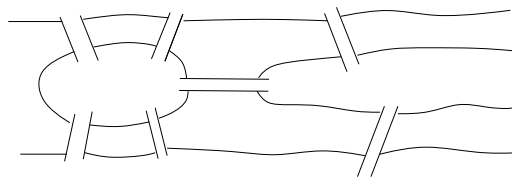
3 Aerophobia:

After graduating you find a job with Aerophobes-R'-Us, the leading traveling agency for aerophobic people. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying so the trip should be as short as possible.

In other words, a person wants to fly from city A to city B in the shortest possible time. S/he turns to the traveling agent who knows all the departure and arrival times of all the flights on the planet. Give an algorithm that will allow the agent to choose an optimal route to minimize the total time in transit. Hint: rather than modify Dijkstra's algorithm, modify the data. The total transit time is from departure to arrival at the destination, so it will include layover time (time waiting for a connecting flight).

4 The Seven Bridges of Königsberg:

During the eighteenth century the city of Königsberg in East Prussia was divided into four sections by the Pregel river. Seven bridges connected these regions, as shown below. It was said that residents spent their Sunday walks trying to find a way to walk about the city so as to cross each bridge exactly once and then return to their starting point.



1. Show how the residents of the city could accomplish such a walk or prove no such walk exists.
2. Given any undirected graph $G = (V, E)$, give an algorithm that finds a cycle in the graph that visits every edge exactly once, or says that it can't be done.

5 Minimum Spanning Tree changes:

Suppose you have a graph G and an MST of that graph (i.e. the MST has already been constructed).

1. Give an algorithm to update the MST when an edge is added to G .
2. Give an algorithm to update the MST when an edge is deleted from G .
3. Give an algorithm to update the MST when a vertex (and possibly edges to it) is added to G .

6 Nesting Envelopes

[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.] You are given an unlimited number of each of n different types of envelopes. The dimensions of envelope type i are $x_i \times y_i$. In nesting envelopes inside one another, you can place envelope A inside envelope B if and only if the dimensions A are *strictly smaller* than the dimensions of B . Design and analyze an algorithm to determine the largest number of envelopes that can be nested inside one another.

7 Makefiles:

In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called ‘make’ that only recompiles those files that were changed after the most recent compilation, *and* any intermediate files in the compilation that depend on those that were changed. A Makefile is typically composed of a list of source files that must be compiled. Each of these source files is dependent on some of the other files which are listed. Thus a source file must be recompiled if a file on which it depends is changed.

Assuming you have a list of which files have been recently changed, as well as a list for each source file of the files on which it depends, design an algorithm to recompile only those necessary. DO NOT worry about the details of parsing a Makefile.

(Really HARD)

8 Let the hash function for a table of size m be

$$h(x) = \lfloor Amx \rfloor \bmod m$$

where $A = \hat{\phi} = \frac{\sqrt{5}-1}{2}$. Show that this gives the best possible spread, i.e. if the x are hashed in order, $x + 1$ will be hashed in the largest remaining contiguous interval.

9 The incidence matrix of an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \begin{cases} 1 & (i, j) \in E, \\ 0 & (i, j) \notin E. \end{cases}$$

1. Describe what all the entries of the matrix product BB^T represent (B^T is the matrix transpose). Justify.
2. Describe what all the entries of the matrix product B^TB represent. Justify.

(Really HARD)

3. Let $C = BB^T - 2A$. Let C' be C with the first row and column removed. Show that $\det C'$ is the number of spanning trees. (A is the adjacency matrix of G , with zeroes on the diagonal).

10 $o(V^2)$ Adjacency Matrix Algorithms

1. Give an $O(V)$ algorithm to decide whether a directed graph contains a *sink* in an adjacency matrix representation. A sink is a vertex with in-degree $V - 1$.
2. An undirected graph is a scorpion if it has a vertex of degree 1 (the sting) connected to a vertex of degree two (the tail) connected to a vertex of degree $V - 2$ (the body) connected to the other $V - 3$ vertices (the feet). Some of the feet may be connected to other feet.

Design an algorithm that decides whether a given adjacency matrix represents a scorpion by examining only $O(V)$ of the entries.

3. Show that it is impossible to decide whether G has at least one edge in $O(V)$ time.

11 Shortest Cycle:
Given an **undirected** graph $G = (V, E)$, and a weight function $f : E \rightarrow \mathbf{R}$ on the **edges**, give an algorithm that finds (in time polynomial in V and E) a cycle of smallest weight in G .

12 Longest Simple Path:
Let graph $G = (V, E)$, $|V| = n$. A *simple path* of G , is a path that does not contain the same vertex twice. Use dynamic programming to design an algorithm (not polynomial time) to find a simple path of maximum length in G . Hint: It can be done in $O(n^c 2^n)$ time, for some constant c .

13 Minimum Spanning Tree:
Suppose all edge weights in a graph G are equal. Give an algorithm to compute an MST.

14 Transitive reduction:
Give an algorithm to construct a *transitive reduction* of a directed graph G , i.e. a graph G^{TR} with the fewest edges (but with the same vertices) such that there is a path from a to b in G iff there is also such a path in G^{TR} .

15 1. What is $5^{2^{29}5^0 + 23^41 + 17^32 + 11^23 + 5^14} \pmod{6}$?
2. What is the capital of Nebraska? Hint: It is not Omaha. It is named after a famous president of the United States that was not George Washington. The distance from the Earth to the Moon averages roughly 384,000 km.

16 k-universal hashing and authentication Let $H = \{h\}$ be a class of hash function in which each h maps the universe U of keys to $\{0, 1, \dots, m-1\}$. We say that H is **k-universal** if, for every fixed sequence of k distinct keys $\langle x^{(1)}, x^{(2)}, \dots, x^{(k)} \rangle$ and for any h chosen at random from H , the sequence $\langle h(x^{(1)}), h(x^{(2)}), \dots, h(x^{(k)}) \rangle$ is equally likely to be any of the m^k sequence of length k with elements drawn from $\{0, 1, \dots, m-1\}$.

1. Show that if H is 2-universal, then it is universal.
2. Let U be the set of n -tuples of values drawn from Z_p , and let $B = Z_p$, where p is prime. For any n -tuple $a = \langle a_0, a_1, \dots, a_{n-1} \rangle$ of values from Z_p and from any $b \in Z_p$, define the hash function $h_{a,b} : U \rightarrow B$ on an input n -tuple $x = \langle x_0, x_1, \dots, x_{n-1} \rangle$ from U as

$$h_{a,b}(x) = \left(\sum_{j=0}^{n-1} a_j b_j + b \right) \pmod{p}$$

and let $H = \{h_{a,b}\}$. Argue that H is 2-universal.

3. Suppose that Alice and Bob agree secretly on a hash function $h_{a,b}$ from a 2-universal family H of hash function. Later, Alice sends a message m to Bob over the Internet, where $m \in U$. She authenticates her message to Bob by also sending an authentication tag $t = h_{a,b}(m)$, and Bob checks that the pair (m, t) he receives satisfies $t = h_{a,b}(m)$. Suppose that an adversary intercepts (m, t) en route and tries to fool Bob by replacing the pair with a different pair (m', t') . Argue that the probability that the adversary succeeds in fooling Bob into accepting (m', t') is at most $1/p$, no matter how much computing power the adversary has.

47.1.5. Homework 4

CS 373: Combinatorial Algorithms, Fall 2001

Homework 4, due Tuesday, October 30, 2001, at 23:59:59

Required Problems

1 Range searching (10 PTS.)

In the two dimensional plane, a *vertical strip* is an infinitely long closed region enclosed between two vertical lines (a *closed* region contains its boundaries).

- (4 PTS.) Given n points in the plane ($p_1 \dots p_n$), design a data structure that when queried with a vertical strip (specified by two numbers l, r , which specify the strip $l \leq x \leq r$), output the number of points within the strip in $O(\log n)$ time. See [Figure 47.2](#) (i).

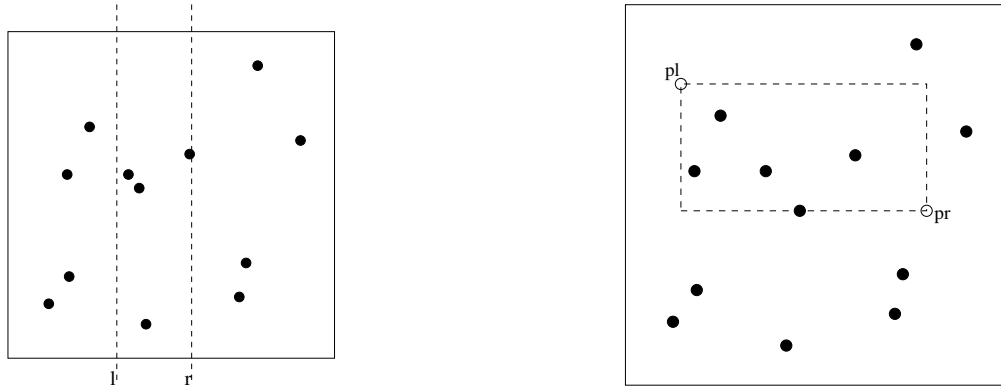
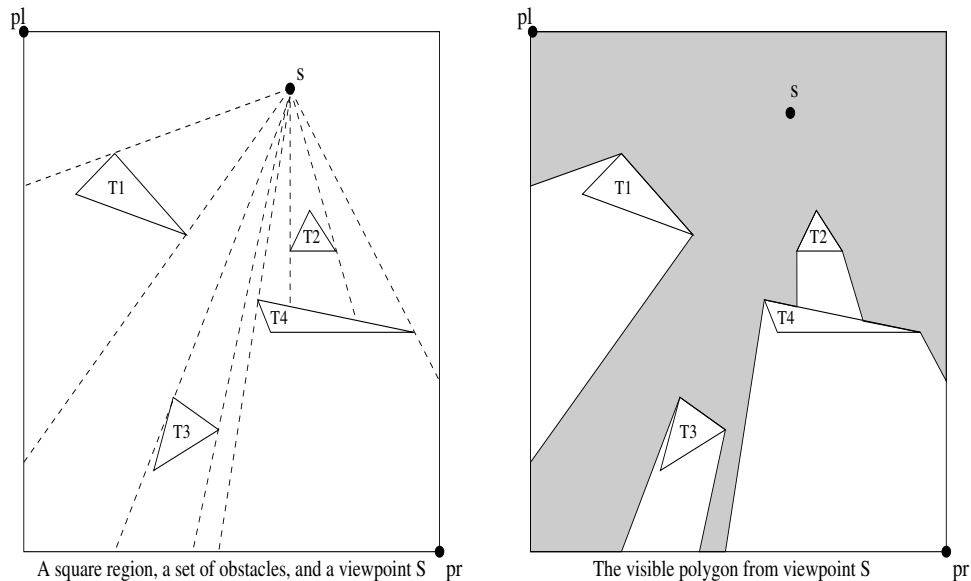


Figure 47.2: (i) A vertical strip containing four points, (ii) A rectangle containing five points

- (6 PTS.) Given n points in the plane ($p_1 \dots p_n$), design a data structure that when queried with an axis parallel rectangular region R (specified by an upper left point $p_l = (x_l, y_l)$ and a lower right point $p_r = (x_r, y_r)$), can output the number of points within the rectangle R in $O(\log^2 n)$ time (*hint*: consider the rectangular region to be the intersection of a vertical strip and a horizontal strip). See [Figure 47.2](#) (ii).

2 Visibility polygon (10 PTS.)

- (5 PTS.) Given a large square area (with upper left boundary point p_l and lower right boundary point p_r) that contains a set of n obstacles (triangles $T_1 \dots T_n$) and a location (i.e., a point s), compute the set of all visible points in the square when looking FROM s in every direction. This set will be a polygon P (with vertices p_1, \dots, p_k). The obstacles are opaque (i.e. one cannot see through them). The following figure illustrates the situation. Your algorithm should run in $O(n \log n)$ time (*hint*: perform a sweep while marking visible points).

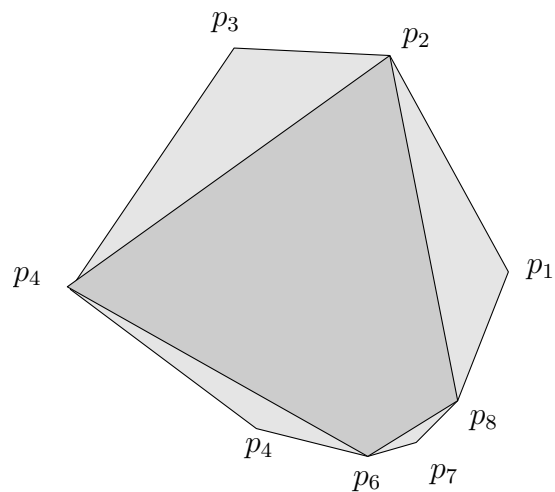


2. (5 PTS.) Give an $O(n^2 \log n)$ time version of the GOINGHOME algorithm in the last homework (Hw 3, problem 4).

3 Largest polygon (10 PTS.)

Given a convex polygon P in the plane with n vertices p_1, \dots, p_n , and a parameter k , describe an algorithm that computes the largest area polygon with k vertices from P . Namely, you have to compute $1 \leq i_1 < i_2 < i_3 < \dots < i_k \leq n$, such that the polygon $p_{i_1}p_{i_2} \dots p_{i_k}$ has maximum area. You can assume that you have a function $\Delta(a, b, c)$ that computes the area of the triangle a, b, c . The following figure provides an illustration.

Hint: First solve for the case that $i_1 = 1, i_2 = 2$, and then extend it to other cases.



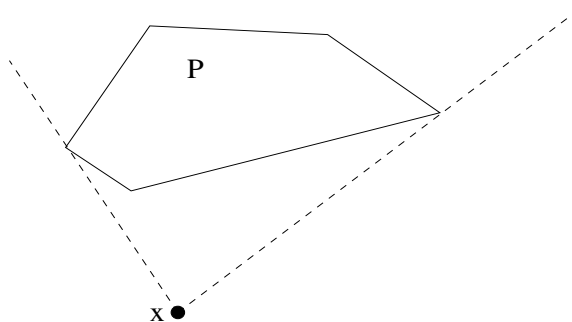
A convex polygon, and the largest quadrangle in it ($k = 4$). The algorithm should output p_2, p_4, p_6, p_8

(HARD)

4 Fast tangents (10 PTS.)

[This problem is required only for graduate students. Undergraduates can submit a solution for extra credit.]

Consider a *convex* polygon P and a point x that is outside P . The two tangents to P that pass through x are defined as the two lines that pass through x , touch the polygon at a vertex, and are located completely to one side of the polygon (except for the touched vertex). The following figure illustrates this.

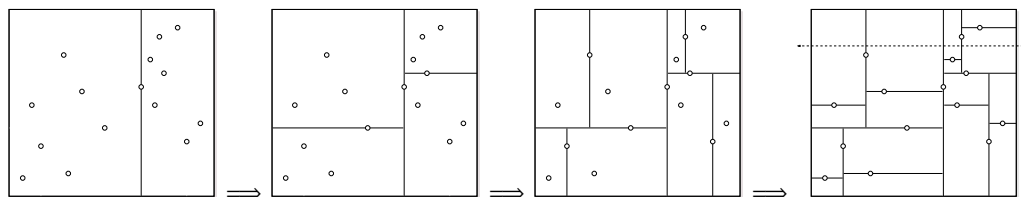


A polygon, an outside point, and two tangents

The problem is as follows. You are given a convex polygon P in the plane (with n vertices p_1, \dots, p_n). Design a data structure so that when given a query point x that is outside P it computes the two tangents to P that pass through x . You are allowed linear preprocessing time to build the data structure, after which, a query should be answered in logarithmic time. Hint: Use various forms of binary search several times - the details are *very* messy.

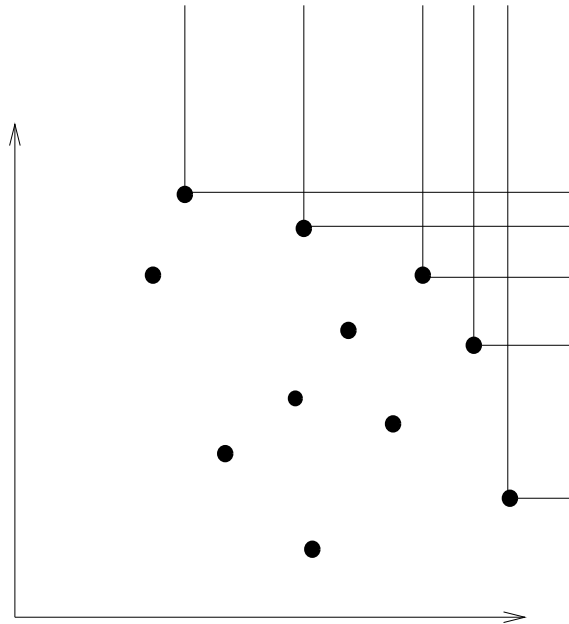
Practice Problems

- 1** Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the rectangle as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points *as evenly as possible* by passing through a median point inside the box (*not* on the boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.



Successive divisions of a kd-tree for 15 points. The dashed line crosses four cells.

- How many cells are there, as a function of n ? Prove your answer is correct.
- In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n ? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k .
- Suppose we have n points stored in a kd-tree. Describe an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) in $O(\sqrt{n})$ time.
(HARD)
- Find an algorithm that counts the number of points that lie inside a rectangle R and show that it takes $O(\sqrt{n})$ time. You may assume that the sides of the rectangle are parallel to the sides of the box.



An example staircase as in problem 3.

2 Circle Intersection

Describe an algorithm to decide, given n circles in the plane, whether any two of them intersect, in $O(n \log n)$ time. Each circle is specified by three numbers: its radius and the x - and y -coordinates of its center.

We only care about intersections between circle boundaries; concentric circles do not intersect. What general position assumptions does your algorithm require? [Hint: Modify an algorithm for detecting line segment intersections, but describe your modifications very carefully! There are at least two very different solutions.]

3 Staircases

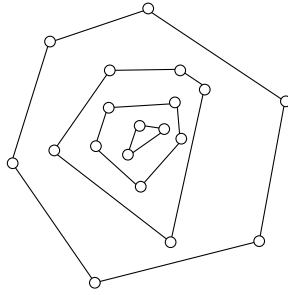
You are given a set of points in the first quadrant. A *left-up* point of this set is defined to be a point that has no points both greater than it in both coordinates. The left-up subset of a set of points then forms a *staircase* (see figure).

1. Prove that left-up points do not necessarily lie on the convex hull.
2. Give an $O(n \log n)$ algorithm to find the staircase of a set of points.
3. Assume that points are chosen uniformly at random within a rectangle. What is the average number of points in a staircase? Justify. Hint: you will be able to give an exact answer rather than just asymptotics. You have seen the same analysis before.

4 Convex Layers

Given a set Q of points in the plane, define the *convex layers* of Q inductively as follows: The first convex layer of Q is just the convex hull of Q . For all $i > 1$, the i th convex layer is the convex hull of Q after the vertices of the first $i - 1$ layers have been removed.

Give an $O(n^2)$ -time algorithm to find all convex layers of a given set of n points.



A set of points with four convex layers.

5 Solve the travelling salesman problem for points in convex position (ie, the vertices of a convex polygon). Finding the shortest cycle that visits every point is easy – it’s just the convex hull. Finding the shortest path that visits every point is a little harder, because the path can cross through the interior.

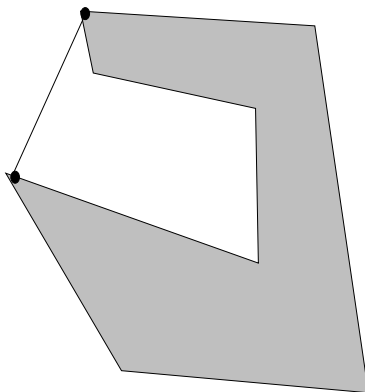
1. Show that the optimal path cannot be one that crosses itself.
2. Describe an $O(n^2)$ time dynamic programming algorithm to solve the problem.

6 Basic Computation (assume two dimensions and *exact* arithmetic)

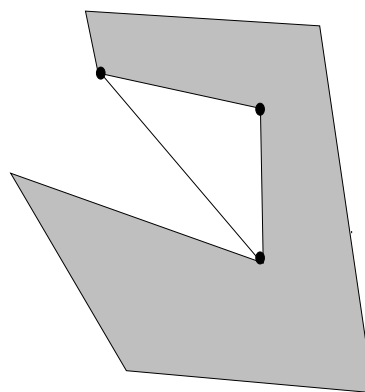
1. Intersection: Extend the basic algorithm to determine if two line segments intersect by taking care of *all* degenerate cases.
2. Simplicity: Give an $O(n \log n)$ algorithm to determine whether an n -vertex polygon is simple.
3. Area: Give an algorithm to compute the area of a simple n -polygon (not necessarily convex) in $O(n)$ time.
4. Inside: Give an algorithm to determine whether a point is within a simple n -polygon (not necessarily convex) in $O(n)$ time.

7 External Diagonals and Mouths

1. A pair of polygon vertices defines an *external diagonal* if the line segment between them is completely outside the polygon. Show that every nonconvex polygon has at least one external diagonal.
2. Three consecutive polygon vertices p, q, r form a *mouth* if p and r define an external diagonal. Show that every nonconvex polygon has at least one mouth.



An external diagonal



A mouth

8 On-Line Convex Hull

We are given the set of points one point at a time. After receiving each point, we must compute the convex hull of all those points so far. Give an algorithm to solve this problem in $O(n^2)$ (We could obviously use Graham's scan n times for an $O(n^2 \log n)$ algorithm). Hint: How do you maintain the convex hull?

9 Reachability

Given an undirected graph G with n vertices, and two vertices s and t , describe a *deterministic* algorithm that decides if s and t are in the same connected component (i.e., is there a path between s and t in G ?) of G using only $O(\log n)$ space. (The running time of this algorithm is quite bad, but we do not care.)

10 Another On-Line Convex Hull Algorithm

1. Given an n -polygon and a point outside the polygon, give an algorithm to find a tangent. (**HARD**)
2. Suppose you have found both tangents. Give an algorithm to remove the points from the polygon that are within the angle formed by the tangents (as segments!) and the opposite side of the polygon.
3. Use the above to give an algorithm to compute the convex hull on-line in $O(n \log n)$

11 Order of the size of the convex hull

The convex hull on $n \geq 3$ points can have anywhere from 3 to n points. The average case depends on the distribution.

1. Prove that if a set of points is chosen randomly within a given rectangle then the average size of the convex hull is $O(\log n)$.
(**Really HARD**)
2. Prove that if a set of points is chosen randomly within a given circle then the average size of the convex hull is $O(n^{1/3})$.

12 Ghostbusters and Ghosts

A group of n ghostbusters is battling n ghosts. Each ghostbuster can shoot a single energy beam at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits a ghost. The ghostbusters must all fire at the same time and no two energy beams may cross (it would be bad). The positions of the ghosts and ghostbusters is fixed in the plane (assume that no three points are collinear).

1. Prove that for any configuration of ghosts and ghostbusters there exists such a non-crossing matching.
2. Show that there exists a line passing through one ghostbuster and one ghost such that the number of ghostbusters on one side of the line equals the number of ghosts on the same side. Give an efficient algorithm to find such a line.
3. Give an efficient divide and conquer algorithm to pair ghostbusters and ghosts so that no two streams cross.

47.1.6. Homework 5

CS 373: Combinatorial Algorithms, Fall 2001

<http://www-courses.cs.uiuc.edu/~cs373>

Homework 5 (due Thu. Nov. 15, 2001 at 11:59:59 pm)

Required Problems

1 *Closest-Pair* (16 PTS.)

In the following, you are given a set P of n points in the plane. We are interested in the closet-pair of points in P . Those are the two points that realizes the distance $CP(P) = \min_{p,q \in P} |pq|$.

- (2 PTS.) Show that deciding if two points of P are equal takes $\Omega(n \log n)$ time in the worst case in the decision tree model with bounded degree. In particular, conclude that finding the closest-pair in P requires $\Omega(n \log n)$ time.
- (4 PTS.) Show that given a parameter r , you can decide whether $CP(P) \geq r$ in linear time. In particular, if there exists two points, $p', q' \in P$ such that $|p'q'| < r$, then the algorithm would print them out. Hint: Use a grid to partition the plane into squares of size r , and a hash table.
- (2 PTS.) Modify the above data-structure so that it supports insertions in $O(1)$ time. Namely, when we create it, we specify r . Next, we can perform insertions of points. The data-structure return FALSE after an insertion as soon as there are two points that were inserted that have distance smaller than r between them. It also provide those two points.
- (2 PTS.) For a random permutation of the points of P : p_1, \dots, p_n , let $d_i = CP(p_1, \dots, p_i)$ be the distance between the closest-pair in p_1, \dots, p_i . Show that the probability that $d_i \neq d_{i-1}$ is $O(1/i)$.
- (2 PTS.) Describe an algorithm that uses (c), that computes d_1, \dots, d_n in expected linear time.
- (2 PTS.) Show an expected linear-time algorithm that computes the closest-pair of points of P .
- (2 PTS.) (a) states that we can not solve the closest-pair problem in time faster than $\Omega(n \log n)$ time, and the other hand, (f) describes an algorithm that works in linear time. How do you explain those contradictory results?

2 *The convolution theorem* (10 PTS.)

Given two vectors $A = [a_0, a_1, \dots, a_n]$ and $B = [b_0, \dots, b_n]$ it is sometimes useful to have all the dot products of A_r and B , for $r = 0, \dots, 2n$, where $A_r = [a_{n-r}, a_{n+1-r}, a_{n+2-r}, \dots, a_{2n-r}]$, where $a_j = 0$ if $j \notin [0, \dots, n]$. (Namely, A_n is just A , and A_i is just a translation of A , where we pad with zero if needed. For example, for $A = [3, 7, 9, 15]$, we have $A_2 = [7, 9, 15, 0]$, and $A_5 = [0, 0, 3, 7]$.)

We would like to compute $c_i = A_i \cdot B$. The resulting vector $[c_0, \dots, c_{2n}]$ is known as the *convolution* of A and B .^①

- (4 PTS.) Let $p(x) = \sum_{i=0}^n \alpha_i x^i$, and $q(x) = \sum_{i=0}^n \beta_i x^i$. Write an explicit formula for the coefficient of x^i in the product polynomial $r(x) = p(x)q(x)$.
- (6 PTS.) Show how to compute the convolution of two vectors A and B of length n in time $O(n \log n)$.

3 *Random string matching* (10 PTS.)

Suppose that pattern P and text T are randomly chosen strings of length m and n , respectively, from the d -ary alphabet $\Sigma_d = \{0, 1, \dots, d-1\}$, where $d \geq 2$. Show that the expected number of character-to-character comparisons made by the inner loop of the ALMOSTBRUTEFORCE algorithm is $(n-m+1) \frac{1-d^{-m}}{1-d^{-1}} \leq 2(n-m+1)$.

^①The standard definition is in fact slightly different, but let us ignore it for the sake of this exercise.

```

ALMOSTBRUTEFORCE( $T[1..n], P[1..m]$ ):
  for  $s \leftarrow 1$  to  $n - m + 1$ 
     $equal \leftarrow true$ 
     $i \leftarrow 1$ 
    while  $equal$  and  $i \leq m$ 
      if  $T[s + i - 1] \neq P[i]$ 
         $equal \leftarrow false$ 
      else
         $i \leftarrow i + 1$ 
    if  $equal$ 
      return  $s$ 
  return  $-1$ 

```

4 *Multi-string matching* (10 PTS.)

Assume that you are given a text T of length n , and k patterns P_1, \dots, P_k (assume that no pattern is a prefix of another pattern), such that the total length of the patterns is m . Show how to find a single match of one of the P_i s with T in time $O(n + m)$. (Hint: Build a finite automata for each pattern, and “merge” this finite automata into a single large finite automata that works for all patterns simultaneously.)

5 *Fast Classification* (10 PTS.)

You are given k numbers a_1, \dots, a_k , and n numbers: b_1, \dots, b_n . For each b_i you should output whether it is equal to any of the a_i s, and if so the index of the a_i it is equal to.

- (4 PTS.) Show that any algorithm that falls under the decision tree model with bounded degree for this problem (for $k < n$) must take $\Omega(n \log k)$ time in the worst case.
- (2 PTS.) Show an algorithm for this problem with $O(n \log k)$ running time.
- (4 PTS.) Present a linear time algorithm for this problem, for the case where all of the a_i s and b_i s are integer numbers.

6 *Hamiltonian Graph* (10 PTS.)

Given an undirected graph G with n vertices, assume that you are given a subroutine $\text{EXISTENCE}(G)$ that decides whether there exists a cycle that passes through all the vertices of G *exactly once* in $O(n^c)$ time, where c is a constant. Such a cycle is called *Hamiltonian cycle*.

Present an algorithm that computes this cycle in $O(n^{2+c})$ time.

Conclude, that if all algorithms for computing the Hamiltonian cycle run in at least $\Omega(2^n)$ time in the worst case, then any algorithm for EXISTENCE must take at least $\Omega(2^n/n^2)$ time in the worst case.^②

Practice Problems

- 1** Prove that finding the second smallest of n elements takes EXACTLY $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. Prove for both upper and lower bounds. Hint: find the (first) smallest using an elimination tournament.

^②Be famous! Make money fast! Show that Hamiltonian cycle indeed requires $\Omega(2^n)$ time in the worst case. I would even give you an A+ in the course [although you would immediately get a PhD for that, so this might be slightly redundant.]

2 *Fibonacci strings* are defined as follows:

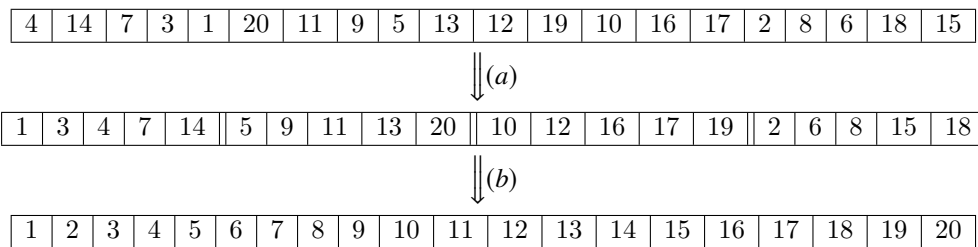
$$F_1 = \text{"b"}, \quad F_2 = \text{"a"}, \quad \text{and } F_n = F_{n-1}F_{n-2}, (n > 2)$$

where the recursive rule uses concatenation of strings, so F_3 is "ab", F_4 is "aba". Note that the length of F_n is the n th Fibonacci number.

1. Prove that in any Fibonacci string there are no two b's adjacent and no three a's.
2. Give the not-optimized and optimized 'prefix' (fail) function for F_7 .
3. Prove that, in searching for the Fibonacci string F_k , the unoptimized KMP algorithm can shift $\lceil k/2 \rceil$ times in a row trying to match the last character of the pattern. In other words, prove that there is a chain of failure links $m \rightarrow \text{fail}[m] \rightarrow \text{fail}[\text{fail}[m]] \rightarrow \dots$ of length $\lceil k/2 \rceil$, and find an example text T that would cause KMP to traverse this entire chain at a single text position.
4. Prove that the unoptimized KMP algorithm can shift $k - 2$ times in a row at the same text position when searching for F_k . Again, you need to find an example text T that would cause KMP to traverse this entire chain on the same text character.
5. How do the failure chains in parts (c) and (d) change if we use the optimized failure function instead?

3 Two-stage sorting

1. Suppose we are given an array $A[1..n]$ of distinct integers. Describe an algorithm that splits A into n/k subarrays, each with k elements, such that the elements of each subarray $A[(i - 1)k + 1..ik]$ are sorted. Your algorithm should run in $O(n \log k)$ time.
2. Given an array $A[1..n]$ that is already split into n/k sorted subarrays as in part (a), describe an algorithm that sorts the entire array in $O(n \log(n/k))$ time.
3. Prove that your algorithm from part (a) is optimal.
4. Prove that your algorithm from part (b) is optimal.

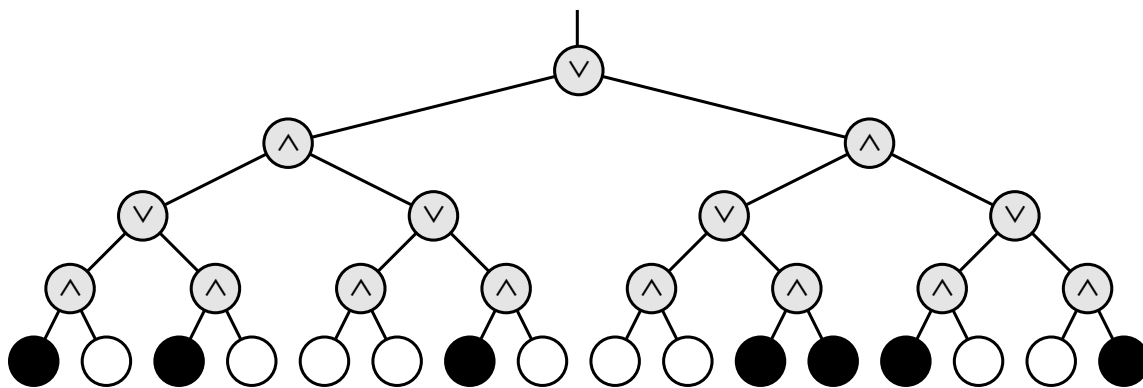


4 Show how to extend the Rabin-Karp fingerprinting method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. (The pattern may be shifted horizontally and vertically, but it may not be rotated.)

5 Death knocks on your door once more on a warm spring day. He remembers that you are an algorithms student and that you soundly defeated him last time and are now living out your immortality. Death is in a bit of a quandary. He has been losing a lot and doesn't know why. He wants you to prove a lower bound on your deterministic algorithm so that he can reap more souls. If you have forgotten, the game goes like this: It is a complete binary tree with 4^n leaves, each colored black or white. There is a token at the root of the tree. To play the game, you and Death took turns moving the token from its current node to one of its children. The game ends after $2n$ moves, when the token lands on a leaf. If the final leaf is black, the player dies; if it's white, you will live forever. You move first, so Death gets the last turn.

You decided whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are *and* gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should've challenge Death to a game of Twister instead.

Prove that *any* deterministic algorithm must examine *every* leaf of the tree in the worst case. Since there are 4^n leaves, this implies that any deterministic algorithm must take $\Omega(4^n)$ time in the worst case. Use an adversary argument, or in other words, assume Death cheats.



6 [This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

Lower Bounds on Adjacency Matrix Representations of Graphs

1. Prove that the time to determine if an undirected graph has a cycle is $\Omega(V^2)$.
2. Prove that the time to determine if there is a path between two nodes in an undirected graph is $\Omega(V^2)$.

7 String matching with wild-cards

Suppose you have an alphabet for patterns that includes a 'gap' or wild-card character; any length string of any characters can match this additional character. For example if '*' is the wild-card, then the pattern 'foo*bar*nad' can be found in 'foofoowangbarnad'. Modify the computation of the prefix function to correctly match strings using KMP.

8 Prove that there is no comparison sort whose running time is linear for at least $1/2$ of the $n!$ inputs of length n . What about at least $1/n$? What about at least $1/2^n$?

9 Prove that $2n - 1$ comparisons are necessary in the worst case to merge two sorted lists containing n elements each.

10 Find asymptotic upper and lower bounds to $\lg(n!)$ without Stirling's approximation (Hint: use integration).

11 Given a sequence of n elements of n/k blocks (k elements per block) all elements in a block are less than those to the right in sequence, show that you cannot have the whole sequence sorted in better than $\Omega(n \lg k)$. Note that the entire sequence would be sorted if each of the n/k blocks were individually sorted in place. Also note that combining the lower bounds for each block is not adequate (that only gives an upper bound).

12 Show how to find the occurrences of pattern P in text T by computing the prefix function of the string PT (the concatenation of P and T).

CS 373: Combinatorial Algorithms, Fall 2001

Homework 6 (due Thursday, Dec 6, 2001 at 11:59.99 p.m.)

Required Problems

1 VERTEX COVER (10 PTS.)

VERTEX COVER

Instance: A graph $G = (V, E)$ and a positive integer $K \leq |V|$.

Question: Is there a *vertex cover* of size K or less for G , that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and for each edge $\{u, v\} \in E$, at least one of u and v belongs to V' ?

- 1.A. (2 PTS.) Show that VERTEX COVER is NP-Complete. Hint: Do a reduction from INDEPENDENT SET to VERTEX COVER.
- 1.B. (3 PTS.) Show a polynomial approximation algorithm to the VERTEX-COVER problem which is a factor 2 approximation of the optimal solution. Namely, your algorithm should output a set $X \subseteq V$, such that X is a vertex cover, and $|X| \leq 2K_{opt}$, where K_{opt} is the cardinality of the smallest vertex cover of G .^①
- 1.C. (2 PTS.) Present a linear time algorithm that solves this problem for the case that G is a tree.
- 1.D. (3 PTS.) For a constant k , a graph G is k -separable, if there are k vertices of G , such that if we remove them from G , each one of the remaining connected components has at most $(2/3)n$ vertices, and furthermore each one of those connected components is also k -separable. (More formally, a graph $G = (V, E)$ is k -separable, if for any subset of vertices $S \subseteq V$, there exists a subset $M \subseteq S$, such that each connected component of $G_{S \setminus M}$ has at most $(2/3)|S|$ vertices, and $|M| \leq k$.)
- Show that given a graph G which is k -separable, one can compute the optimal VERTEX COVER in $n^{O(k)}$ time.

2 BIN PACKING (14 PTS.)

BIN PACKING

Instance: Finite set U of items, a size $s(u) \in \mathbb{Z}^+$ for each $u \in U$, an integer bin capacity B , and a positive integer K .

Question: Is there a partition of U into disjoint sets U_1, \dots, U_K such that the sum of the sizes of the items inside each U_i is B or less?

- 2.A. (2 PTS.) Show that the BIN PACKING problem is NP-Complete

^①It was very recently shown (I. Dinur and S. Safra. On the importance of being biased. Manuscript. <http://www.math.ias.edu/~iritd/mypapers/vc.pdf>, 2001.) that doing better than 1.3600 approximation to VERTEX COVER is NP-Hard. In your free time you can try and improve this constant. Good luck.

- 2.B. (2 PTS.) In the optimization variant of BIN PACKING one has to find the minimum number of bins needed to contain all elements of U . Present an algorithm that is a factor two approximation to optimal solution. Namely, it outputs a partition of U into M bins, such that the total size of each bin is at most B , and $M \leq k_{opt}$, where k_{opt} is the minimum number of bins of size B needed to store all the given elements of U .
- 2.C. (4 PTS.) Assume that B is bounded by an integer constant m . Describe a polynomial algorithm that computes the solution that uses the minimum number of bins to store all the elements.
- 2.D. (2 PTS.) Show that the following problem is NP-Complete.

TILING

Instance: Finite set \mathcal{R} of rectangles and a rectangle R in the plane.

Question: Is there a way of placing the rectangles of \mathcal{R} inside R , so that no pair of the rectangles intersect, and all the rectangles have their edges parallel of the edges of R ?

- 2.E. (4 PTS.) Assume that \mathcal{R} is a set of squares that can be arranged as to tile R completely. Present a polynomial time algorithm that computes a subset $\mathcal{T} \subseteq \mathcal{R}$, and a tiling of \mathcal{T} , so that this tiling of \mathcal{T} covers, say, 10% of the area of R .

3 MINIMUM SET COVER

(12 PTS.)

MINIMUM SET COVER

Instance: Collection C of subsets of a finite set S and an integer k .

Question: Are there k sets S_1, \dots, S_k in C such that $S \subseteq \cup_{i=1}^k S_i$?

- (2 PTS.) Prove that MINIMUM SET COVER problem is NP-Complete
- (3 PTS.) The greedy approximation algorithm for MINIMUM SET COVER, works by taking the largest set in $X \in C$, remove all all the elements of X from S and also from each subset of C . The algorithm repeat this until all the elements of S are removed. Prove that the number of elements not covered after k_{opt} iterations is at most $n/2$, where k_{opt} is the smallest number of sets of C needed to cover S , and $n = |S|$.
- (2 PTS.) Prove the greedy algorithm is $O(\log n)$ factor optimal approximation.
- (1 PTS.) Prove that the following problem is NP-Complete.

HITTING SET

Instance: A collection C of subsets of a set S , a positive integer K .

Question: Does S contain a *hitting set* for C of size K or less, that is, a subset $S' \subseteq S$ with $|S'| \leq K$ and such that S' contains at least one element from each subset in C .

- (4 PTS.) Given a set \mathcal{I} of n intervals on the real line, show a $O(n \log n)$ time algorithm that computes the smallest set of points X on the real line, such that for every interval $I \in \mathcal{I}$ there is a point $p \in X$, such that $p \in I$.

4 k -CENTER Problem

(20 PTS.)

***k*-CENTER**

Instance: A set P of n points in the plane, and an integer k and a radius r .

Question: Is there a cover of the points of P by k disks of radius (at most) r ?

1. (4 PTS.) Describe an $n^{O(k)}$ time algorithm that solves this problem.
2. (3 PTS.) There is a very simple and natural algorithm that achieves a 2-approximation for this cover: First it select an arbitrary point as a center (this point is going to be the center of one of the k covering disks). Then it computes the point that it furthest away from the current set of centers as the next center, and it continue in this fashion till it has k -points, which are the resulting centers. The smallest k equal radius disks centered at those points are the required k disks.
Show an implementation of this approximation algorithm in $O(nk)$ time.
3. (3 PTS.) Prove that that the above algorithm is a factor two approximation to the optimal cover. Namely, the radius of the disks output $\leq 2r_{opt}$, where r_{opt} is the smallest radius, so that we can find k -disks that cover the point-set.
4. (4 PTS.) Provide an ε -approximation algorithm for this problem. Namely, given k and a set of points P in the plane, your algorithm would output k -disks that cover the points and their radius is $\leq (1 + \varepsilon)r_{opt}$, where r_{opt} is the minimum radius of such a cover of P .
5. (2 PTS.) Prove that dual problem r -DISK-COVER problem is NP-Hard. In this problem, given P and a radius r , one should find the smallest number of disks of radius r that cover P .
6. (4 PTS.) Describe an approximation algorithm to the r -DISK COVER problem. Namely, given a point-set P and a radius r , outputs k disks, so that the k disks cover P and are of radius r , and $k = O(k_{opt})$, where k_{opt} is the minimal number of disks needed to cover P by disks of radius r .

5 MAX 3SAT Problem

(10 PTS.)

MAX SAT

Instance: Set U of variables, a collection C of disjunctive clauses of literals where a literal is a variable or a negated variable in U .

Question: Find an assignment that maximized the number of clauses of C that are being satisfied.

1. (3 PTS.) Prove that MAX SAT is NP-Hard.
2. (2 PTS.) Prove that if each clause has exactly three literals, and we randomly assign to the variables values 0 or 1, then the expected number of satisfied clauses is $(7/8)M$, where $M = |C|$.
3. (1 PTS.) Show that for any instance of MAX SAT, where each clause has exactly three different literals, there exists an assignment that satisfies at least $7/8$ of the clauses.
4. (4 PTS.) Let (U, C) be an instance of MAX SAT such that each clause has $\geq 10 \cdot \log n$ distinct variables, where n is the number of clauses. Prove that there exists a satisfying assignment. Namely, there exists an assignment that satisfy all the clauses of C .

Practice Problems

1 Complexity

1. Prove that $P \subseteq \text{co-NP}$.
2. Show that if $\text{NP} \neq \text{co-NP}$, then *every* NP-complete problem is *not* a member of co-NP.

2 2-CNF-SAT

Prove that deciding satisfiability when all clauses have at most 2 literals is in P.

3 Graph Problems

1. SUBGRAPH-ISOMORPHISM
Show that the problem of deciding whether one graph is a subgraph of another is NP-complete.
2. LONGEST-PATH
Show that the problem of deciding whether an unweighted undirected graph has a path of length greater than k is NP-complete.

4 PARTITION, SUBSET-SUM

PARTITION is the problem of deciding, given a set of numbers, whether there exists a subset whose sum equals the sum of the complement, i.e. given $S = s_1, s_2, \dots, s_n$, does there exist a subset S' such that $\sum_{s \in S'} s = \sum_{t \in S - S'} t$. SUBSET-SUM is the problem of deciding, given a set of numbers and a target sum, whether there exists a subset whose sum equals the target, i.e. given $S = s_1, s_2, \dots, s_n$ and k , does there exist a subset S' such that $\sum_{s \in S'} s = k$. Give two reductions, one in both directions.

- 5 BIN-PACKING Consider the bin-packing problem: given a finite set U of n items and the positive integer size $s(u)$ of each item $u \in U$, can U be partitioned into k disjoint sets U_1, \dots, U_k such that the sum of the sizes of the items in each set does not exceed B ? Show that the bin-packing problem is NP-Complete. [Hint: Use the result from the previous problem.]

6 3SUM

[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]

Describe an algorithm that solves the following problem as quickly as possible: Given a set of n numbers, does it contain three elements whose sum is zero? For example, your algorithm should answer TRUE for the set $\{-5, -17, 7, -4, 3, -2, 4\}$, since $-5 + 7 + (-2) = 0$, and FALSE for the set $\{-6, 7, -4, -13, -2, 5, 13\}$.

- 7 Consider finding the median of 5 numbers by using only comparisons. What is the exact worst case number of comparisons needed to find the median. Justify (exhibit a set that cannot be done in one less comparisons). Do the same for 6 numbers.

8 EXACT-COVER-BY-4-SETS

The EXACT-COVER-BY-3-SETS problem is defined as the following: given a finite set X with $|X| = 3q$ and a collection C of 3-element subsets of X , does C contain an *exact cover* for X , that is, a subcollection $C' \subseteq C$ such that every element of X occurs in exactly one member of C' ?

Given that EXACT-COVER-BY-3-SETS is NP-complete, show that EXACT-COVER-BY-4-SETS is also NP-complete.

9 PLANAR-3-COLOR

Using 3-COLOR, and the ‘gadget’ in Figure 47.3, prove that the problem of deciding whether a planar graph can be 3-colored is NP-complete. Hint: show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.

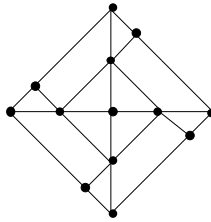


Figure 47.3: Gadget for PLANAR-3-COLOR.

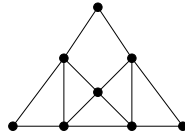


Figure 47.4: Gadget for DEGREE-4-PLANAR-3-COLOR.

10 DEGREE-4-PLANAR-3-COLOR

Using the previous result, and the ‘gadget’ in [Figure 47.4](#), prove that the problem of deciding whether a planar graph with no vertex of degree greater than four can be 3-colored is NP-complete. Hint: show that you can replace any vertex with degree greater than 4 with a collection of gadgets connected in such a way that no degree is greater than four.

11 Poly time subroutines can lead to exponential algorithms

Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

- 12**
1. Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is non-hamiltonian. Give a polynomial time algorithm for finding a **hamiltonian cycle** in an undirected bipartite graph or establishing that it does not exist.
 2. Show that the **hamiltonian-path** problem can be solved in polynomial time on directed acyclic graphs by giving an efficient algorithm for the problem.
 3. Explain why the results in previous questions do not contradict the facts that both HAM-CYCLE and HAM-PATH are NP-complete problems.

13 Consider the following pairs of problems:

- 13.A.** MIN SPANNING TREE and MAX SPANNING TREE
- 13.B.** SHORTEST PATH and LONGEST PATH
- 13.C.** TRAVELING SALESMAN PROBLEM and VACATION TOUR PROBLEM (the longest tour is sought).
- 13.D.** MIN CUT and MAX CUT (between s and t)
- 13.E.** EDGE COVER and VERTEX COVER
- 13.F.** TRANSITIVE REDUCTION and MIN EQUIVALENT DIGRAPH

(all of these seem dual or opposites, except the last, which are just two versions of minimal representation of a graph).

Which of these pairs are polytime equivalent and which are not? Why?

(Really HARD)

14 GRAPH-ISOMORPHISM

Consider the problem of deciding whether one graph is isomorphic to another.

1. Give a brute force algorithm to decide this.
2. Give a dynamic programming algorithm to decide this.
3. Give an efficient probabilistic algorithm to decide this.
4. Either prove that this problem is NP-complete, give a poly time algorithm for it, or prove that neither case occurs.

15 Prove that PRIMALITY (Given n , is n prime?) is in $\text{NP} \cap \text{co-NP}$. Hint: co-NP is easy (what's a certificate for showing that a number is composite?). For NP, consider a certificate involving primitive roots and recursively their primitive roots. Show that knowing this tree of primitive roots can be checked to be correct and used to show that n is prime, and that this check takes poly time.

16 How much wood would a woodchuck chuck if a woodchuck could chuck wood?

47.2. Midterm

47.3. Final

Chapter 48

Spring 2002

Chapter 49

Fall 2002

Chapter 50

Spring 2003

50.1. Homeworks

50.1.1. Homework 0

- 1** (10 PTS.) Sort the following 25 functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice.

$n^{3.5} - (n-1)^{3.5}$	n	$n^{2.1}$	$\lg^*(n/2)$	$1 + \lg n$
$\sin n + 2$	$\lg(\lg^* n)$	$\lg n!$	$(\lg n)^{\lg^* n}$	n^3
$\lg^* 2^n$	$2^{\lg^* n}$	e^n	$\lfloor \lg \lg(n!) \rfloor$	$\sum_{i=1}^n \frac{1}{i^2}$
$n^{3/(2n)}$	$n^{3/(2 \lg n)}$	$(\lg n)^{(n/2)}$	$(\lg(7+n))^{\lg n}$	$(1 + \frac{1}{2000})^{2000n}$
$n^{1/\lg \lg n}$	$n^{\lg \lg n}$	$\lg^{(200)} n$	$n^{1/1000}$	$n(\lg n)^2$

To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2 , n , $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

- 2** (10 PTS.) Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied. Extra credit will be given for more exact solutions.

1. (1 PTS.) $A(n) = A(n/4) + n \log \log n$
2. (1 PTS.) $B(n) = \min_{0 < k < n} (7B(k)B(n-k))$.
3. (1 PTS.) $C(n) = 3C(\lceil n/2 \rceil - 5) + n \log n$
4. (1 PTS.) $D(n) = \frac{n}{n-1}D(n-1) + 1$
5. (1 PTS.) $E(n) = E(\lfloor 2n/3 \rfloor) + 1/n$
6. (1 PTS.) $F(n) = F(\lfloor \log n \rfloor) + \log n$ (**HARD**)
7. (1 PTS.) $G(n) = \sqrt{n} + 4\sqrt{n} \cdot G(\lfloor \sqrt{n} \rfloor)$
8. (1 PTS.) $H(n) = \log(H(n-1)) + 1$
9. (1 PTS.) $I(n) = I(\lfloor \sqrt{n} \rfloor) + 1$
10. (1 PTS.) $J(n) = J(\lfloor n - \sqrt{n} \rfloor) + 1$

3 (10 PTS.)

1. Use induction to prove that in a simple graph, every walk between a pair of vertices, u, v , contains a path between u and v . Recall that a walk is a list of the form $v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k$, in which e_i has endpoints v_{i-1} and v_i .
2. Prove that a graph is connected if and only if for every partition of its vertices into two nonempty sets, there exists an edge that has endpoints in both sets.

4 (10 PTS.)

A *random walk* is a walk on a graph G , generated by starting from a vertex $v_0 = v \in V(G)$, and in the i -th stage, for $i > 0$, randomly selecting one of the neighbors of v_{i-1} and setting v_i to be this vertex. A walk v_0, v_1, \dots, v_m is of length m .

1. For a vertex $u \in V(G)$, let $P_u(m, v)$ be the probability that a random walk of length m , starting from u , visits v (i.e., $v_i = v$ for some i).
Prove that a graph G with n vertices is connected, if and only if, for any two vertices $u, v \in V(G)$, we have $P_u(n-1, v) > 0$.
2. Prove that a graph G with n vertices is connected if and only if for any pair of vertices $u, v \in V(G)$, we have $\lim_{m \rightarrow \infty} P_u(m, v) = 1$.

5 (10 PTS.)

1. Let $f_i(n)$ be a sequence of functions, such that for every i , $f_i(n) = o(\sqrt{n})$ (namely, $\lim_{n \rightarrow \infty} \frac{f_i(n)}{\sqrt{n}} = 0$). Let $g(n) = \sum_{i=1}^n f_i(n)$. Prove or disprove: $g(n) = o(n^{3/2})$.
2. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Prove or disprove:
 - $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$
 - $f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$
 - $f_1(n)^{f_2(n)} = O(g_1(n)^{g_2(n)})$

6 (10 PTS.)

Describe a data structure that supports storing temperatures. The operations on the data structure are as follows:

Insert(t, d) — Insert the temperature t that was measured on date d . Each temperature is a real number between -100 and 150 . For example, **insert**(22, "1/20/03").

$\text{Average}(d_1, d_2)$ report what is the average of all temperatures that were measured between date d_1 and date d_2 .

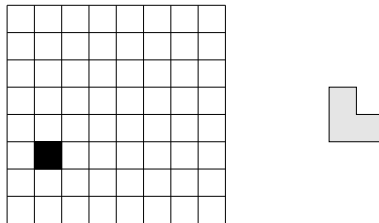
Each operation should take time $O(\log n)$, where n is the number of dates stored in the data structure. You can assume that a date is just an integer which specifies the number of days since the first of January 1970.

7 (10 PTS.)

We toss a fair coin n times. What is the expected number of “runs”? Runs are consecutive tosses with the same result. For example, the toss sequence HHTTHTH has 5 runs.

8 (10 PTS.)

Consider a $2^n \times 2^n$ chessboard with one (arbitrarily chosen) square removed, as in the following picture (for $n = 3$):



Prove that any such chessboard can be tiled without gaps or overlaps by L-shapes consisting of 3 squares each.

Practice Problems

The remaining problems are entirely for your benefit; similar questions will appear in every homework. Don't turn in solutions—we'll just throw them out—but feel free to ask us about practice questions during office hours and review sessions. Think of them as potential exam questions (hint, hint). We'll post solutions to *some* of the practice problems after the homeworks are due.

- 1** Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m .

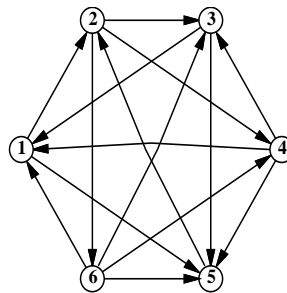
- F_n is even if and only if n is divisible by 3.
- $\sum_{i=0}^n F_i = F_{n+2} - 1$
- $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$ (**Really HARD**)
- If n is an integer multiple of m , then F_n is an integer multiple of F_m .

- 2** 1. Prove the following identity by induction:

$$\binom{2n}{n} = \sum_{k=0}^n \binom{n}{k} \binom{n}{n-k}.$$

- Give a non-inductive combinatorial proof of the same identity, by showing that the two sides of the equation count exactly the same thing in two different ways. There is a correct one-sentence proof.

- 3** A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once. Prove that every tournament contains at least one Hamiltonian path.



A six-vertex tournament containing the Hamiltonian path $6 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1$.

- 4** Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied.

- $A(n) = A(n/2) + n$
- $B(n) = 2B(n/2) + n$ (**Really HARD**)
- $C(n) = n + \frac{1}{2}(C(n-1) + C(3n/4))$
- $D(n) = \max_{n/3 < k < 2n/3} (D(k) + D(n-k) + n)$ (**HARD**)

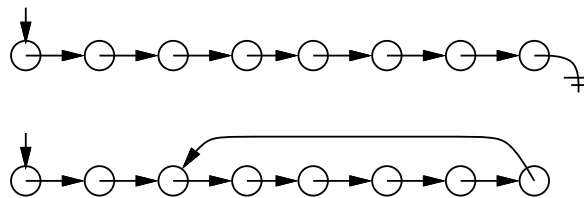
5. $E(n) = 2E(n/2) + n/\lg n$ (**HARD**)
6. $F(n) = \frac{F(n-1)}{F(n-2)}$, where $F(1) = 1$ and $F(2) = 2$. (**HARD**)
7. $G(n) = G(n/2) + G(n/4) + G(n/6) + G(n/12) + n$ [Hint: $\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} = 1$.] (**HARD**)
8. $H(n) = n + \sqrt{n} \cdot H(\sqrt{n})$ (**HARD**)
9. $I(n) = (n-1)(I(n-1) + I(n-2))$, where $F(0) = F(1) = 1$ (**HARD**)
10. $J(n) = 8J(n-1) - 15J(n-2) + 1$

- 5**
1. Prove that $2^{\lceil \lg n \rceil + \lfloor \lg n \rfloor} = \Theta(n^2)$.
 2. Prove or disprove: $2^{\lfloor \lg n \rfloor} = \Theta(2^{\lceil \lg n \rceil})$.
 3. Prove or disprove: $2^{2^{\lfloor \lg \lg n \rfloor}} = \Theta(2^{2^{\lceil \lg \lg n \rceil}})$.
 4. Prove or disprove: If $f(n) = O(g(n))$, then $\log(f(n)) = O(\log(g(n)))$.
 5. Prove or disprove: If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$. (**HARD**)
 6. Prove that $\log^k n = o(n^{1/k})$ for any positive integer k .

6 This problem asks you to simplify some recursively defined boolean formulas as much as possible. In each case, prove that your answer is correct. Each proof can be just a few sentences long, but it must be a *proof*.

1. Suppose $\alpha_0 = p$, $\alpha_1 = q$, and $\alpha_n = (\alpha_{n-2} \wedge \alpha_{n-1})$ for all $n \geq 2$. Simplify α_n as much as possible. [Hint: What is α_5 ?]
2. Suppose $\beta_0 = p$, $\beta_1 = q$, and $\beta_n = (\beta_{n-2} \leftrightarrow \beta_{n-1})$ for all $n \geq 2$. Simplify β_n as much as possible. [Hint: What is β_5 ?]
3. Suppose $\gamma_0 = p$, $\gamma_1 = q$, and $\gamma_n = (\gamma_{n-2} \Rightarrow \gamma_{n-1})$ for all $n \geq 2$. Simplify γ_n as much as possible. [Hint: What is γ_5 ?]
4. Suppose $\delta_0 = p$, $\delta_1 = q$, and $\delta_n = (\delta_{n-2} \bowtie \delta_{n-1})$ for all $n \geq 2$, where \bowtie is some boolean function with two arguments. Find a boolean function \bowtie such that $\delta_n = \delta_m$ if and only if $n - m$ is a multiple of 4. [Hint: There is only one such function.]

7 Suppose you have a pointer to the head of singly linked list. Normally, each node in the list only has a pointer to the next element, and the last node's pointer is NULL. Unfortunately, your list might have been corrupted by a bug in somebody else's code^①, so that the last node has a pointer back to some other node in the list instead.



Top: A standard linked list. Bottom: A corrupted linked list.

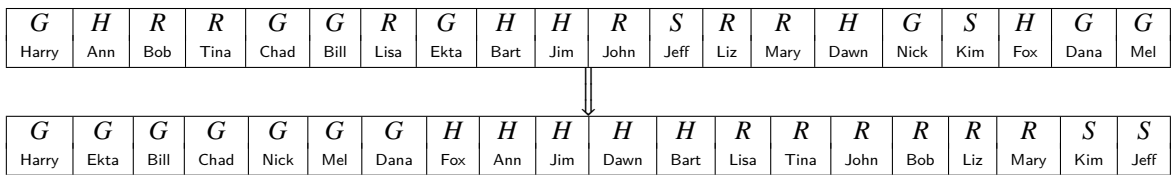
Describe an algorithm that determines whether the linked list is corrupted or not. Your algorithm must not modify the list. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the list, and use $O(1)$ extra space (not counting the list itself).

^①After all, *your* code is always completely 100% bug-free. Isn't that right, Mr. Gates?

8 Every year, upon their arrival at Hogwarts School of Witchcraft and Wizardry, new students are sorted into one of four houses (Gryffindor, Hufflepuff, Ravenclaw, or Slytherin) by the Hogwarts Sorting Hat. The student puts the Hat on their head, and the Hat tells the student which house they will join. This year, a failed experiment by Fred and George Weasley filled almost all of Hogwarts with sticky brown goo, mere moments before the annual Sorting. As a result, the Sorting had to take place in the basement hallways, where there was so little room to move that the students had to stand in a long line.

After everyone learned what house they were in, the students tried to group together by house, but there was too little room in the hallway for more than one student to move at a time. Fortunately, the Sorting Hat took CS 373 many years ago, so it knew how to group the students as quickly as possible. What method did the Sorting Hat use?

1. More formally, you are given an array of n items, where each item has one of four possible values, possibly with a pointer to some additional data. Describe an algorithm² that rearranges the items into four clusters in $O(n)$ time using only $O(1)$ extra space.



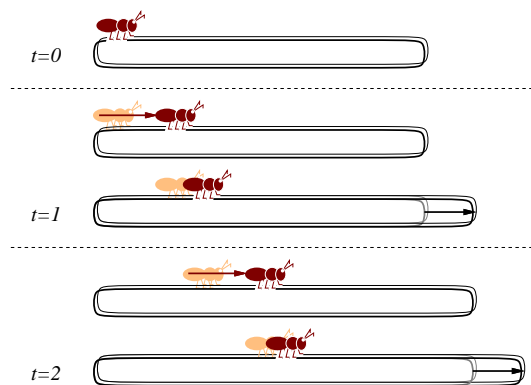
2. Describe an algorithm for the case where there are k possible values (i.e., $1, 2, \dots, k$) that rearranges the items using only $O(\log k)$ extra space. How fast is your algorithm? (A faster algorithm would get more credit)
3. Describe a faster algorithm (if possible) for the case when $O(k)$ extra space is allowed. How fast is your algorithm?

(HARD)

4. Optional practice exercise - *no credit*: Provide a fast algorithm that uses only $O(1)$ additional space for the case where there are k possible values.

(HARD)

9 An ant is walking along a rubber band, starting at the left end. Once every second, the ant walks one inch to the right, and then you make the rubber band one inch longer by pulling on the right end. The rubber band stretches uniformly, so stretching the rubber band also pulls the ant to the right. The initial length of the rubber band is n inches, so after t seconds, the rubber band is $n + t$ inches long.

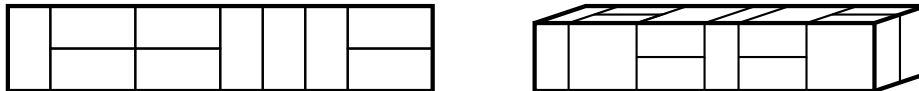


Every second, the ant walks an inch, and then the rubber band is stretched an inch longer.

²Since you've read the Homework Instructions, you know what the phrase 'describe an algorithm' means. Right?

- How far has the ant moved after t seconds, as a function of n and t ? Set up a recurrence and (for full credit) give an *exact* closed-form solution. [Hint: What *fraction* of the rubber band's length has the ant walked?]
- How long does it take the ant to get to the right end of the rubber band? For full credit, give an answer of the form $f(n) + \Theta(1)$ for some explicit function $f(n)$.

- 10**
- A *domino* is a 2×1 or 1×2 rectangle. How many different ways are there to completely fill a $2 \times n$ rectangle with n dominos? Set up a recurrence relation and give an *exact* closed-form solution.
 - A *slab* is a three-dimensional box with dimensions $1 \times 2 \times 2$, $2 \times 1 \times 2$, or $2 \times 2 \times 1$. How many different ways are there to fill a $2 \times 2 \times n$ box with n slabs? Set up a recurrence relation and give an *exact* closed-form solution.



A 2×10 rectangle filled with ten dominos, and a $2 \times 2 \times 10$ box filled with ten slabs.

- 11** Professor George O'Jungle has a favorite 26-node binary tree, whose nodes are labeled by letters of the alphabet. The preorder and postorder sequences of nodes are as follows:

preorder: M N H C R S K W T G D X I Y A J P O E Z V B U L Q F
 postorder: C W T K S G R H D N A O E P J Y Z I B Q L F U V X M

Draw Professor O'Jungle's binary tree, and give the inorder sequence of nodes.

- 12** Alice and Bob each have a fair n -sided die. Alice rolls her die once. Bob then repeatedly throws his die until he rolls a number at least as big as the number Alice rolled. Each time Bob rolls, he pays Alice \$1. (For example, if Alice rolls a 5, and Bob rolls a 4, then a 3, then a 1, then a 5, the game ends and Alice gets \$4. If Alice rolls a 1, then no matter what Bob rolls, the game will end immediately, and Alice will get \$1.)

Exactly how much money does Alice expect to win at this game? Prove that your answer is correct. If you have to appeal to 'intuition' or 'common sense', your answer is probably wrong!

- 13** Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ..., 52 of clubs. (They're big cards.) Penn shuffles the deck until each each of the $52!$ possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the five of clubs.

- On average, how many cards does Penn give Teller?
- On average, what is the smallest-numbered card that Penn gives Teller? (**HARD**)
- On average, what is the largest-numbered card that Penn gives Teller?

[Hint: Solve for an n -card deck and then set $n = 52$.] In each case, give *exact* answers and prove that they are correct. If you have to appeal to "intuition" or "common sense", your answers are probably wrong!

- 14** (10 PTS.) Evaluate the following summations; simplify your answers as much as possible. Significant partial credit will be given for answers in the form $\Theta(f(n))$ for some recognizable function $f(n)$.

14.A. (2 PTS.)
$$\sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{i}$$

(HARD)

14.B. (2 PTS.) $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{j}$

14.C. (2 PTS.) $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=j}^i \frac{1}{k}$

14.D. (2 PTS.) $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j \frac{1}{k}$

14.E. (2 PTS.) $\sum_{i=1}^n \sum_{j=1}^i \sum_{k=1}^j \frac{1}{j \cdot k}$

15 (10 PTS.) Prove that for any nonnegative parameters a and b , the following algorithms terminate and produce identical output. Also, provide bounds on the running times of those algorithms. Can you imagine any reason why WEIRDEUCLID would be preferable to FASTEUCLID?

```
SLOWEUCLID( $a, b$ ):
  if  $b > a$ 
    return SLOWEUCLID( $b, a$ )
  else if  $b = 0$ 
    return  $a$ 
  else
    return SLOWEUCLID( $b, a - b$ )
```

```
FASTEUCLID( $a, b$ ):
  if  $b = 0$ 
    return  $a$ 
  else
    return FASTEUCLID( $b, a \bmod b$ )
```

```
WEIRDEUCLID( $a, b$ ):
  if  $b = 0$ 
    return  $a$ 
  if  $a = 0$ 
    return  $b$ 
  if  $a$  is even and  $b$  is even
    return  $2 * \text{WEIRDEUCLID}(a/2, b/2)$ 
  if  $a$  is even and  $b$  is odd
    return WEIRDEUCLID( $a/2, b$ )
  if  $a$  is odd and  $b$  is even
    return WEIRDEUCLID( $a, b/2$ )
  if  $b > a$ 
    return WEIRDEUCLID( $b - a, a$ )
  else
    return WEIRDEUCLID( $a - b, b$ )
```

16 (10 PTS.) Suppose we have a binary search tree. You perform a long sequence of operations on the binary tree (insertion, deletions, searches, etc), and the maximum depth of the tree during those operations is at most h .

Modify the binary search tree T so that it supports the following operations. Implementing some of those operations would require you to modify the information stored in each node of the tree, and the way insertions/deletions are being handled in the tree. For each of the following, describe separately the changes made in detail, and the algorithms for answering those queries. (Note, that under the modified version of the binary search tree, insertion and deletion should still take $O(h)$ time, where h is the maximum height of the tree during all the execution of the algorithm.)

- (2 PTS.) Find the smallest element stored in T in $O(h)$ time.
- (2 PTS.) Given a query k , find the k -th smallest element stored in T in $O(h)$ time.
- (3 PTS.) Given a query $[a, b]$, find the number of elements stored in T with their values being in the range $[a, b]$, in $O(h)$ time.
- (3 PTS.) Given a query $[a, b]$, report (i.e., printout) all the elements stored in T in the range $[a, b]$, in $O(h + u)$ time, where u is the number of elements printed out.

17 (10 PTS.) There are n balls (numbered from 1 to n) and n boxes (numbered from 1 to n). We put each ball in a randomly selected box.

- (4 PTS.) A box may contain more than one ball. Suppose X is the number on the box that has the smallest number among all nonempty boxes. What is the expectation of X ? (It's OK to just give a big expression.)
- (4 PTS.) We put the balls into the boxes in such a way that there is exactly one ball in each box. If the number written on a ball is the same as the number written on the box containing the ball, we say there is a match. What is the expected number of matches?
- (2 PTS.) What is the probability that there are exactly k matches? ($1 \leq k < n$)

[Hint: If you have to appeal to “intuition” or “common sense”, your answers are probably wrong!]

50.1.2. Homework 1

1 PARTITION (10 PTS.)

The **Partition** satyr, the uncle of the deduction fairy, had visited you on winter break and gave you, as a token of appreciation, a black-box that can solve **Partition** in polynomial time (note that this black box solves the decision problem). Let S be a given set of n integer numbers. Describe a polynomial time algorithm that computes, using the black box, a partition of S if such a solution exists. Namely, your algorithm should output a subset $T \subseteq S$, such that

$$\sum_{s \in T} s = \sum_{s \in S \setminus T} s.$$

2 PARTITION REVISITED (10 PTS.)

Let S be an instance of partition, such that $n = |S|$, and $M = \max_{s \in S} s$. Show a polynomial time (in n and M) algorithm that solves partition.

3 POLY TIME SUBROUTINES CAN LEAD TO EXPONENTIAL ALGORITHMS (10 PTS.)

Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

4 WHY MIKE CAN NOT GET IT. (10 PTS.)

Not-3SAT

Instance: A 3CNF formula F

Question: Is F not satisfiable? (Namely, for all inputs for F , it evaluates to **FALSE**.)

1. Prove that **Not-3SAT** is *co-NP*.
2. Here is a proof that **Not-3SAT** is in *NP*: If the answer to the given instance is **Yes**, we provide the following proof to the verifier: We list every possible assignment, and for each assignment, we list the output (which is FALSE). Given this proof, of length L , the verifier can easily verify it in polynomial time in L . QED.
What is wrong with this proof?
3. Show that given a black-box that can solve **Not-3SAT**, one can find the satisfying assignment of a formula F in polynomial time, using polynomial number of calls to the black-box (if such an assignment exists).

5 NP-COMPLETENESS COLLECTION (20 PTS.)

Prove that the following problems are *NP-Complete*.

MINIMUM SET COVER

1. **Instance:** Collection C of subsets of a finite set S and an integer k .
Question: Are there k sets S_1, \dots, S_k in C such that $S \subseteq \bigcup_{i=1}^k S_i$?

HITTING SET

2. **Instance:** A collection C of subsets of a set S , a positive integer K .
Question: Does S contain a *hitting set* for C of size K or less, that is, a subset $S' \subseteq S$ with $|S'| \leq K$ and such that S' contains at least one element from each subset in C .

Hamiltonian Path

3. **Instance:** Graph $G = (V, E)$
Question: Does G contain a Hamiltonian path? (Namely a path that visits all vertices of G .)

Max Degree Spanning Tree

4. **Instance:** Graph $G = (V, E)$ and integer k
Question: Does G contain a spanning tree T where every node in T is of degree at most k ?

6 INDEPENDENCE (10 PTS.)

Let $G = (V, E)$ be an undirected graph over n vertices. Assume that you are given a numbering $\pi : V \rightarrow \{1, \dots, n\}$ (i.e., every vertex has a unique number), such that for any edge $ij \in E$, we have $|\pi(i) - \pi(j)| \leq 20$.
Either prove that it is *NP-Hard* to find the largest independent set in G , or provide a polynomial time algorithm.

Practice Problems

1 PARTITION (10 PTS.)

We already know the following problem is *NP-Complete*

SUBSET SUM

- Instance:** A finite set A and a “size” $s(a) \in \mathbb{Z}^+$ for each $a \in A$, an integer B .
Question: Is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = B$?

Now let's consider the following problem:

PARTITION

Instance: A finite set A and a "size" $s(a) \in \mathbb{Z}^+$ for each $a \in A$.

Question: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)?$$

Show that PARTITION is NP-Complete.

2 MINIMUM SET COVER (15 PTS.)

MINIMUM SET COVER

Instance: Collection C of subsets of a finite set S and an integer k .

Question: Are there k sets S_1, \dots, S_k in C such that $S \subseteq \cup_{i=1}^k S_i$?

1. (5 PTS.) Prove that MINIMUM SET COVER problem is NP-Complete
2. (5 PTS.) Prove that the following problem is NP-Complete.

HITTING SET

Instance: A collection C of subsets of a set S , a positive integer K .

Question: Does S contain a *hitting set* for C of size K or less, that is, a subset $S' \subseteq S$ with $|S'| \leq K$ and such that S' contains at least one element from each subset in C .

3. (5 PTS.) *Hitting set on the line*
Given a set \mathcal{I} of n intervals on the real line, show a $O(n \log n)$ time algorithm that computes the smallest set of points X on the real line, such that for every interval $I \in \mathcal{I}$ there is a point $p \in X$, such that $p \in I$.

3 BIN PACKING (10 PTS.)

BIN PACKING

Instance: Finite set U of items, a size $s(u) \in \mathbb{Z}^+$ for each $u \in U$, an integer bin capacity B , and a positive integer K .

Question: Is there a partition of U into disjoint sets U_1, \dots, U_K such that the sum of the sizes of the items inside each U_i is B or less?

1. (5 PTS.) Show that the BIN PACKING problem is NP-Complete
2. (5 PTS.) Show that the following problem is NP-Complete.

TILING

Instance: Finite set \mathcal{RECTS} of rectangles and a rectangle R in the plane.

Question: Is there a way of placing all the rectangles of \mathcal{RECTS} inside R , so that no pair of the rectangles intersect in their interior, and all the rectangles have their edges parallel of the edges of R ?

4 GRAPH ISOMORPHISMS (10 PTS.)

- (5 PTS.) Show that the following problem is NP-Complete.

SUBGRAPH ISOMORPHISM

Instance: Graphs $G = (V_1, E_1), H = (V_2, E_2)$.

Question: Does G contain a subgraph isomorphic to H , i.e., a subset $V \subseteq V_1$ and a subset $E \subseteq E_1$ such that $|V| = |V_2|$, $|E| = |E_2|$, and there exists a one-to-one function $f : V_2 \rightarrow V$ satisfying $\{u, v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E$?

- (5 PTS.) Show that the following problem is NP-Complete.

LARGEST COMMON SUBGRAPH

Instance: Graphs $G = (V_1, E_1), H = (V_2, E_2)$, positive integer K .

Question: Do there exist subsets $E'_1 \subseteq E_1$ and $E'_2 \subseteq E_2$ with $|E'_1| = |E'_2| \geq K$ such that the two subgraphs $G' = (V_1, E'_1)$ and $H' = (V_2, E'_2)$ are isomorphic?

5 KNAPSACK (15 PTS.)

- (5 PTS.) Show that the following problem is NP-Complete.

KNAPSACK

Instance: A finite set U , a "size" $s(u) \in \mathbb{Z}^+$ and a "value" $v(u) \in \mathbb{Z}^+$ for each $u \in U$, a size constraint $B \in \mathbb{Z}^+$, and a value goal $K \in \mathbb{Z}^+$.

Question: Is there a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq K$?

- (5 PTS.) Show that the following problem is NP-Complete.

MULTIPROCESSOR SCHEDULING

Instance: A finite set A of "tasks", a "length" $l(a) \in \mathbb{Z}^+$ for each $a \in A$, a number $m \in \mathbb{Z}^+$ of "processors", and a "deadline" $D \in \mathbb{Z}^+$.

Question: Is there a partition $A = A_1 \cup A_2 \cup \dots \cup A_m$ of A into m disjoint sets such that $\max\{\sum_{a \in A_i} l(a) : 1 \leq i \leq m\} \leq D$?

- SCHEDULING WITH PROFITS AND DEADLINES (5 PTS.)

Suppose you have one machine and a set of n tasks a_1, a_2, \dots, a_n . Each task a_j has a processing time t_j , a profit p_j , and a deadline d_j . The machine can process only one task at a time, and task a_j must run uninterruptedly for t_j consecutive time units to complete. If you complete task a_j by its deadline d_j , you receive a profit p_j . But you receive no profit if you complete it after its deadline. As an optimization problem, you are given the processing times, profits and deadlines for a set of n tasks, and you wish to find a schedule that completes all the tasks and returns the greatest amount of profit.

- (3 PTS.) State this problem as a decision problem.
- (2 PTS.) Show that the decision problem is NP-complete.

1 VERTEX COVER

VERTEX COVER

Instance: A graph $G = (V, E)$ and a positive integer $K \leq |V|$.

Question: Is there a *vertex cover* of size K or less for G , that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and for each edge $\{u, v\} \in E$, at least one of u and v belongs to V' ?

1. Show that VERTEX COVER is NP-Complete. Hint: Do a reduction from INDEPENDENT SET to VERTEX COVER.
2. Show a polynomial approximation algorithm to the VERTEX-COVER problem which is a factor 2 approximation of the optimal solution. Namely, your algorithm should output a set $X \subseteq V$, such that X is a vertex cover, and $|X| \leq 2K_{opt}$, where K_{opt} is the cardinality of the smallest vertex cover of G .^①
3. Present a linear time algorithm that solves this problem for the case that G is a tree.
4. For a constant k , a graph G is k -separable, if there are k vertices of G , such that if we remove them from G , each one of the remaining connected components has at most $(2/3)n$ vertices, and furthermore each one of those connected components is also k -separable. (More formally, a graph $G = (V, E)$ is k -separable, if for any subset of vertices $S \subseteq V$, there exists a subset $M \subseteq S$, such that each connected component of $G_{S \setminus M}$ has at most $(2/3)|S|$ vertices, and $|M| \leq k$.)
Show that given a graph G which is k -separable, one can compute the optimal VERTEX COVER in $n^{O(k)}$ time.

2 BIN PACKING

BIN PACKING

Instance: Finite set U of items, a size $s(u) \in \mathbb{Z}^+$ for each $u \in U$, an integer bin capacity B , and a positive integer K .

Question: Is there a partition of U into disjoint sets U_1, \dots, U_K such that the sum of the sizes of the items inside each U_i is B or less?

1. Show that the BIN PACKING problem is NP-Complete
2. In the optimization variant of BIN PACKING one has to find the minimum number of bins needed to contain all elements of U . Present an algorithm that is a factor two approximation to optimal solution. Namely, it outputs a partition of U into M bins, such that the total size of each bin is at most B , and $M \leq k_{opt}$, where k_{opt} is the minimum number of bins of size B needed to store all the given elements of U .
3. Assume that B is bounded by an integer constant m . Describe a polynomial algorithm that computes the solution that uses the minimum number of bins to store all the elements.
4. Show that the following problem is NP-Complete.

TILING

Instance: Finite set \mathcal{R} of rectangles and a rectangle R in the plane.

Question: Is there a way of placing the rectangles of \mathcal{R} inside R , so that no pair of the rectangles intersect, and all the rectangles have their edges parallel of the edges of R ?

5. Assume that \mathcal{R} is a set of squares that can be arranged as to tile R completely. Present a polynomial time algorithm that computes a subset $\mathcal{T} \subseteq \mathcal{R}$, and a tiling of R , so that this tiling of \mathcal{T} covers, say, 10% of the area of R .

3 MINIMUM SET COVER

^①It was very recently shown (I. Dinur and S. Safra. On the importance of being biased. Manuscript. <http://www.math.ias.edu/~iritd/mypapers/vc.pdf>, 2001.) that doing better than 1.3600 approximation to VERTEX COVER is NP-Hard. In your free time you can try and improve this constant. Good luck.

MINIMUM SET COVER

Instance: Collection C of subsets of a finite set S and an integer k .

Question: Are there k sets S_1, \dots, S_k in C such that $S \subseteq \cup_{i=1}^k S_i$?

1. Prove that MINIMUM SET COVER problem is NP-Complete
2. The greedy approximation algorithm for MINIMUM SET COVER, works by taking the largest set in $X \in C$, remove all the elements of X from S and also from each subset of C . The algorithm repeat this until all the elements of S are removed. Prove that the number of elements not covered after k_{opt} iterations is at most $n/2$, where k_{opt} is the smallest number of sets of C needed to cover S , and $n = |S|$.
3. Prove the greedy algorithm is $O(\log n)$ factor optimal approximation.
4. Prove that the following problem is NP-Complete.

HITTING SET

Instance: A collection C of subsets of a set S , a positive integer K .

Question: Does S contain a *hitting set* for C of size K or less, that is, a subset $S' \subseteq S$ with $|S'| \leq K$ and such that S' contains at least one element from each subset in C .

5. Given a set \mathcal{I} of n intervals on the real line, show a $O(n \log n)$ time algorithm that computes the smallest set of points X on the real line, such that for every interval $I \in \mathcal{I}$ there is a point $p \in X$, such that $p \in I$.

4 k -CENTER Problem

k -CENTER

Instance: A set P of n points in the plane, and an integer k and a radius r .

Question: Is there a cover of the points of P by k disks of radius (at most) r ?

1. Describe an $n^{O(k)}$ time algorithm that solves this problem.
2. There is a very simple and natural algorithm that achieves a 2-approximation for this cover: First it select an arbitrary point as a center (this point is going to be the center of one of the k covering disks). Then it computes the point that is furthest away from the current set of centers as the next center, and it continues in this fashion till it has k -points, which are the resulting centers. The smallest k equal radius disks centered at those points are the required k disks.
Show an implementation of this approximation algorithm in $O(nk)$ time.
3. Prove that the above algorithm is a factor two approximation to the optimal cover. Namely, the radius of the disks output $\leq 2r_{opt}$, where r_{opt} is the smallest radius, so that we can find k -disks that cover the point-set.
4. Provide an ε -approximation algorithm for this problem. Namely, given k and a set of points P in the plane, your algorithm would output k -disks that cover the points and their radius is $\leq (1 + \varepsilon)r_{opt}$, where r_{opt} is the minimum radius of such a cover of P .
5. Prove that dual problem r -DISK-COVER problem is NP-Hard. In this problem, given P and a radius r , one should find the smallest number of disks of radius r that cover P .
6. Describe an approximation algorithm to the r -DISK COVER problem. Namely, given a point-set P and a radius r , outputs k disks, so that the k disks cover P and are of radius r , and $k = O(k_{opt})$, where k_{opt} is the minimal number of disks needed to cover P by disks of radius r .

5 MAX 3SAT Problem

MAX SAT

Instance: Set U of variables, a collection C of disjunctive clauses of literals where a literal is a variable or a negated variable in U .

Question: Find an assignment that maximized the number of clauses of C that are being satisfied.

1. Prove that MAX SAT is NP-Hard.
2. Prove that if each clause has exactly three literals, and we randomly assign to the variables values 0 or 1, then the expected number of satisfied clauses is $(7/8)M$, where $M = |C|$.
3. Show that for any instance of MAX SAT, where each clause has exactly three different literals, there exists an assignment that satisfies at least $7/8$ of the clauses.
4. Let (U, C) be an instance of MAX SAT such that each clause has $\geq 10 \cdot \log n$ distinct variables, where n is the number of clauses. Prove that there exists a satisfying assignment. Namely, there exists an assignment that satisfies all the clauses of C .

6 Complexity

1. Prove that $P \subseteq \text{co-NP}$.

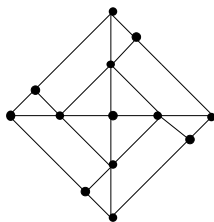


Figure 50.1: Gadget for PLANAR-3-COLOR.

2. Show that if $\text{NP} \neq \text{co-NP}$, then *every* NP-complete problem is *not* a member of co-NP.

7 2-CNF-SAT

Prove that deciding satisfiability when all clauses have at most 2 literals is in P.

8 Graph Problems

1. LONGEST-PATH

Show that the problem of deciding whether an unweighted undirected graph has a path of length greater than k is NP-complete.

9 PARTITION, SUBSET-SUM

PARTITION is the problem of deciding, given a set of numbers, whether there exists a subset whose sum equals the sum of the complement, i.e. given $S = s_1, s_2, \dots, s_n$, does there exist a subset S' such that $\sum_{s \in S'} s = \sum_{t \in S - S'} t$. SUBSET-SUM is the problem of deciding, given a set of numbers and a target sum, whether there exists a subset whose sum equals the target, i.e. given $S = s_1, s_2, \dots, s_n$ and k , does there exist a subset S' such that $\sum_{s \in S'} s = k$. Give two reduction, one in both directions.

10 3SUM

Describe an algorithm that solves the following problem as quickly as possible: Given a set of n numbers, does it contain three elements whose sum is zero? For example, your algorithm should answer TRUE for the set $\{-5, -17, 7, -4, 3, -2, 4\}$, since $-5 + 7 + (-2) = 0$, and FALSE for the set $\{-6, 7, -4, -13, -2, 5, 13\}$.

- 11 Consider finding the median of 5 numbers by using only comparisons. What is the exact worst case number of comparisons needed to find the median. Justify (exhibit a set that cannot be done in one less comparisons). Do the same for 6 numbers.

12 EXACT-COVER-BY-4-SETS

The EXACT-COVER-BY-3-SETS problem is defines as the following: given a finite set X with $|X| = 3q$ and a collection C of 3-element subsets of X , does C contain an *exact cover* for X , that is, a subcollection $C' \subseteq C$ such that every element of X occurs in exactly one member of C' ?

Given that EXACT-COVER-BY-3-SETS is NP-complete, show that EXACT-COVER-BY-4-SETS is also NP-complete.

13 PLANAR-3-COLOR

Using 3-COLOR, and the ‘gadget’ in Figure 50.1, prove that the problem of deciding whether a planar graph can be 3-colored is NP-complete. Hint: show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.

14 DEGREE-4-PLANAR-3-COLOR

Using the previous result, and the ‘gadget’ in Figure 50.2, prove that the problem of deciding whether a planar graph with no vertex of degree greater than four can be 3-colored is NP-complete. Hint: show

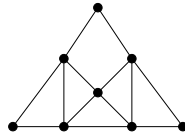


Figure 50.2: Gadget for DEGREE-4-PLANAR-3-COLOR.

that you can replace any vertex with degree greater than 4 with a collection of gadgets connected in such a way that no degree is greater than four.

15 Poly time subroutines can lead to exponential algorithms
 Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

- 16**
1. Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is non-hamiltonian. Give a polynomial time algorithm for finding a **hamiltonian cycle** in an undirected bipartite graph or establishing that it does not exist.
 2. Show that the **hamiltonian-path** problem can be solved in polynomial time on directed acyclic graphs by giving an efficient algorithm for the problem.
 3. Explain why the results in previous questions do not contradict the facts that both HAM-CYCLE and HAM-PATH are NP-complete problems.

17 Consider the following pairs of problems:

1. MIN SPANNING TREE and MAX SPANNING TREE
2. SHORTEST PATH and LONGEST PATH
3. TRAVELING SALESMAN PROBLEM and VACATION TOUR PROBLEM (the longest tour is sought).
4. MIN CUT and MAX CUT (between s and t)
5. EDGE COVER and VERTEX COVER
6. TRANSITIVE REDUCTION and MIN EQUIVALENT DIGRAPH

(all of these seem dual or opposites, except the last, which are just two versions of minimal representation of a graph).

Which of these pairs are polytime equivalent and which are not? Why?

(Really HARD)

18 GRAPH-ISOMORPHISM

Consider the problem of deciding whether one graph is isomorphic to another.

1. Give a brute force algorithm to decide this.
2. Give a dynamic programming algorithm to decide this.
3. Give an efficient probabilistic algorithm to decide this.
4. Either prove that this problem is NP-complete, give a poly time algorithm for it, or prove that neither case occurs.

19 Prove that PRIMALITY (Given n , is n prime?) is in $NP \cap co-NP$. Hint: $co-NP$ is easy (what's a certificate for showing that a number is composite?). For NP, consider a certificate involving primitive roots and recursively their primitive roots. Show that knowing this tree of primitive roots can be checked to be correct and used to show that n is prime, and that this check takes poly time.

50.1.3. Homework 2

CS 373: Combinatorial Algorithms, Spring 2003
Homework 2 (due Tuesday, Feb 18, 2003 at 11:59.99 p.m.)

Required Problems

1 FREE LUNCH.
(10 PTS.)

- 1.A. (3 PTS.) Provide a *detailed* description of the procedure that computes the *longest ascending subsequence* in a given sequence of n numbers. The procedure should use only arrays, and should output together with the length of the subsequence, the subsequence itself.
- 1.B. (4 PTS.) Provide a data-structure, that store pairs (a_i, b_i) of numbers, such that an insertion/deletion operation takes $O(\log n)$ time, where n is the total number of elements inserted. And furthermore, given a query interval $[\alpha, \beta]$, it can output in $O(\log n)$ time, the pair realizing

$$\max_{(a_i, b_i) \in S, a_i \in [\alpha, \beta]} b_i,$$

where S is the current set of pairs.

- 1.C. (3 PTS.) Using (b), describe an $O(n \log n)$ time algorithm for computing the longest ascending subsequence given a sequence of n numbers.

2 GREEDY ALGORITHM DOES NOT WORK FOR INDEPENDENT SET.
(20 PTS.)

A natural algorithm, GREEDYINDEPENDENT, for computing maximum independent set in a graph, is to repeatedly remove the vertex of lowest degree in the graph, and add it to the independent set, and remove all its neighbors.

1. (5 PTS.) Show an example, where this algorithm fails to output the optimal solution.
2. (5 PTS.) Let G be a $(k, k + 1)$ -uniform graph (this is a graph where every vertex has degree either k or $k + 1$). Show that the above algorithm outputs an independent set of size $\Omega(n/k)$, where n is the number of vertices in G .
3. (5 PTS.) Let G be a graph with average degree δ (i.e., $\delta = 2|E(G)|/|V(G)|$). Prove that the above algorithm outputs an independent set of size $\Omega(n/\delta)$.
4. (5 PTS.) For any integer k , present an example of a graph G_k , such that GREEDYINDEPENDENT outputs an independent set of size $\leq |OPT(G_k)|/k$, where $OPT(G_k)$ is the largest independent set in G_k . How many vertices and edges does G_k has? What is the average degree of G_k ?

3 GREEDY ALGORITHM DOES NOT WORK FOR VERTEXCOVER.
(10 PTS.)

Extend the example shown in class for the greedy algorithm for **Vertex Cover**. Namely, for any n , show a graph G_n , with n vertices, for which the greedy **Vertex Cover** algorithm, outputs a vertex cover which is of size $\Omega(Opt(G_n) \log n)$, where $Opt(G_n)$ is the cardinality of the smallest **Vertex Cover** of G_n .

4 GREEDY ALGORITHM DOES NOT WORK FOR TSP WITH THE TRIANGLE INEQUALITY.
(10 PTS.)

In the greedy Traveling Salesman algorithm, the algorithm starts from a starting vertex $v_1 = s$, and in i -th stage, it goes to the closest vertex to v_i that was not visited yet.

1. (5 PTS.) Show an example that prove that the greedy traveling salesman does not provide any constant factor approximation to the TSP.

Formally, for any constant $c > 0$, provide a complete graph G and positive weights on its edges, such that the length of the greedy TSP is by a factor of (at least) c longer than the length of the shortest TSP of G .

2. (5 PTS.) Show an example, that prove that the greedy traveling salesman does not provide any constant factor approximation to the TSP with *triangle inequality*.

Formally, for any constant $c > 0$, provide a complete graph G , and positive weights on its edges, such that the weights obey the triangle inequality, and the length of the greedy TSP is by a factor of (at least) c longer than the length of the shortest TSP of G . (In particular, *prove* that the triangle inequality holds for the weights you assign to the edges of G .)

5 YES. GREEDY ALGORITHM DOES NOT WORK FOR COLORING. REALLY.
(10 PTS.)

Let G be a graph defined over n vertices, and let the vertices be ordered: v_1, \dots, v_n . Let G_i be the induced subgraph of G on v_1, \dots, v_i . Formally, $G_i = (V_i, E_i)$, where $V_i = \{v_1, \dots, v_i\}$ and

$$E_i = \left\{ uv \in E \mid u, v \in V_i \text{ and } uv \in E(G) \right\}.$$

The greedy coloring algorithm, colors the vertices, one by one, according to their ordering. Let k_i denote the number of colors the algorithm uses to color the first i vertices.

In the i -th iteration, the algorithm considers v_i in the graph G_i . If all the neighbors of v_i in G_i are using all the k_{i-1} colors used to color G_{i-1} , the algorithm introduces a new color (i.e., $k_i = k_{i-1} + 1$) and assigns it to v_i . Otherwise, it assign v_i one of the colors $1, \dots, k_{i-1}$ (i.e., $k_i = k_{i-1}$).

Give an example of a graph G with n vertices, and an ordering of its vertices, such that even if G can be colored using $O(1)$ (in fact, it is possible to do this with two) colors, the greedy algorithm would color it with $\Omega(n)$ colors. (Hint: consider an ordering where the first two vertices are not connected.)

6 GREEDY COLORING DOES NOT WORK EVEN IF YOU DO IT IN THE RIGHT ORDER.
(10 PTS.)

Given a graph G , with n vertices, let us define an ordering on the vertices of G where the min degree vertex in the graph is last. Formally, we set v_n to be a vertex of minimum degree in G (breaking ties arbitrarily), define the ordering recursively, over the graph $G \setminus v_n$, which is the graph resulting from removing v_n from G . Let v_1, \dots, v_n be the resulting ordering, which is known as MIN LAST ORDERING.

1. (5 PTS.) Prove that the greedy coloring algorithm, if applied to a planar graph G , which uses the min last ordering, outputs a coloring that uses 6 colors.^①
2. (5 PTS.) Give an example of a graph G_n with $O(n)$ vertices which is 3-colorable, but nevertheless, when colored by the greedy algorithm using min last ordering, the number of colors output is n . (Hint: Extend your solution to 5.)

^①There is a quadratic time algorithm for coloring planar graphs using 4 colors (i.e., follows from a constructive proof of the four color theorem). Coloring with 5 colors requires slightly more cleverness.

Practice Problems

1 (10 PTS.) EVEN MORE ON VERTEX COVER

(Based on CLRS 35.1-1 and 35.1-4)

1. (3 PTS.) Give an example of a graph for which APPROX-VERTEX-COVER always yields a suboptimal solution.
2. (2 PTS.) Give an efficient algorithm that finds an optimal vertex cover for a tree in linear time.
3. (5 PTS.) (Based on CLRS 35.1-3)

Professor Nixon proposes the following heuristic to solve the vertex-cover problem. Repeatedly select a vertex of highest degree, and remove all of its incident edges. Give an example to show that the professor's heuristic does not have an approximation ratio of 2. [Hint: Try a bipartite graph with vertices of uniform degree on the left and vertices of varying degree on the right.]

2 (10 PTS.) GREEDY TRAVELING

(Based on CLRS 35.2-3)

Consider the following *closest-point heuristic* for building an approximate traveling-salesman tour. Begin with a trivial cycle consisting of a single arbitrary chosen vertex. At each step, identify the vertex u that is not on the cycle but whose distance to any vertex on the cycle is minimum. Suppose that the vertex on the cycle that is nearest u is vertex v . Extend the cycle to include u by inserting u just after v . Repeat until all vertices are on the cycle. Prove that this heuristic returns a tour whose total cost is not more than twice the cost of an optimal tour.

3 (10 PTS.) BIN PACKING

(Based on CLRS35-1)

Suppose that we are given a set of n objects, where the size s_i of the i -th object satisfies $0 < s_i < 1$. We wish to pack all the objects into the minimum number of unit-size bins. Each bin can hold any subset of the objects whose total size does not exceed 1.

1. (4 PTS.) Prove that the problem of determining the minimum number of bins required is NP-hard.
2. (6 PTS.) Give a heuristic that has an approximation ratio of 2. And give an $O(n \log n)$ time algorithm for the heuristic.

4 (10 PTS.) MAXIMUM CLIQUE (Based on CLRS 35-2)

Let $G = (V, E)$ be an undirected graph. For any $k \geq 1$, define $G^{(k)}$ to be the undirected graph $(V^{(k)}, E^{(k)})$, where $V^{(k)}$ is the set of all ordered k -tuples of vertices from V and $E^{(k)}$ is defined so that (v_1, v_2, \dots, v_k) is adjacent to (w_1, w_2, \dots, w_k) if and only if for each i ($1 \leq i \leq k$) either vertex v_i is adjacent to w_i in G , or else $v_i = w_i$.

1. (5 PTS.) Prove that the size of the maximum clique in $G^{(k)}$ is equal to the k -th power of the size of the maximum clique in G .
2. (5 PTS.) Argue that if there is an approximation algorithm that has a constant approximation ratio for finding a maximum-size clique, then there is a fully polynomial time approximation scheme for the problem.

5 (10 PTS.) APPROX PARTITION

Approx Partition

Instance: A finite set A and a “size” $s(a)$ for each $a \in A$, an approximation parameter $\epsilon > 0$.

Question: Is there a subset $A' \subseteq A$ such that

$$\left| \sum_{a \in A'} s(a) - \sum_{a \in A \setminus A'} s(a) \right| < \epsilon \sum_{a \in A} s(a)?$$

1. (5 PTS.) Suppose $s(a) \in \mathbb{Z}^+$ and $s(a) \leq k$ for each $a \in A$. Give an $O(\frac{nk}{\epsilon})$ time algorithm to find a ϵ -partition.
2. (5 PTS.) Suppose $s(a) \in \mathbb{R}^+$. Give an polynomial time algorithm to find a ϵ -partition.

6 (10 PTS.) JUST A LITTLE BIT MORE ABOUT GRAPH COLORING

1. (2 PTS.) Prove that a graph G with a chromatic number k (i.e., k is the minimal number of colors needed to color G), must have $\Omega(k^2)$ edges.
2. (2 PTS.) Prove that a graph G with m edges can be colored using $4\sqrt{m}$ colors.
3. (6 PTS.) Describe a polynomial time algorithm that given a graph G , which is 3-colorable, it computes a coloring of G using, say, at most $n/(5 \log n)$ colors.

7 (10 PTS.) SPLITTING AND SPLICING

Let $G = (V, E)$ be a graph with n vertices and m edges. A *splitting* of G is a partition of V into two sets V_1, V_2 , such that $V = V_1 \cup V_2$, and $V_1 \cap V_2 = \emptyset$. The cardinality of the split (V_1, V_2) , denoted by $m(V_1, V_2)$, is the number of edges in G that has one vertex in V_1 , and one vertex in V_2 . Namely,

$$m(V_1, V_2) = \left| \left\{ e \mid e = \{uv\} \in E(G), u \in V_1, v \in V_2 \right\} \right|.$$

Let $\text{sn}(G) = \max_{V_1} m(V_1, V_2)$ be the maximum cardinality of such a split. Describe a deterministic polynomial time algorithm that computes a splitting (V_1, V_2) of G , such that $m(V_1, V_2) \geq \text{sn}(G)/2$. (Hint: Start from an arbitrary split, and continue in a greedy fashion to improve it.)

8 VERTEX COVER

VERTEX COVER

Instance: A graph $G = (V, E)$ and a positive integer $K \leq |V|$.

Question: Is there a *vertex cover* of size K or less for G , that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and for each edge $\{u, v\} \in E$, at least one of u and v belongs to V' ?

1. Show that VERTEX COVER is NP-Complete. Hint: Do a reduction from INDEPENDENT SET to VERTEX COVER.
2. Show a polynomial approximation algorithm to the VERTEX-COVER problem which is a factor 2 approximation of the optimal solution. Namely, your algorithm should output a set $X \subseteq V$, such that X is a vertex cover, and $|X| \leq 2K_{opt}$, where K_{opt} is the cardinality of the smallest vertex cover of G .^②

^②It was very recently shown (I. Dinur and S. Safra. On the importance of being biased. Manuscript. <http://www.math.ias.edu/~iritd/mypapers/vc.pdf>, 2001.) that doing better than 1.3600 approximation to VERTEX COVER is NP-Hard. In your free time you can try and improve this constant. Good luck.

3. Present a linear time algorithm that solves this problem for the case that G is a tree.
4. For a constant k , a graph G is k -separable, if there are k vertices of G , such that if we remove them from G , each one of the remaining connected components has at most $(2/3)n$ vertices, and furthermore each one of those connected components is also k -separable. (More formally, a graph $G = (V, E)$ is k -separable, if for any subset of vertices $S \subseteq V$, there exists a subset $M \subseteq S$, such that each connected component of $G_{S \setminus M}$ has at most $(2/3)|S|$ vertices, and $|M| \leq k$.)

Show that given a graph G which is k -separable, one can compute the optimal VERTEX COVER in $n^{O(k)}$ time.

9 BIN PACKING

BIN PACKING

Instance: Finite set U of items, a size $s(u) \in \mathbb{Z}^+$ for each $u \in U$, an integer bin capacity B , and a positive integer K .

Question: Is there a partition of U into disjoint sets U_1, \dots, U_K such that the sum of the sizes of the items inside each U_i is B or less?

1. Show that the BIN PACKING problem is NP-Complete
2. In the optimization variant of BIN PACKING one has to find the minimum number of bins needed to contain all elements of U . Present an algorithm that is a factor two approximation to optimal solution. Namely, it outputs a partition of U into M bins, such that the total size of each bin is at most B , and $M \leq k_{opt}$, where k_{opt} is the minimum number of bins of size B needed to store all the given elements of U .
3. Assume that B is bounded by an integer constant m . Describe a polynomial algorithm that computes the solution that uses the minimum number of bins to store all the elements.
4. Show that the following problem is NP-Complete.

TILING

Instance: Finite set \mathcal{R} of rectangles and a rectangle R in the plane.

Question: Is there a way of placing the rectangles of \mathcal{R} inside R , so that no pair of the rectangles intersect, and all the rectangles have their edges parallel of the edges of R ?

5. Assume that \mathcal{R} is a set of squares that can be arranged as to tile R completely. Present a polynomial time algorithm that computes a subset $\mathcal{T} \subseteq \mathcal{R}$, and a tiling of \mathcal{T} , so that this tiling of \mathcal{T} covers, say, 10% of the area of R .

10 MINIMUM SET COVER

MINIMUM SET COVER

Instance: Collection C of subsets of a finite set S and an integer k .

Question: Are there k sets S_1, \dots, S_k in C such that $S \subseteq \cup_{i=1}^k S_i$?

1. Prove that MINIMUM SET COVER problem is NP-Complete

2. The greedy approximation algorithm for MINIMUM SET COVER, works by taking the largest set in $X \in C$, remove all the elements of X from S and also from each subset of C . The algorithm repeat this until all the elements of S are removed. Prove that the number of elements not covered after k_{opt} iterations is at most $n/2$, where k_{opt} is the smallest number of sets of C needed to cover S , and $n = |S|$.
3. Prove the greedy algorithm is $O(\log n)$ factor optimal approximation.
4. Prove that the following problem is NP-Complete.

HITTING SET

Instance: A collection C of subsets of a set S , a positive integer K .

Question: Does S contain a *hitting set* for C of size K or less, that is, a subset $S' \subseteq S$ with $|S'| \leq K$ and such that S' contains at least one element from each subset in C .

5. Given a set \mathcal{I} of n intervals on the real line, show a $O(n \log n)$ time algorithm that computes the smallest set of points X on the real line, such that for every interval $I \in \mathcal{I}$ there is a point $p \in X$, such that $p \in I$.

11 k -CENTER Problem

k -CENTER

Instance: A set P of n points in the plane, and an integer k and a radius r .

Question: Is there a cover of the points of P by k disks of radius (at most) r ?

- 11.A. Describe an $n^{O(k)}$ time algorithm that solves this problem.
- 11.B. There is a very simple and natural algorithm that achieves a 2-approximation for this cover: First it select an arbitrary point as a center (this point is going to be the center of one of the k covering disks). Then it computes the point that it furthest away from the current set of centers as the next center, and it continue in this fashion till it has k -points, which are the resulting centers. The smallest k equal radius disks centered at those points are the required k disks. Show an implementation of this approximation algorithm in $O(nk)$ time.
- 11.C. Prove that that the above algorithm is a factor two approximation to the optimal cover. Namely, the radius of the disks output $\leq 2r_{opt}$, where r_{opt} is the smallest radius, so that we can find k -disks that cover the point-set.
- 11.D. Provide an ε -approximation algorithm for this problem. Namely, given k and a set of points P in the plane, your algorithm would output k -disks that cover the points and their radius is $\leq (1 + \varepsilon)r_{opt}$, where r_{opt} is the minimum radius of such a cover of P .
- 11.E. Prove that dual problem r -DISK-COVER problem is NP-Hard. In this problem, given P and a radius r , one should find the smallest number of disks of radius r that cover P .
- 11.F. Describe an approximation algorithm to the r -DISK COVER problem. Namely, given a point-set P and a radius r , outputs k disks, so that the k disks cover P and are of radius r , and $k = O(k_{opt})$, where k_{opt} is the minimal number of disks needed to cover P by disks of radius r .

12 MAX 3SAT

MAX SAT

Instance: Set U of variables, a collection C of disjunctive clauses of literals where a literal is a variable or a negated variable in U .

Question: Find an assignment that maximized the number of clauses of C that are being satisfied.

- 12.A. Prove that MAX SAT is NP-Hard.
- 12.B. Prove that if each clause has exactly three literals, and we randomly assign to the variables values 0 or 1, then the expected number of satisfied clauses is $(7/8)M$, where $M = |C|$.
- 12.C. Show that for any instance of MAX SAT, where each clause has exactly three different literals, there exists an assignment that satisfies at least $7/8$ of the clauses.
- 12.D. Let (U, C) be an instance of MAX SAT such that each clause has $\geq 10 \cdot \log n$ distinct variables, where n is the number of clauses. Prove that there exists a satisfying assignment. Namely, there exists an assignment that satisfy all the clauses of C .

50.1.4. Homework 3

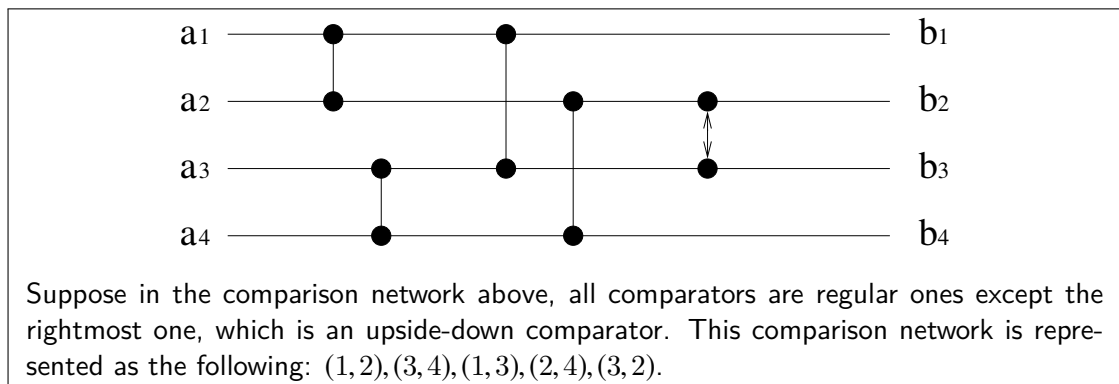
Required Problems

1 UPSIDE DOWN (10 PTS.)

Suppose that in addition to the standard kind of comparator, we introduce an "upside-down" comparator that produces its minimum output on the bottom wire and its maximum output on the top wire.

- (6 PTS.) An n -input sorting network with m comparators, is represented by a list of m pairs of integers in the range from 1 to n . Thus, a comparator between the wire i and j is represented as (i, j) . If $i < j$ then this is a regular comparator, and if $i > j$ it is an upside-down comparator.

Describe an algorithm that converts a sorting network with n inputs, c upside-down gates and m overall gates into an equivalent (i.e., with the same number of gates) sorting network that uses only regular gates. How fast is your algorithm?



- (4 PTS.) *Prove* that your algorithm is correct (i.e., it indeed outputs a network that uses only regular comparators, it always terminate, and the output network is equivalent to the input network).

2 MERGE THOSE SEQUENCES (20 PTS.)

- (10 PTS.) Consider a merging network with inputs a_1, a_2, \dots, a_n , for n an exact power of 2, in which the two monotonic sequences to be merged are $\langle a_1, a_3, \dots, a_{n-1} \rangle$ and $\langle a_2, a_4, \dots, a_n \rangle$ (namely, the input is a sequence of n numbers, where the odd numbers are sorted, and the even numbers are sorted). Prove that the number of comparators in this kind of merging network is $\Omega(n \log n)$. Why is this an interesting lower bound? (Hint: Partition the comparators into three sets.)
- (10 PTS.) Prove that any merging network, regardless of the order of inputs, requires $\Omega(n \log n)$ comparators. (Hint: Use question 1.)

3 PERMUTATIONS. (20 PTS.)

A *permutation network* on n inputs and n outputs has switches that allow it to connect its inputs to its outputs according to any $n!$ possible permutations. Figure 50.3 shows the 2-input, 2-output permutation network P_2 , which consists of a single switch that can be set either to feed its inputs straight through to its outputs or to cross them.

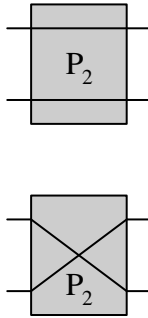


Figure 50.3: Permutation Switch

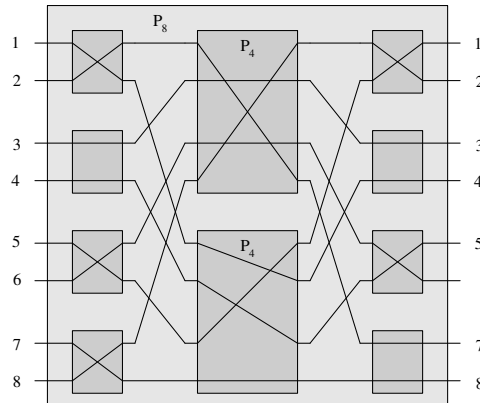


Figure 50.4: Permutation Network of Size 8

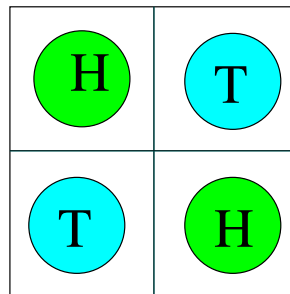
- (5 PTS.) Argue that if we replace each comparator in a sorting network with the switch of Figure 50.3, the resulting network is a permutation network. That is, for any permutation π , there is a way to set the switches in the network so that input i is connected to output $\pi(i)$.
- (2 PTS.) Figure 50.4 shows the recursive construction of an 8-input, 8-output permutation network P_8 that uses two copies of P_4 and 8 switches. The switches have been set to realize the permutation $\pi = \langle \pi(1), \pi(2), \dots, \pi(8) \rangle = \langle 4, 7, 3, 5, 1, 6, 8, 2 \rangle$, which requires (recursively) that the top P_4 realize $\langle 4, 2, 3, 1 \rangle$ and the bottom P_4 realize $\langle 2, 3, 1, 4 \rangle$.

Show how to realize the permutation $\langle 5, 3, 4, 6, 1, 8, 2, 7 \rangle$ on P_8 by drawing the switch settings and the permutations performed by the two P_4 's.

- (5 PTS.) Let n be an exact power of 2. Define P_n recursively in terms of two $P_{n/2}$'s in a manner similar to the way we defined P_8 . Describe an algorithm (ordinary random-access machine) that runs in $O(n)$ -time that sets the n switches connected to the inputs and outputs of P_n and specifies the permutations that must be realized by each $P_{n/2}$ in order to accomplish any given n -element permutation. Prove that your algorithm is correct.
- (3 PTS.) What are the depth size of P_n ? How long does it take on an ordinary random-access machine to compute all switch settings, including those within the $P_{n/2}$'s?
- (5 PTS.) Argue that for $n > 2$, any permutation network (not just P_n) must realize some permutation by two distinct combinations of switch settings.

4 THE HARD LIFE OF A JOURNALIST
(20 PTS.)

- (2 PTS.) A journalist, named Jane Austen, travels to Afghanistan, and unfortunately falls into the hands of Bin Laden. Bin Laden offer Jane a game for her life – if she wins she can leave. The game board is made out of 2×2 coins:



At each round, Jane can decide to flip one or two coins, by specifying which coins she is flipping (for example, flip the left bottom coin, and the right top coin), next Bin Laden goes and rotates the board by either 90, 180, 270, or 0 degrees (of course, rotation by 0 degrees is just keeping the coins in their current configuration).

The game is over when all the four coins are either all heads or all tails. To make things interesting, Jane does not see the board, and does not know the starting configuration.

Describe an algorithm that Jane can deploy, so that she always win. How many rounds are required by your algorithm?

- (5 PTS.) After escaping from Bin Laden, and on her way to Kabul, Jane meets a peace loving, nuclear reactor selling, French diplomat. The French diplomat is outraged to hear that Jane prefers Hummus to French Fries, and instruct his bodyguards to arrest Jane immediately, accusing her of being a quisling of the French cuisine (Jane has French citizenship). Again, the diplomat offers her a game for her life, similar to the Bin Laden game, with the following twist: after Jane flips her coins, the diplomat will reorder the coins in an arbitrary order (without flipping any coin). Describe an algorithm that Jane can use to win the game. What is the expected number of rounds Jane has to play before winning (the lower your bound, the better).
- (5 PTS.) After escaping from the French diplomat, Jane travels to Hanoi to investigate rumors that the priests in charge of the Towers of Hanoi games, are spending all the money they get on buying computer games and playing them, instead of playing the holy game of Towers of Hanoi, as they are suppose to do. However, the head priest is willing to do an interview with Jane, only if she plays the coin game (using the French diplomat version), with n coins. Describe an algorithm that guarantees that Jane wins. Provide an upper bound (as tight as possible) on the number of rounds Jane has to play before winning. (Providing an exact bound here is probably hard. As such, a rough upper bound would be acceptable.)
- (5 PTS.) Jane, tired of all those coin games, goes to Nashville for a vacation. Unfortunately for her, she is kidnapped by an Elvis lookalike. Not surprisingly, he offers her to play the coin game for her life, with the following variants: There are n coins, and at each round Jane can choose which of the n coins she wants to flip. Before flipping the coin, the Elvis lookalike tells her whether the coin is currently head or tail, and Jane can decide whether she wants to flip this coin or not. After each round, the Elvis lookalike takes the coins and reorder them in any order he likes. Describe an algorithm that guarantees that Jane wins. Provide an *exact* bound on the expected number of rounds that Jane has to play before she wins. (The smaller your bound, the better.)

5 THE HARD HARD LIFE OF THE IRS.
(10 PTS.)

The IRS receives, every year, n forms with personal tax returns. The IRS, of course, can not verify all n forms, but they can check some of them. Describe an algorithm, as fast as possible, that decides whether the number of incorrect tax forms is larger than εn , where ε is a prespecified constant between 0 and 1.

The decision of the algorithm is considered to be incorrect if it declares that the number of incorrect forms is smaller than εn , but it is in fact larger than $2\varepsilon n$. Similarly, the algorithm is considered to be incorrect if it claims that the number of incorrect forms is larger than $2\varepsilon n$, where it is in fact smaller than εn . (Namely, if the number of incorrect forms is between εn and $2\varepsilon n$, any of the two answers are acceptable.)

Your algorithm should output a correct result with probability $\geq 1 - 1/n^{10}$. What is the running time of your algorithm, assuming that verifying the correctness of a single tax form takes $O(1)$ time? (Hint: Use the Chernoff inequalities.)

6 CLOSEST NUMBERS

Let P be a set of n real numbers. The purpose of this exercise is to develop a linear time algorithm for deciding whether there are two equal numbers in P . Let x_1, \dots, x_n be a random permutation of the numbers in P .

1. (5 PTS.) Let $\pi_i = \min_{1 \leq k < j \leq i} |x_k - x_j|$ be the distance between the closest pair of numbers in x_1, \dots, x_i . Prove that $\mathbb{P}[\pi_i \neq \pi_{i-1}] \leq 2/i$.
2. (5 PTS.) Given a parameter r , describe an algorithm, that decides in $O(i)$ time, whether $\pi_i < r$. Furthermore, if $\pi_{i-1} = r$ but $\pi_i < r$, then it computes π_i . (Hint: use hashing and the floor function.)
3. (5 PTS.) Show how to modify the previous algorithm into a data-structure, so that after computing π_i , one can insert x_{i+1}, \dots, x_j into the data-structure in $O(1)$ time per element, where $\pi_{i+1} = \pi_{i+2} = \dots = \pi_{j-1} > \pi_j$. And furthermore, the data-structure returns π_j .
4. (5 PTS.) Describe an algorithm, with $O(n)$ expected running time, that computes π_n . Clearly, if $\pi_n = 0$ then there are two identical numbers in P . (Hint: Use (a) and (c).)

(Note, that the algorithm of (d) is faster than one can achieve in the comparison model (i.e., we only only to compare numbers). One can prove that the fastest algorithm for this problem in the comparison model requires $\Omega(n \log n)$ time. Namely, the only way to solve it is using sorting.)

Practice Problems

Sorting Networks

1. (10 PTS.) (Based on CLRS 27.1-2 and 27.1-4, or CLR 28.1-2 and 28.1-4)
 - (a) (5 PTS.) Let n be an exact power of 2. Show how to construct an n -input, n -output comparison network of depth $\lg n$ in which the top output wire always carries the minimum input value and the bottom output wire always carries the maximum input value.
 - (b) (5 PTS.) Prove that any sorting network on n inputs has depth at least $\lg n$.

2. (10 PTS.) (Based on CLRS 27.2-5 or CLR 28.2-5)

Prove that an n -input sorting network must contain at least one comparator between the i th and $(i + 1)$ st lines for all $i = 1, 2, \dots, n - 1$.

3. (10 PTS.) (Based on CLRS 27.5-1 and 27.5-2, or CLR 28.5-1 and 28.5-2)

The sorting network SORTER[n] was present in class (it is also shown in CLRS Figure 27.12 or CLR Figure 28.12), where n is an exact power of 2. Answer the following questions about SORTER[n].

 - (a) (5 PTS.) Give a tight bound for the number of comparators in SORTER[n].

(b) (5 PTS.) Show that the depth of $\text{SORTER}[n]$ is exactly $(\lg n)(\lg n + 1)/2$.

4. (10 PTS.) (Based on CLRS 27.5-3 or CLR 28.5-3)

Suppose that we have $2n$ elements $\langle a_1, a_2, \dots, a_{2n} \rangle$ and wish to partition them into the n smallest and the n largest. Prove that we can do this in constant additional depth after separately sorting $\langle a_1, a_2, \dots, a_n \rangle$ and $\langle a_{n+1}, a_{n+2}, \dots, a_{2n} \rangle$.

5. (10 PTS.) (Based on CLRS 27.5-4 or CLR 28.5-4)

Let $S(k)$ be the depth of a sorting network with k inputs, and let $M(k)$ be the depth of a merging network with $2k$ inputs. Suppose that we have a sequence of n numbers to be sorted and we know that every number is within k positions of its correct position in the sorted order, which means that we need to move each number at most $(k - 1)$ positions to sort the inputs. For example, in the sequence 3 2 1 4 5 8 7 6 9, every number is within 3 positions of its correct position. But in sequence 3 2 1 4 5 9 8 7 6, the number 9 and 6 are outside 3 positions of its correct position.

Show that we can sort the n numbers in depth $S(k) + 2M(k)$. (You need to prove your answer is correct.)

6. (20 PTS.) (Based on CLRS 27.5-5 or CLR 28.5-5)

We can sort the entries of an $m \times m$ matrix by repeating the following procedure k times:

- (a) Sort each odd-numbered row into monotonically increasing order.
- (b) Sort each even-numbered row into monotonically decreasing order.
- (c) Sort each column into monotonically increasing order.

(a) (8 PTS.) Suppose the matrix contains only 0's and 1's. We repeat the above procedure again and again until no changes occur. In what order should we read the matrix to obtain the sorted output ($m \times m$ numbers in increasing order)? Prove that any $m \times m$ matrix of 0's and 1's will be finally sorted.

(b) (8 PTS.) Prove that by repeating the above procedure, any matrix of real numbers can be sorted. [Hint: Refer to the proof of the zero-one principle.]

(c) (4 PTS.) Suppose k iterations are required for this procedure to sort the $m \times m$ numbers. Give an upper bound for k . The tighter your upper bound the better (prove your bound).

Randomized Algorithms

1 TAIL INEQUALITIES (10 PTS.)

1. Prove the following theorem:

Theorem 50.1.1. Let X_1, X_2, \dots, X_n be independent coin flips such that for $1 \leq i \leq n$, we have $\mathbb{P}[X_i = 1] = p_i$, where $0 < p_i < 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = E[X] = \sum_i p_i$ and for any $\delta > 0$,

$$\mathbb{P}[X > (1 + \delta)\mu] < \left[\frac{e^\delta}{(1 + \delta)^{(1 + \delta)}} \right]^\mu,$$

and

$$\mathbb{P}[X < (1 - \delta)\mu] < \exp\left(\frac{-\mu\delta^2}{2}\right).$$

2. Consider a collection of n random variables X_i drawn independently from the *geometric distribution* with mean 2 – that is, X_i is the number of flips of an unbiased coin up to and including the first head. Let $X = \sum X_i$. Derive an upper bound as small as possible on the probability that $X > (1 + \delta)(2n)$ for any fixed δ .

2 TOURNAMENT WITHOUT A WINNER
(10 PTS.)

Consider a *tournament* on n teams, in which each pair of teams play against each other once and a winner is always declared. Suppose we try to rank the teams in some total order based on the outcome of the tournament. Say that a game *agrees* with the ranking we have chosen if the team we ranked better won. Prove that for sufficiently large n , there is a possible set of outcomes such that no ranking agrees with more than 51% of the games. (Hint: Pick the winner in a game randomly and use the results of exercise 1 above.)

3 FUZZY SORTING OF INTERVALS (Based on CLRS 7-6)
(10 PTS.)

Consider a sorting problem in which the numbers are not known exactly. Instead, for each number, we know an interval on the real line to which it belongs. That is, we are given n closed intervals of the form $[a_i, b_i]$, where $a_i \leq b_i$. The goal is to *fuzzy-sort* these intervals, i.e., produce a permutation $\langle i_1, i_2, \dots, i_n \rangle$ of the intervals such that there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \dots \leq c_n$.

1. (5 PTS.) Design an algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksort the left endpoints (the a_i 's), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals gets easier and easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.)
2. (5 PTS.) Argue that your algorithm runs in expected time $\Theta(n \lg n)$ in general, but runs in expected time $\Theta(n)$ when all of the intervals overlap (i.e., when there exists a value x such that $x \in [a_i, b_i]$ for all i). Your algorithm should not be checking for this case explicitly; rather, its performance should naturally improve as the amount of overlap increases.

4 APPROX MAX CUT
(5 PTS.)

Given a graph $G = (V, E)$ with n vertices and m edges, describe an algorithm that runs in $O(n)$ times, and output a cut $S \subseteq V$, such that the expected number of edges in the cut is $\geq M/2$, where M is the number of edges in the maximum cut, where the number of edges in the cut is $|(S \times (V \setminus S)) \cap E|$.

5 MODIFIED PARTITION (Based on CLRS 7.4-6)
(5 PTS.)

Consider modifying the PARTITION procedure by randomly picking three elements from array A and partitioning about their median. Approximate the probability of getting at worst an α -to- $(1 - \alpha)$ split, as a function of α in the range $0 < \alpha < 1$.

6 SORTING RANDOM INPUT
(10 PTS.)

Let a_1, \dots, a_n be n real numbers chosen independently and uniformly from the range $[0, 1]$.

- (5 PTS.) Describe an algorithm with an expected linear running time that sorts the numbers.
- (5 PTS.) Show that the linear running time is with high probability.

7 ESTIMATING QUANTITIES
(10 PTS.)

1. (5 PTS.) Assume that you are given a function `RandBit` that returns a truly random bit. However, you do not know what is the probability p that `RandBit` returns 1. Describe an algorithm (and prove its correctness), as fast as possible, that receives parameters ϵ, δ , and outputs a number x , such that

with probability $\geq 1 - \delta$ we have $p \leq x \leq p + \varepsilon$. Namely, the program estimates the value of p “reliably”.

- (5 PTS.) Let $G = (V, E)$ be a graph with n vertices and m edges. Assume that the only way you can know whether there is an edge between vertices u and v is to probe the graph G and ask whether there is an edge $uv \in E$ in constant time. You are given parameters $\varepsilon > 0$ and $\delta > 0$. Describe an algorithm, *as fast as possible*, that output a number k which is a good estimate of the number of edges of G . Namely, such that $m \leq k \leq m + \varepsilon n^2$ with probability larger than $1 - \delta$.

8 YEH, WHATEVER.

Provide a sub-quadratic ($o(n^2)$ time) deterministic algorithm for the nuts and bolts matching problem. Your solution should be self contained.

9 RANDOM BITS IN A TREAP

Let’s analyze the number of random bits needed to implement the operations of a treap. Suppose we pick a priority p_i at random from the unit interval. Then the binary representation of each p_i can be generated as a potentially infinite series of bits that are the outcome of unbiased coin flips. The idea is to generate only as many bits in this sequence as is necessary for resolving comparisons between different priorities. Suppose we have only generated some prefixes of the binary representations of the priorities of the elements in the treap T . Now, while inserting an item y , we compare its priority p_y to other’s priorities to determine how y should be rotated. While comparing p_y to some p_i , if their current partial binary representation can resolve the comparison, then we are done. Otherwise, they have the same partial binary representations (upto the length of the shorter of the two) and we keep generating more bits for each until they first differ.

- Compute a tight upper bound on the expected number of coin flips or random bits needed for a single priority comparison. (Note that during insertion every time we decide whether or not to perform a rotation, we perform a priority comparison. We are interested in the number of bits generated in such a single comparison.)
- Generating bits one at a time like this is probably a bad idea in practice. Give a more practical scheme that generates the priorities in advance, using a small number of random bits, given an upper bound n on the treap size. Describe a scheme that works correctly with probability $\geq 1 - n^{-c}$, where c is a prespecified constant.

10 MAJORITY TREE

Consider a uniform rooted tree of height h (every leaf is at distance h from the root). The root, as well as any internal node, has 3 children. Each leaf has a boolean value associated with it. Each internal node returns the value returned by the majority of its children. The evaluation problem consists of determining the value of the root; at each step, an algorithm can choose one leaf whose value it wishes to

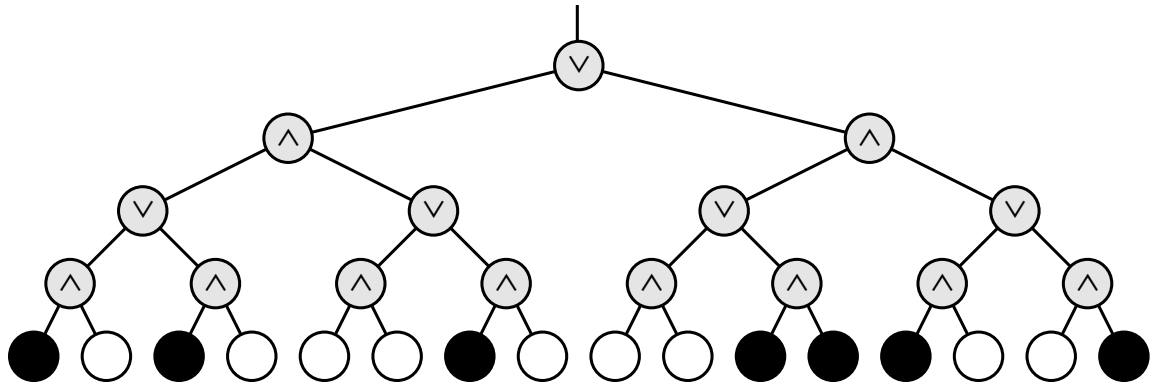
- [5 points] Describe a deterministic algorithm that runs in $O(n)$ time, that computes the value of the tree, where $n = 3^h$.
- [10 points] Consider the recursive randomized algorithm that evaluates two subtrees of the root chosen at random. If the values returned disagree, it proceeds to evaluate the third sub-tree. Show the expected number of leaves read by the algorithm on any instance is at most $n^{0.9}$.

(HARD)

- [5 points] Show that for any deterministic algorithm, there is an instance (a set of boolean values for the leaves) that forces it to read all $n = 3^h$ leaves. (hint: Consider an adversary argument, where you provide the algorithm with the minimal amount of information as it request bits from you. In particular, one can devise such an adversary algorithm.).

11 A GAME OF DEATH

Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are *and* gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

- 11.A. Describe and analyze a deterministic algorithm to determine whether or not you can win. [Hint: *This is easy!*]
- 11.B. Unfortunately, Death won't let you even look at every node in the tree. Describe a *randomized* algorithm that determines whether you can win in $\Theta(3^n)$ expected time. [Hint: *Consider the case $n = 1$.*]

50.1.5. Homework 4

CS 373: Algorithms, Spring 2003

Homework 4 (due Thursday, March 20, 2003 at 23:59:59)

Required Problems

1 SOME NUMBER THEORY. (10 PTS.)

- (5 PTS.) Prove that if $\gcd(m, n) = 1$, then $m^{\phi(n)} + n^{\phi(m)} \equiv 1 \pmod{mn}$.
- (5 PTS.) Give two distinct proofs that there are an infinite number of prime numbers.

2 EVEN MORE NUMBER THEORY

(10 PTS.)

Prove that $|P(n)| = \Omega(n^2)$, where $P(n) = \{(a, b) \mid a, b \in \mathbb{Z}, 0 < a < b \leq n, \gcd(a, b) = 1\}$.

3 YET ANOTHER NUMBER THEORY QUESTION

(20 PTS.)

1. (2 PTS.) Prove that the product of all primes p , for $m < p \leq 2m$ is at most $\binom{2m}{m}$.
2. (4 PTS.) **Using (a)**, prove that the number of all primes between m and $2m$ is $O(m/\ln m)$.
3. (3 PTS.) **Using (b)**, prove that the number of primes smaller than n is $O(n/\ln n)$.
4. (2 PTS.) Prove that if 2^k divides $\binom{2m}{m}$ then $2^k \leq 2m$.
5. (5 PTS.) (Hard) Prove that for a prime p , if p^k divides $\binom{2m}{m}$ then $p^k \leq 2m$.
6. (4 PTS.) **Using (e)**, prove that the number of primes between 1 and n is $\Omega(n/\ln n)$. (Hint: use the fact that $\binom{2m}{m} \geq 2^{2m}/(2m)$.)

4 AND NOW FOR SOMETHING COMPLETELY DIFFERENT.

(10 PTS.)

Prove that the following problems are NPC or provide a polynomial time algorithm to solve them:

1. Given a directed graph G , and two vertices $u, v \in V(G)$, find the maximum number of edge disjoint paths between u and v .
2. Given a directed graph G , and two vertices $u, v \in V(G)$, find the maximum number of *vertex* disjoint paths between u and v (the paths are disjoint in their vertices, except of course, for the vertices u and v).

5 MINIMUM CUT

(10 PTS.)

Present a *deterministic* algorithm, such that given an undirected graph G , it computes the minimum cut in G . How fast is your algorithm? How does your algorithm compare with the randomized algorithm shown in class?

6 INDEPENDENCE MATRIX

(10 PTS.)

Consider a 0 – 1 matrix H with n_1 rows and n_2 columns. We refer to a row or a column of the matrix H as a line. We say that a set of 1's in the matrix H is *independent* if no two of them appear in the same line. We also say that a set of lines in the matrix is a *cover* of H if they include (i.e., “cover”) all the 1's in the matrix. Using the max-flow min-cut theorem on an appropriately defined network, show that the maximum number of independent 1's equals the minimum number of lines in the cover.

Practice Problems

1 SCALAR FLOW PRODUCT (10 PTS.)

Let f be a flow in a network, and let α be a real number. The *scalar flow product*, denoted by αf , is a function from $V \times V$ to \mathbb{R} defined by

$$(\alpha f)(u, v) = \alpha \cdot f(u, v).$$

Prove that the flows in a network form a *convex set*. That is, show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha)f_2$ for all α in the range $0 \leq \alpha \leq 1$.

- 2** (Based on CLRS 26.1-9)
(10 PTS.)

Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining if both his children can go to the same school as a maximum-flow problem.

- 3** (Based on CLRS 26.2-8 and 26.2-10)
(10 PTS.)

1. (5 PTS.) Show that a maximum flow in a network $G = (V, E)$ can always be found by a sequence of at most $|E|$ augmenting paths. [Hint: Determine the paths after finding the maximum flow.]
2. (5 PTS.) Suppose that a flow network $G = (V, E)$ has symmetric edges, that is, $(u, v) \in E$ if and only if $(v, u) \in E$. Show that the Edmonds-Karp algorithm terminates after at most $|V||E|/4$ iterations. [Hint: For any edge (u, v) , consider how both $\delta(s, u)$ and $\delta(v, t)$ change between times at which (u, v) is critical.]

- 4** EDGE CONNECTIVITY
(10 PTS.) (Based on CLRS 26.2-9)

The *edge connectivity* of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how the edge connectivity of an undirected graph $G = (V, E)$ can be determined by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

- 5** PERFECT MATCHING
(20 PTS.) (Based on CLRS 26.3-4 and 26.3-5)

1. (10 PTS.) A *perfect matching* is a matching in which every vertex is matched. Let $G = (V, E)$ be an undirected bipartite graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the *neighborhood* of X as

$$N(X) = \left\{ y \in V \mid (x, y) \in E \text{ for some } x \in X \right\},$$

that is, the set of vertices adjacent to some member of X . Prove *Hall's theorem*: there exists a perfect matching in G if and only if $|A| \leq |N(A)|$ for every subset $A \subseteq L$.

2. (10 PTS.) We say that a bipartite graph $G = (V, E)$, where $V = L \cup R$, is *d-regular* if every vertex $v \in V$ has degree exactly d . Every d -regular bipartite graph has $|L| = |R|$. Prove that every d -regular bipartite graph has a matching of cardinality $|L|$ by arguing that a minimum cut of the corresponding flow network has capacity $|L|$.

- 6** MAXIMUM FLOW BY SCALING
(20 PTS.) (Based on CLRS 26-5)

Let $G = (V, E)$ be a flow network with source s , sink t , and an integer capacity $c(u, v)$ on each edge $(u, v) \in E$. Let $C = \max_{(u, v) \in E} c(u, v)$.

1. (2 PTS.) Argue that a minimum cut of G has capacity at most $C|E|$.

2. (5 PTS.) For a given number K , show that an augmenting path of capacity at least K can be found in $O(E)$ time, if such a path exists.

The following modification of FORD-FULKERSON-METHOD can be used to compute a maximum flow in G .

```

MAX-FLOW-BY-SCALING( $G, s, t$ )
1   $C \leftarrow \max_{(u,v) \in E} c(u,v)$ 
2  initialize flow  $f$  to 0
3   $K \leftarrow 2^{\lceil \lg C \rceil}$ 
4  while  $K \geq 1$  do {
5      while (there exists an augmenting path  $p$  of
              capacity at least  $K$ ) do {
6          augment flow  $f$  along  $p$ 
7      }
8       $K \leftarrow K/2$ 
9  }
10 return  $f$ 

```

3. (3 PTS.) Argue that MAX-FLOW-BY-SCALING returns a maximum flow.
4. (4 PTS.) Show that the capacity of a minimum cut of the residual graph G_f is at most $2K|E|$ each time line 4 is executed.
5. (4 PTS.) Argue that the inner **while** loop of lines 5-6 is executed $O(E)$ times for each value of K .
6. (2 PTS.) Conclude that MAX-FLOW-BY-SCALING can be implemented so that it runs in $O(E^2 \lg C)$ time.

7 THE HOPCROFT-KARP BIPARTITE MATCHING ALGORITHM
(20 PTS.) (Based on CLRS 26-7)

In this problem, we describe a faster algorithm, due to Hopcroft and Karp, for finding a maximum matching in a bipartite graph. The algorithm runs in $O(\sqrt{V}E)$ time. Given an undirected, bipartite graph $G = (V, E)$, where $V = L \cup R$ and all edges have exactly one endpoint in L , let M be a matching in G . We say that a simple path P in G is an *augmenting path* with respect to M if it starts at an unmatched vertex in L , ends at an unmatched vertex in R , and its edges belong alternatively to M and $E - M$. (This definition of an augmenting path is related to, but different from, an augmenting path in a flow network.) In this problem, we treat a path as a sequence of edges, rather than as a sequence of vertices. A shortest augmenting path with respect to a matching M is an augmenting path with a minimum number of edges. Given two sets A and B , the *symmetric difference* $A \oplus B$ is defined as $(A - B) \cup (B - A)$, that is, the elements that are in exactly one of the two sets.

- 7.A.** (4 PTS.) Show that if M is a matching and P is an augmenting path with respect to M , then the symmetric difference $M \oplus P$ is a matching and $|M \oplus P| = |M| + 1$. Show that if P_1, P_2, \dots, P_k are vertex-disjoint augmenting paths with respect to M , then the symmetric difference $M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ is a matching with cardinality $|M| + k$.

The general structure of our algorithm is the following:


```

HOPCROFT-KARP( $G$ )
1   $M \leftarrow \emptyset$ 
2  repeat
3      let  $P \leftarrow \{P_1, P_2, \dots, P_k\}$  be a maximum set of
          vertex-disjoint shortest augmenting paths
          with respect to  $M$ 
4       $M \leftarrow M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$ 
5  until  $P = \emptyset$ 

6  return  $M$ 

```

The remainder of this problem asks you to analyze the number of iterations in the algorithm (that is, the number of iterations in the **repeat** loop) and to describe an implementation of line 3.

- 7.B.** (4 PTS.) Given two matchings M and M^* in G , show that every vertex in the graph $G' = (V, M \oplus M^*)$ has degree at most 2. Conclude that G' is a disjoint union of simple paths or cycles. Argue that edges in each such simple path or cycle belong alternatively to M or M^* . Prove that if $|M| \leq |M^*|$, then $M \oplus M^*$ contains at least $|M^*| - |M|$ vertex-disjoint augmenting paths with respect to M .
- Let l be the length of a shortest augmenting path with respect to a matching M , and let P_1, P_2, \dots, P_k be a maximum set of vertex-disjoint augmenting paths of length l with respect to M . Let $M' = M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$, and suppose that P is a shortest augmenting path with respect to M' .
- 7.C.** (2 PTS.) Show that if P is vertex-disjoint from P_1, P_2, \dots, P_k , then P has more than l edges.
- 7.D.** (2 PTS.) Now suppose P is not vertex-disjoint from P_1, P_2, \dots, P_k . Let A be the set of edges $(M \oplus M') \oplus P$. Show that $A = (P_1 \cup P_2 \cup \dots \cup P_k) \oplus P$ and that $|A| \geq (k + 1)l$. Conclude that P has more than l edges.
- 7.E.** (2 PTS.) Prove that if a shortest augmenting path for M has length l , the size of the maximum matching is at most $|M| + |V|/l$.
- 7.F.** (2 PTS.) Show that the number of **repeat** loop iterations in the algorithm is at most $2\sqrt{V}$. [Hint: By how much can M grow after iteration number \sqrt{V} ?]
- 7.G.** (4 PTS.) Give an algorithm that runs in $O(E)$ time to find a maximum set of vertex-disjoint shortest augmenting paths P_1, P_2, \dots, P_k for a given matching M . Conclude that the total running time of HOPCROFT-KARP is $O(\sqrt{V}E)$.

50.1.6. Homework 5

CS 373: Algorithms, Spring 2003

Homework 5 (due Thursday, April 17, 2003 at 23:59:59)

Required Problems

1 HASHING TO VICTORY (20 PTS.)

In this question we will investigate the construction of hash table for a set W , where W is static, provided in advance, and we care only for search operations.

1. (2 PTS.) Let $U = \{1, \dots, m\}$, and $p = m + 1$ is a prime.
 Let $W \subseteq U$, such that $n = |W|$, and s an integer number larger than n . Let $g_k(x, s) = (kx \bmod p) \bmod s$.
 Let $\beta(k, j, s) = \left| \left\{ x \mid x \in W, g_k(x, s) = j \right\} \right|$. Prove that

$$\sum_{k=1}^{p-1} \sum_{j=1}^s \binom{\beta(k, j, s)}{2} < \frac{(p-1)n^2}{s}.$$

2. (2 PTS.) Prove that there exists $k \in U$, such that

$$\sum_{j=1}^s \binom{\beta(k, j, s)}{2} < \frac{n^2}{s}.$$

3. (2 PTS.) Prove that $\sum_{j=1}^n \beta(k, j, n) = |W| = n$.
 4. (3 PTS.) Prove that there exists a $k \in U$ such that $\sum_{j=1}^n (\beta(k, j, n))^2 < 3n$.
 5. (3 PTS.) Prove that there exists a $k' \in U$, such that the function $h(x) = (k'x \bmod p) \bmod n^2$ is one-to-one when restricted to W .
 6. (3 PTS.) Conclude, that one can construct a hash-table for W , of $O(n^2)$, such that there are no collisions, and a search operation can be performed in $O(1)$ time (note that the time here is worst case, also note that the construction time here is quite bad - ignore it).
 7. (3 PTS.) Using (d) and (f), conclude that one can build a two-level hash-table that uses $O(n)$ space, and perform a lookup operation in $O(1)$ time (worst case).

2 FIND k TH SMALLEST NUMBER.
 (20 PTS.)

This question asks you to design and analyze a *randomized incremental* algorithm to select the k th smallest element from a given set of n elements (from a universe with a linear order).

In an *incremental* algorithm, the input consists of a sequence of elements x_1, x_2, \dots, x_n . After any prefix x_1, \dots, x_{i-1} has been considered, the algorithm has computed the k th smallest element in x_1, \dots, x_{i-1} (which is undefined if $i \leq k$), or if appropriate, some other invariant from which the k th smallest element could be determined. This invariant is updated as the next element x_i is considered.

Any incremental algorithm can be *randomized* by first randomly permuting the input sequence, with each permutation equally likely.

- (5 PTS.) Describe an incremental algorithm for computing the k th smallest element.
- (5 PTS.) How many comparisons does your algorithm perform in the worst case?
- (10 PTS.) What is the expected number (over all permutations) of comparisons performed by the randomized version of your algorithm? (Hint: When considering x_i , what is the probability that x_i is smaller than the k th smallest so far?) You should aim for a bound of at most $n + O(k \log(n/k))$.
 Revise (a) if necessary in order to achieve this.

3 ANOTHER LOWER BOUND
 (20 PTS.)

Let $b_1 \leq b_2 \leq b_3 \leq \dots \leq b_k$ be k given sorted numbers, and let A be a set of n arbitrary numbers $A = \{a_1, \dots, a_n\}$, such that $b_1 < a_i < b_k$, for $i = 1, \dots, n$

The rank $v = r(a_i)$ of a_i is the index, such that $b_v < a_i < b_{v+1}$.

Prove, that in the comparison model, any algorithm that outputs the ranks $r(a_1), \dots, r(a_n)$ must take $\Omega(n \log k)$ running time in the worst case.

4 ACKERMANN FUNCTION

(20 PTS.)

The Ackermann's function $A_i(n)$ is defined as follows:

$$A_i(n) = \begin{cases} 4 & \text{if } n = 1 \\ 4n & \text{if } i = 1 \\ A_{i-1}(A_i(n-1)) & \text{otherwise} \end{cases}$$

Here we define $A(x) = A_x(x)$. And we define $\alpha(n)$ as a pseudo-inverse function of $A(x)$. That is, $\alpha(n)$ is the least x such that $n \leq A(x)$.

1. (4 PTS.) Give a precise description of what are the functions: $A_2(n)$, $A_3(n)$, and $A_4(n)$.
2. (4 PTS.) What is the number $A(4)$?
3. (4 PTS.) **Prove** that $\lim_{n \rightarrow \infty} \frac{\alpha(n)}{\log^*(n)} = 0$.
4. (4 PTS.) We define

$$\log^{**} n = \min \left\{ i \geq 1 \mid \underbrace{\log^* \dots \log^* n}_{i \text{ times}} \leq 2 \right\}$$

(i.e., how many times do you have to take \log^* of a number before you get a number smaller than

- 2). **Prove** that $\lim_{n \rightarrow \infty} \frac{\sqrt{\alpha(n)}}{\log^{**}(n)} = 0$.
5. (4 PTS.) **Prove** that $\log(\alpha(n)) \leq \alpha(\log^{**} n)$ for n large enough.

5 DIVIDE-AND-CONQUER MULTIPLICATION

(20 PTS.)

1. (7 PTS.) Show how to multiply two linear polynomials $ax + b$ and $cx + d$ using only three multiplications. (Hint: One of the multiplications is $(a + b) \cdot (c + d)$.)
2. (7 PTS.) Give two divide-and-conquer algorithms for multiplying two polynomials of degree-bound n that run in time $\Theta(n^{\lg 3})$. The first algorithm should divide the input polynomial coefficients into a high half and a low half, and the second algorithm should divide them according to whether their index is odd or even.
3. (6 PTS.) Show that two n -bit integers can be multiplied in $O(n^{\lg 3})$ steps, where each step operates on at most a constant number of 1-bit values.

Practice Problems

1 (10 PTS.)

1. (1 PTS.) With path compression and union by rank, during the lifetime of a Union-Find data-structure, how many elements would have rank equal to $\lfloor \lg n - 5 \rfloor$, where there are n elements stored in the data-structure?
2. (1 PTS.) Same question, for rank $\lfloor (\lg n)/2 \rfloor$.
3. (2 PTS.) Prove that in a set of n elements, a sequence of n consecutive FIND operations take $O(n)$ time in total.
4. (1 PTS.) (Based on CLRS 21.3-2)
Write a nonrecursive version of FIND with path compression.

5. (3 PTS.) Show that any sequence of m MAKESET, FIND, and UNION operations, where all the UNION operations appear before any of the FIND operations, takes only $O(m)$ time if both path compression and union by rank are used.
6. (2 PTS.) What happens in the same situation if only the path compression is used?

2 (10 PTS.) OFF-LINE MINIMUM
(Based on CLRS 21-1)

The *off-line minimum problem* asks us to maintain a dynamic set T of elements from the domain $\{1, 2, \dots, n\}$ under the operations INSERT and EXTRACT-MIN. We are given a sequence S of n INSERT and m EXTRACT-MIN calls, where each key in $\{1, 2, \dots, n\}$ is inserted exactly once. We wish to determine which key is returned by each EXTRACT-MIN call. Specifically, we wish to fill in an array $extracted[1 \dots m]$, where for $i = 1, 2, \dots, m$, $extracted[i]$ is the key returned by the i th EXTRACT-MIN call. The problem is “off-line” in the sense that we are allowed to process the entire sequence S before determining any of the returned keys.

1. (2 PTS.)
In the following instance of the off-line minimum problem, each INSERT is represented by a number and each EXTRACT-MIN is represented by the letter E:

4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5.

Fill in the correct values in the *extracted* array.

2. (4 PTS.)
To develop an algorithm for this problem, we break the sequence S into homogeneous subsequences. That is, we represent S by $I_1, E, I_2, E, I_3, \dots, I_m, E, I_{m+1}$, where each E represents a single EXTRACT-MIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j , we initially place the keys inserted by these operations into a set K_j , which is empty if I_j is empty. We then do the following.

```

OFF-LINE-MINIMUM( $m, n$ )
1  for  $i \leftarrow 1$  to  $n$ 
2      do determine  $j$  such that  $i \in K_j$ 
3          if  $j \neq m + 1$ 
4              then  $extracted[j] \leftarrow i$ 
5                  let  $l$  be the smallest value greater than  $j$  for which set  $K_l$  exists
6                   $K_l \leftarrow K_j \cup K_l$ , destroying  $K_j$ 
7  return  $extracted$ 

```

Argue that the array *extracted* returned by OFF-LINE-MINIMUM is correct.

3. (4 PTS.)
Describe how to implement OFF-LINE-MINIMUM efficiently with a disjoint-set data structure. Give a tight bound on the worst-case running time of your implementation.

3 (10 PTS.) TARJAN'S OFF-LINE LEAST-COMMON-ANCESTORS ALGORITHM
(Based on CLRS 21-3)

The *least common ancestor* of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T . In the *off-line least-common-ancestors problem*, we are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T , and we wish to determine the least common ancestor of each pair in P .

To solve the off-line least-common-ancestors problem, the following procedure performs a tree walk of T with the initial call $LCA(\text{root}[T])$. Each node is assumed to be colored WHITE prior to the walk.

```

LCA( $u$ )
1  MAKESET( $u$ )
2   $\text{ancestor}[\text{FIND}(u)] \leftarrow u$ 
3  for each child  $v$  of  $u$  in  $T$ 
4      do LCA( $v$ )
5          UNION( $u, v$ )
6           $\text{ancestor}[\text{FIND}(u)] \leftarrow u$ 
7   $\text{color}[u] \leftarrow \text{BLACK}$ 
8  for each node  $v$  such that  $\{u, v\} \in P$ 
9      do if  $\text{color}[v] = \text{BLACK}$ 
10         then print "The least common ancestor of"  $u$  "and"  $v$  "is"  $\text{ancestor}[\text{FIND}(v)]$ 

```

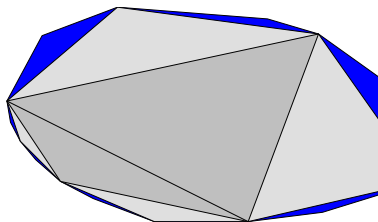
- 3.A. (2 PTS.) Argue that line 10 is executed exactly once for each pair $\{u, v\} \in P$.
- 3.B. (2 PTS.) Argue that at the time of the call $LCA(u)$, the number of sets in the disjoint-set data structure is equal to the depth of u in T .
- 3.C. (3 PTS.) Prove that LCA correctly prints the least common ancestor of u and v for each pair $\{u, v\} \in P$.
- 3.D. (3 PTS.) Analyze the running time of LCA , assuming that we use the implementation of the disjoint-set data structure with path compression and union by rank.

50.1.7. Homework 6

CS 373: Algorithms, Spring 2003 Homework 6

Problems

- 1 Given a convex polygon P , its balanced triangulation is created by recursively triangulating the convex polygon P' defined by its even vertices, and finally adding consecutive diagonals between even points. For example:



Alternative interpretation of this construction, is that we create a sequence of polygons where P_0 is the highest level polygon (a quadrangle in the above example), and P_i is the refinement of P_{i-1} till $P_{\lceil \log n \rceil} = P$.

1. Given a polygon P , show how to compute its balanced triangulation in linear time.

2. Let T be the dual tree to the balanced triangulation. Show how to use T and the balanced triangulation to answer a query to decide whether point q is inside P or outside it. The query time should be $O(\log n)$, where n is the number of vertices of P . (Hint: use T to maintain the closest point in P_i to q , and use this to decide in constant time what is the closest point in P_{i+1} to q .)

2 Given a x -monotone polygonal chain C with n vertices, show how to preprocess it in linear time, such that given a query point q , one can decide, in $O(\log n)$ time, whether q is below and above C , and what is the segment of C that intersects the vertical line that passes through q . Show how to use this to decide, in $O(\log n)$ whether a point p is inside a x -monotone polygon P with n vertices. Why would this method be preferable to the balanced triangulation used in the previous question (when used on a convex polygon)?

3 SWEEPING

- 3.A.** Given two x -monotone polygons, show how to compute their intersection polygon (which might be made out of several connected components) in $O(n)$ time, where n is the total number of vertices of P and X .
- 3.B.** You are given a set \mathcal{H} of n half-planes (a half plane is the region defined by a line - it is either all the points above a given line, or below it). Show an algorithm to compute the *convex* polygon $\cap_{h \in \mathcal{H}} h$ in $O(n \log n)$ time. (Hint: use (a).)
- 3.C.** Given two simple polygons P and Q , show how to compute their intersection polygon. How fast is your algorithm?
What the maximum number of connected components of the polygon $P \cap Q$ (provide a tight upper and lower bounds)?

4 Convexity revisited.

- 4.A.** Prove that for any set S of four points in the plane, there exists a partition of S into two subsets S_1, S_2 , such that $\mathcal{CH}(S_1) \cap \mathcal{CH}(S_2) \neq \emptyset$.
- 4.B.** Prove that any point x which is a convex combination of n points p_1, \dots, p_n in the plane, can be defined as a convex combination of three points of p_1, \dots, p_n . (Hint: use (a) and induction on the number of points.)
- 4.C.** Prove that for any set S of $d + 2$ points in \mathbb{R}^d , there exists a partition of S into two subsets S_1, S_2 , such that $\mathcal{CH}(S_1) \cap \mathcal{CH}(S_2) \neq \emptyset$, $S = S_1 \cup S_2$, and $S_1 \cap S_2 = \emptyset$. (Hint: Use (a) and induction on the dimension.)

5 ROBOT NAVIGATION

Given a set S of m simple polygons in the plane (called obstacles), with total complexity n , and start point s and end point t , find the shortest path between s and t (this is the path that a robot would take to move from s to t).

- 5.A.** For a point $q \in \mathbb{R}^2$, which is not contained in any of the obstacles, the *visibility polygon* of q , is the set of all the points in the plane that are visible from q . Show how to compute this visibility polygon in $O(n \log n)$ time.
- 5.B.** Show a $O(n^3)$ time algorithm for this problem. (Hint: Consider the shortest path, and analyze its structure. Build an appropriate graph, and do a Dijkstra in this graph.).
- 5.C.** Show a $O(n^2 \log n)$ time algorithm for this problem.

6 You are given a set of n triangles in the plane, show an algorithm, as fast as possible, that decides whether the square $[0, 1] \times [0, 1]$ is completely covered by the triangles.

7 FURTHEST NEIGHBOR

Let $P = \{p_1, \dots, p_n\}$ be a set of n points in the plane.

- 7.A.** A *partition* $\mathcal{P} = (S, T)$ of P is a decomposition of P into two sets $S, T \subseteq P$, such that $P = S \cup T$, and $S \cap T = \emptyset$.

Describe a *deterministic*^① algorithm to compute $m = O(\log n)$ partitions $\mathcal{P}_1, \dots, \mathcal{P}_m$ of P , such that for any pair of distinct points $p, q \in P$, there exists a partition $\mathcal{P}_i = (S_i, T_i)$, where $1 \leq i \leq m$, such that $p \in S_i$ and $q \in T_i$ or vice versa (i.e., $p \in T_i$ and $q \in S_i$). The running time of your algorithm should be $O(n \log n)$.

- 7.B.** Assume that you are given a black-box \mathcal{B} , such that given a point w and a set of points Q in the plane, one can compute in $O(|Q| \log |Q|)$ time, a data-structure \mathcal{X} , such that given any query point w in the plane, one can compute, in $O(\log |Q|)$ time, using the data-structure, the furthest point in Q from w (i.e., this is the point in Q with largest distance from w). To make things interesting, assume that if $q \in Q$, then the data-structure does not work.

Describe an algorithm that uses \mathcal{B} , and such that computes, in $O(n \log^2 n)$ time, for any point $p \in P$, its furthest neighbor in $P \setminus \{p\}$.

8 NEAREST NEIGHBOR.

Let P be a set of n points in the plane. For a point $p \in P$, its nearest neighbor in P , is the point in $P \setminus \{p\}$ which has the smallest distance to p . Show how to compute for every point in P its nearest neighbor in $O(n \log n)$ time.

^①There is a very nice and simple randomized algorithm for this problem, you can think about it if you are interested.

50.2. Midterm 1

1 Consider the following two problems:

Problem Π_1 : Given a graph G , what is the size of the largest clique in G ?

Problem Π_2 : Given a graph G , what is the size of the smallest vertex cover of G ?

Which of the following best describes the relationship between problems Π_1 and Π_2 ?

1. Π_1 can be reduced to Π_2 in polynomial time.
2. Π_2 can be reduced to Π_1 in polynomial time.
3. A and B.
4. none of the above.

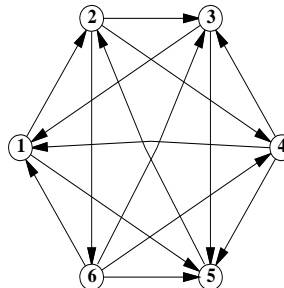
2 In the **2CNF-SAT** problem, you are given a formula which is a conjunction of clauses, where each clause is a \vee (i.e., or) of two literals, and you have to decide whether there is a satisfying assignment. For example, $F = (x_1 \vee \bar{x}_2) \wedge (x_3 \vee x_4)$ is a valid instance of **2CNF-SAT**. Then:

1. **2CNF-SAT** is in P and in NP .
2. **2CNF-SAT** is NP -Complete.
3. **2CNF-SAT** is NP -Hard.
4. **2CNF-SAT** can be solved in sub-linear time, using random assignment.
5. None of the above.

3 A person goes into a grocery store, and decides to buy items with total weight as large as possible, restricted only by the current amount of money he has. (Assume that the person knows the weight of every item in the grocery store.) What of the following statements is correct?

1. This problem is not actually hard, there is a polynomial time algorithm.
2. This problem is **NP**-hard, and it can not be approximated.
3. This problem is **NP**-hard, but it can be solved in polynomial time if he is allowed to buy fractions of items.
4. None of the above.

4 A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once.



A six-vertex tournament containing the Hamiltonian path $6 \rightarrow 4 \rightarrow 5 \rightarrow 2 \rightarrow 3 \rightarrow 1$.

Given a tournament G , determining if it has a Hamiltonian path, takes:

1. $O(n^2)$ time.
2. $O(n^2 \log n)$ time.
3. This problem is NP-Hard.
4. Constant time, because all tournaments have a Hamiltonian path.

5 Given a graph G with n vertices, deciding if G has a simple path of length at least $n/2$:

1. Can be done in $\Theta(n^3)$ time.
2. Can be done in $\Theta(n^3 \log n)$ time.
3. Can be done in $\Theta(n^2 \log n)$ time.
4. Can be done in $\Theta(n^4 \log n)$ time.
5. None of the above.

6 Let $G = (V, E)$ be a directed acyclic graph with n vertices and at most one edge between any two distinct nodes. What is the largest $|E|$ can be?

1. $n(n-1)(n-2)/6$.
2. $n^{3/2}$.
3. $7n + 6$.
4. $\frac{n(n-1)}{2}$.
5. $n!$.

7 Given a tree T with n vertices, finding the largest independent set in T is:

1. NP-Hard.
2. Can be done in $O(n)$ time.
3. Can be done in $O(n^2)$ time and there is no faster algorithm.
4. Can be done in $O(n \log n)$ time and there is no faster algorithm.

8 If $f(n) = O(F(n))$ and $g(n) = O(G(n))$ (assume that $g(n), G(n) \geq 1$ for all n) then

1. $\frac{f(n)}{g(n)} = O\left(\frac{F(n)}{G(n)}\right)$.
2. $\frac{f(n)}{g(n)} = o\left(\frac{F(n)}{G(n)}\right)$.
3. $\frac{f(n)}{g(n)} = \Theta\left(\frac{F(n)}{G(n)}\right)$.
4. $\frac{f(n)}{g(n)} = \Omega\left(\frac{F(n)}{G(n)}\right)$.
5. None of the above.

9 Consider the following problem:

Largest Subset

Instance: A set P of n real numbers, and parameters k and l .

Question: Is there a subset of k numbers of P such that their sum exceeds l ?

The Largest Subset problem is

1. NP-Complete.
2. NP-Hard.
3. Unsolvable.
4. In P .
5. None of the above.

10 Consider the following problem:

Subgraph embedding

Instance: A graph G and a graph H both with at most n vertices.

Question: Is there a mapping $f : V(G) \rightarrow V(H)$, such that $uv \in E(G)$ implies that $f(u)f(v) \in E(H)$, and for any $u, v \in V(G)$ such that $u \neq v$, we have $f(u) \neq f(v)$.

The problem **Subgraph embedding** is

1. Can be solved in $O(n^2)$ time.
2. Can be solved in $O(n^2 \log n)$ time.
3. Can be solved in $O(n^3 \log n)$ time.
4. None of the above.

11 Given a graph G , with positive weights on the edges. Computing the shortest path between two vertices $u, v \in V(G)$, is

1. NP-Complete.
2. Can be done in polynomial time.
3. None of the above.

12 Depressed after seeing a play by Arthur Miller, the traveling salesman, decided that the next time he plans his tour, he would allow himself to visit a town more than once, if it means he can drive less. Let us call the decision version of this problem **TSP REVISIT**. The problem **TSP REVISIT** is

1. Can be solved in $O(n^6)$ time using dynamic programming.
2. NP-Complete.
3. Equivalent to computing MST of the weighted graph.
4. None of the above.

13 You are given a standard chessboard with a specific configuration of white and black pieces. You are asked to decide whether if the white moves first, it wins the game. This can be determined by a program in:

1. Constant time.
2. Linear time.
3. Quadratic time.
4. Cubic time.
5. none of the above.

14 The **Knapsack** problem, is the following:

Knapsack

Instance: A set U of n elements, for each $u \in U$ a size $s(u) \in \mathbb{Z}^+$ and a value $v(u) \in \mathbb{Z}^+$, and $B, V \in \mathbb{Z}^+$. (Note, that all the numbers are positive integers.)

Question: Is there a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u) \geq V$?

1. The Knapsack problem can be solved in polynomial time.
2. The Knapsack problem can not be solved in polynomial time, unless $P = NP$. It remains NP -complete even if $B = V = O(n)$.
3. The Knapsack problem is NPC . However, it can be solved in polynomial time if $B = V = O(n)$.
4. None of the above.

15 Given two sorted arrays A and B (containing all distinct values), given a number x , determining how many elements of $A[1..n] \cup B[1..m]$ are smaller than x can be done in:

1. $O(\log(nm))$ time.
2. Linear time (i.e., $O(m+n)$).
3. $O(nm)$ time.
4. None of the above.

16 Consider the following *closest-point heuristic* for building an approximate traveling-salesman tour. Begin with a trivial cycle consisting of a single arbitrary chosen vertex. At each step, identify the vertex u that is not on the cycle but whose distance to any vertex on the cycle is minimum. Suppose that the vertex on the cycle that is nearest u is vertex v . Extend the cycle to include u by inserting u just after v . Repeat until all vertices are on the cycle.

This heuristic returns a path of length

1. at most $O(\log n)$ longer than optimal.
2. arbitrarily longer than the optimal TSP.
3. at most twice the optimal.
4. None of the above.

17 Given an instance of **Max 5SAT** (every clause has five literals), one can α -approximate it in polynomial time. Where α is

1. $O(n^2)$.
2. $1/4$.
3. $7/8$.
4. $31/32$.

18 Joe Smith was asked during a previous exam in 373 to prove that **Min Edge Coloring** is NP-Complete.

Min Edge Coloring

Instance: An undirected graph $G = (V, E)$, and a parameter k .

Question: Can one find a coloring χ , of the *edges* of G using k colors, such that for any two edges e, e' that share an endpoint, we have $\chi(e) \neq \chi(e')$.

And here is the proof that Joe Smith provided:

Proof: Clearly, the problem is in NP, as given a coloring, we can verify that it is valid by just scanning the graph. As for the reduction, we will reduce it from *Graph Coloring*. Indeed, given an instance G, k of the **Min Edge Coloring** problem, we construct a graph $H = (E(G), W)$, where every edge of G is a vertex in the new graph, and two vertices of H are connected iff the two corresponding edges in G share an endpoint. Clearly, there is a valid coloring of the vertices of H using k colors, iff there is a valid coloring of the edges of G using k colors. Thus, **Min Edge Coloring** is NP-Complete. ■

This proof is

1. Correct.
2. Incorrect because there was never a student name Joe Smith in cs373, he does not own the Brooklyn bridge, and **Min Edge Coloring** is not NP-Complete.
3. Incorrect.
4. None of the above.

19 During your visit to Greece, the oracle of Delphi had given you a black-box that can output, in constant time, given a circuit, the smallest circuit that is equivalent to it (i.e., circuit with minimum number of gates). Using this black box, you can:

1. Solve **3SAT** in polynomial time.
2. Solve all problems in **NP** in linear time.
3. Solve all problems in **co-NP** in linear time.
4. Solve the **Halting** problem in polynomial time (namely, given a program and an input, decide whether the program stops on the given input).
5. None of the above.

20 What is the solution to the recurrence $f(n) = f(\lfloor n/6 \rfloor) + f(\lfloor n/3 \rfloor) + f(\lfloor n/2 \rfloor) + 5$?

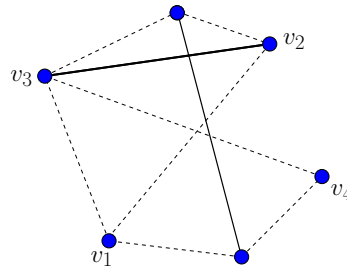
1. 42.
2. $\Theta(n^2 / \log^* n)$.
3. $\Theta(n)$.
4. $\Theta(n^{\log_2 6})$.
5. $\Theta(\sqrt{n})$.

50.3. Midterm 2

1 MATCHINGS (25 PTS.)

Let be a matching M in an undirected graph G , such that M is not a maximum matching. A *fixup path* is a path v_1, v_2, \dots, v_k in G , such that v_1 and v_k are unmatched by M (i.e., there is no edge in M adjacent to v_1 and v_k), the edges $v_2v_3, v_4v_5, v_6v_7, \dots, v_{k-2}v_{k-1}$ are in M , and $v_1v_2, v_3v_4, v_5v_6, \dots, v_{k-1}v_k$ are *not* in M .

For example, in the following graph: ◦



$v_1v_2v_3v_4$ is a fixup path.

- 1.A. (10 PTS.) Given a matching M , and a fixup path π for M , describe an algorithm that computes a matching M' such that $|M'| > |M|$.
- 1.B. (15 PTS.) Prove that for a non-maximum matching M in a graph G , there always exists a fixup path for M .
(You can not assume that G is bipartite. However, partial credit would be given for a proof that works only for the special case that G is bipartite.)

2 SORT THOSE NUMBERS (25 PTS.)

You had decided to build a sorting network, and you bought enough gates from your supplier Cheap Gates. Unfortunately, instead of the high quality comparators you expected, you got random comparators. Formally, a random comparator, receives two inputs, and with probability half, do nothing (i.e., passes the inputs directly to the outputs), and with probability half, it works correctly outputting the maximum number on the max output, and the minimum on the min output.

Describe a construction of a sorting network, that uses only random comparators, and sort correctly the n inputs, with probability $\geq 1 - 1/n$. How many gates does your sorting network have? Provide a proof that your sorting network works with this required confidence.

3 ALMOST MAGIC SQUARE (25 PTS.)

You are asked to fill the entries of an $n \times n$ matrix A by integers between 0 and a bound k , so that the sum of all entries in each row, and each column, comes to one of $2n$ numbers prespecified in advance. For example, the following instance

	17	5	4
6	(?	?	?)
9	(?	?	?)
11	(?	?	?)

with $k = 9$ has a solution

	17	5	4
6	(6	0	0)
9	(2	3	4)
11	(9	2	0)

Assume that the sum of the rows be specified in an array $R[1..n]$, and the sum of the columns specified in an array $C[1..n]$.

- 3.A.** (15 PTS.) Formulate this problem as a network flow problem.
- 3.B.** (10 PTS.) Write an algorithm for this problem and analyze its running time.

4 NUMBERS.
(25 PTS.)

Prove that if p is prime, then $(p - 1)! \equiv -1 \pmod{p}$.

50.4. Final

CS 373: Combinatorial Algorithms, Spring 2003

Final — 1:30-4:30 PM, Thursday, May 15, 2003

1 SWEEPING (20 PTS.)

- 1.A. (10 PTS.) Given two x -monotone polygons P and Q , show how to compute their intersection polygon (which might be made out of several connected components) in $O(n)$ time, where n is the total number of vertices of P and Q . (300 words)^②
- 1.B. (10 PTS.) You are given a set $\mathcal{H} = \{h_1, \dots, h_n\}$ of n half-planes (a half plane is the region defined by a line - it is either all the points above a given line, or below it). *Using (A)*, show an algorithm to compute the *convex* polygon $\cap_{i=1}^n h_i$ in $O(n \log n)$ time.

2 VERTEX COVER (20 PTS.)

VERTEX COVER

Instance: A graph $G = (V, E)$ and a positive integer $K \leq |V|$.

Question: Is there a *vertex cover* of size K or less for G , that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and for each edge $\{u, v\} \in E$, at least one of u and v belongs to V' ?

- 2.A. (9 PTS.) Prove that VERTEX COVER is NP-Complete.
- 2.B. (6 PTS.) Show a polynomial approximation algorithm to the VERTEX-COVER problem which is a factor 2 approximation of the optimal solution. Namely, your algorithm should output a set $X \subseteq V$, such that X is a vertex cover, and $|X| \leq 2K_{opt}$, where K_{opt} is the cardinality of the smallest vertex cover of G .
- 2.C. (5 PTS.) Prove that your approximation algorithm from (B) indeed provides a factor 2 approximation.

3 MAJORITY TREE (20 PTS.)

Consider a uniform rooted tree of height h (every leaf is at distance h from the root). The root, as well as any internal node, has 3 children. Each leaf has a boolean value associated with it. Each internal node returns the value returned by the majority of its children. The evaluation problem consists of determining the value of the root; at each step, an algorithm can choose one leaf whose value it wishes to read.

- 3.A. (5 PTS.) Describe a deterministic algorithm that runs in $O(n)$ time, that computes the value of the tree, where $n = 3^h$.
- 3.B. (5 PTS.) Describe (i.e., provide pseudo-code) a randomized algorithm for this problem, which is faster than the deterministic algorithm.

^②Your solution for this subquestion should not exceed 300 words. One line of text is about ten words. Note that the limit is quite conservative. Much shorter answers (that would get full credit) are possible and would cause us infinite happiness.

- 3.C.** (10 PTS.) Prove that the expected number of leaves read by your randomized algorithm on any instance is at most $O(n^c)$ (modify your randomized algorithm to achieve this if necessary), where c is a constant smaller than 1. (Of course, no credit would be given to an algorithm with expected linear running time.)

4 UNION FIND
(20 PTS.)

In the following, we consider a union-find data-structure constructed for n elements.

- 4.A.** (5 PTS.) For an element x in the union-find data-structure, describe how $\text{rank}(x)$ is being computed.
- 4.B.** (5 PTS.) Prove that during the execution of union-find there are at most $n/2^k$ elements that are assigned rank k .
- 4.C.** (5 PTS.) Prove that in a set of n elements, a sequence of n consecutive FIND operations take $O(n)$ time in total.
- 4.D.** (5 PTS.) Prove that in the worst case, the time to perform a single find operation is $O(\log n)$.

5 ADD THEM UP
(20 PTS.)

Consider two sets $A = \{a_1, \dots, a_k\}$ and $B = \{b_1, \dots, b_m\}$, each having at most n integers in the range from 0 to $10n$. We wish to compute the *Cartesian sum* of A and B , defined by

$$C = \{x + y \mid x \in A \text{ and } y \in B\}.$$

Note that the integers in C are in the range from 0 to $20n$. We want to find the elements of C and the number of times each element of C is realized as a sum of elements in A and B .

- 5.A.** (10 PTS.) Show how to reduce this problem to the problem of polynomial multiplication.
- 5.B.** (10 PTS.) Present an algorithm that solves this problem in $O(n \log n)$ time. (Partial credit would be given for a subquadratic algorithm for this problem. Slower algorithms would not get any points.)

Chapter 51

Fall 2003

Chapter 52

Spring 2005

Chapter 53

Spring 2006

Chapter 54

Fall 2007

Chapter 55

Fall 2009

Chapter 56

Spring 2011

Chapter 57

CS 473: Fundamental Algorithms, Spring 2013

HW 0 (due Tuesday, at noon, January 22, 2018)

Version: 1.12.

Required problems

- 1** (50 PTS.) The multi-trillion dollar game.



You are given k piles having in total n identical coins (each coin is a trillion dollar coin, of course, see picture above). In every step of the game, you take a coin from a pile P , and you can put it in any pile Q , if Q has at least 2 less coins than P . In particular, if P has two coins, you are allowed to take one of the coins of P and create a new pile. If a pile has only a single coin, you are allowed to take the coin and remove it from the game^①. The game is over when there are no more coins left. Note, that you can start a new pile by taking a coin from any pile that has at least 2 coins. As an example, consider the following sequence of moves in a game:

$$\begin{aligned} \{4, 2, 1\} &\implies \{4, 1, 1, 1\} \implies \{3, 2, 1, 1\} \implies \{2, 2, 2, 1\} \implies \{2, 2, 1, 1, 1\} \\ &\implies \{2, 1, 1, 1, 1, 1\} \implies \{2, 1, 1, 1, 1\} \implies \{2, 1, 1, 1\} \implies \{2, 1, 1\} \\ &\implies \{2, 1\} \implies \{2\} \implies \{1, 1\} \implies \{1\} \implies \{\}. \end{aligned}$$

- 1.A.** (25 PTS.) Prove, formally, that this game always terminates after a finite number of steps.
- 1.B.** (25 PTS.) Unfortunately, the world economy is suffering from hyper-inflation. A trillion dollar bucks do not go as far as they used to go. To keep you interested in the game, the rules have changed – every time you remove a coin from a pile, you are required to insert two coins into two different piles (again, these new piles need to have 2 less coins than the original pile before the move) [you get the extra coin for such a move from the IMF]. Again, if a pile has a single coin, you can take the coin. Prove, formally, that this game always terminates after a finite number of steps. A valid game in this case might look like:

$$\begin{aligned} \{4, 2, 1\} &\implies \{3, 3, 2\} \implies \{3, 2, 2, 1, 1\} \implies \{3, 2, 2, 1\} \implies \{3, 2, 2\} \\ &\implies \{3, 2, 1, 1, 1\} \implies \{2, 2, 2, 2, 1\} \implies \{2, 2, 2, 2\} \implies \{2, 2, 2, 1, 1, 1\} \\ &\implies \{2, 2, 2, 1, 1\} \implies \{2, 2, 2, 1\} \implies \{2, 2, 2\} \implies \{2, 2, 1, 1, 1\} \\ &\implies \{2, 2, 1, 1\} \implies \{2, 2, 1\} \implies \{2, 2\} \implies \{2, 1, 1, 1\} \\ &\implies \{1, 1, 1, 1, 1, 1\} \implies \{1, 1, 1, 1, 1\} \implies \{1, 1, 1, 1\} \implies \{1, 1, 1\} \\ &\implies \{1, 1\} \implies \{1\} \implies \{\}. \end{aligned}$$

- 2** (50 PTS.) The baobab tree.

^①And then you can buy whatever banana republic you want with it.

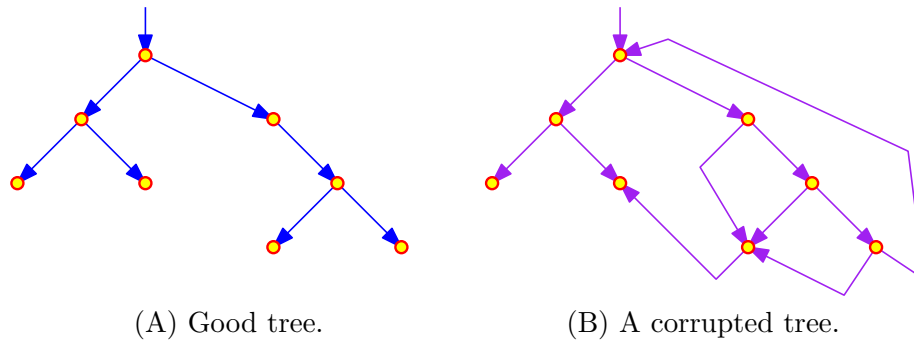


Figure 60.1: A good & bad trees. For clarity, the figure does show the parent pointers that each node has

You are given a pointer to the root of a binary tree. The tree is stored in a read only memory, and you can not modify it in any way. Now, normally, each node in the tree has a pointer to its left child, right child, and its parent (these pointers are NULL if they have nothing to point to). Unfortunately, your tree might have been corrupted by a bug in somebody else code^②, so that some of the pointers, now either point to some other node in the tree or contain NULL, or vice versa (i.e., potentially any pointer in the tree [or potentially all of them] might be corrupted [including parent, left and right pointers]). See Figure 60.1.

Describe an algorithm^③ that determines whether the tree is corrupted or not. Your algorithm must not modify the tree. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the tree, and use $O(1)$ extra space (not counting the tree itself).

Observe that a regular recursive algorithm uses a stack, and the space it uses for the stack might be unbounded if the depth of the recursion is not bounded by a constant. Similarly, using arrays or stacks of unbounded size violates the requirement that the space used is $O(1)$.

60.1.2. Homework 1

CS 473: Fundamental Algorithms, Spring 2013 HW 1 (due Monday, at noon, January 28, 2018)

Version: 1.01.

1 (30 PTS.) The closet mayhem problem.

A new broom closet was built for the Magical Science department of the Unseen University. The department has n students, and every student has a broom. The closet has $m \geq n$ slots where one can place a broom. The i th broom b_i , can be placed only into two possible slots, denoted by x_i and x'_i , where $x_i, x'_i \in \{1, \dots, m\}$, for $i = 1, \dots, n$. Given these locations, design an algorithm that decides in polynomial time if there is a legal way to place all of them in the closet, such that no two brooms share a slot. To this end, answer the following.

^②After all, *your* code is always completely 100% bug-free. Isn't that right, Mr. Gates?

^③Since you've read the Homework Instructions, so you know what the phrase 'describe an algorithm' means. Right?

- 1.A. (5 PTS.) Consider the graph G with $2n$ nodes, where for every broom there are two nodes $[i : x_i]$ or $[i : x'_i]$. For $\alpha \in \{x_i, x'_i\}$ and $\beta \in \{x_j, x'_j\}$, place an edge from $[i : \alpha]$ to $[j : \beta]$, if placing the i th broom at α implies that the j th broom must be placed in the slot β because the other placement of the j th broom is α . How many edges can this graph has in the worst case? What is the running time of your algorithm to compute this graph?
- 1.B. (5 PTS.) If there is a path in G from $[i : \alpha]$ to $[j : \beta]$, then we say that $b_i = \alpha$ **forces** $b_j = \beta$. Prove that if $b_i = x_i$ forces $b_j = x_j$ then the “reverse” must hold; that is, $b_j = x'_j$ forces $b_i = x'_i$.
- 1.C. (5 PTS.) Prove that if $[i : x_i]$ and $[i : x'_i]$ are in the same strong connected component of G , then there is no legal way to place the brooms in the closet.
- 1.D. (5 PTS.) Assume that there is a legal solution, and consider a strong connected component X of G involving brooms, say, b_1, \dots, b_t in G ; that is, X is a set of vertices of the form $[1 : x_1], \dots, [t : x_t]$. Then, prove that $[1 : x'_1], \dots, [t : x'_t]$ form their own connected component in G . Let this component be the **mirror** of X .
- 1.E. (5 PTS.) Prove that if X is a strong connected component of G that is a sink in the meta graph G^{SCC} , then the mirror of X is a source in the meta graph G^{SCC} .
- 1.F. (5 PTS.) Consider the algorithm that takes the sink X of the meta-graph G^{SCC} , use the associated slots as specified by the nodes in X , remove the vertices of X from G and the mirror of X from G , and repeating this process on the remaining graph. Prove that this algorithm generates a legal placement of the brooms in the closet (or otherwise outputs that no such placement exists). Also, describe how to implement this algorithm efficiently.
- What is the running time of your algorithm in the worst case as a function of n and m .

BTW, for this specific problem, there is a significantly simpler solution. However, the above solution is more general and can be used to solve other problems.

2 (30 PTS.) Heavy time.

Consider a DAG G with n vertices and m edges.

- 2.A. (5 PTS.) Assume that s is a sink in G . Describe how to compute in linear time a set of new edges such that s is the only sink in the resulting graph G (G has to be a DAG). How many edges does your algorithm add (the fewer, the better)?
- 2.B. (10 PTS.) Assume G has a sink vertex s . Some of the vertices of G are marked as being **significant**. Show an algorithm that in linear time computes all the vertices that can reach s via a path that goes through at least t significant vertices, where t is a prespecified parameter. (Hint: Solve the problem first for $t = 1$ and then generalize.)
- 2.C. (10 PTS.) Assume the edges of G have weights assigned to them. Show an algorithm, as fast as possible, that computes for all the vertices v in G the weight of the **heaviest** path from v to s .
- 2.D. (5 PTS.) Using the above, describe how to compute, in linear time, a path that visits all the vertices of G if such a path exists.

3 (40 PTS.) Wishful graph.

Let $G = (V, E)$ be a directed graph. Define a relation R on the nodes V as follows: uRv iff u can reach v or v can reach u .

- 3.A. (10 PTS.) Is R an equivalence relation? If yes, give a proof, otherwise give an example to show it is false.
- 3.B. (30 PTS.) Call G **uselessly-connected** if for every pair of nodes $u, v \in V$, we have that there is either a path from u to v in G , or a path from v to u in G . Give a linear time algorithm to determine if G is uselessly-connected, here linear time is $O(m + n)$, where $m = |E|$ and $n = |V|$.

60.1.3. Homework 2

HW 2 (due Monday, at noon, February 4, 2018)

CS 473: Fundamental Algorithms, Spring 2013

1 (30 PTS.) Climb On.

Kris is a professional rock climber who is competing in the U.S. climbing nationals. The competition requires him to complete the following task: He is given a set of n holds that he might use to create a route while climbing, and a list of m pairs (x_i, x'_i) of holds indicating that it is possible to move from x_i to x'_i (but it might not be possible to move in the other direction). Kris needs to figure out his climbing sequence so that he uses as many holds as possible, since using each hold earns him points. The rules of the competition are that he has to figure out the start and end of his climbing route, he can only move between pre-specified pairs of holds and he is allowed to use each hold as many times as he needs, but reusing holds doesn't earn him any more points (and makes his arms more tired).

1.A. (15 PTS.) Define the natural graph representing the input. Describe an algorithm to solve the problem if the graph is a DAG. How fast is your algorithm? (The faster, the better.)

1.B. (15 PTS.) Describe an algorithm to solve this problem for a general directed graph. How fast is your algorithm? (The faster, the better.)

Note, that your algorithm should output the number of holds by the optimal solution. What is the running time of your algorithm for this?

You also need a way to output the solution itself. Observe that finding the shortest solution in the number of edges is NP-Hard. As such, it is enough if your algorithm output any solution, that has polynomial length in the graph size. How fast is your algorithm in the worst case for outputting this path (which is formally a walk since it potentially visits vertices more than once)? (We are not expecting a fast algorithm for outputting the path itself - any solution that works in polynomial time would be acceptable. In particular, a short intuitive explanation for your algorithm would be enough for the part outputting the path.)

2 (30 PTS.) Climb Off.

Many years later, in a land far far away, Kris retired from competitive climbing (after he won all the U.S. national competitions for 10 years in a row), he became a route setter for competitions. However, as the years passed, the rules changed. Climbers now, along with the set of n holds and valid moves (x_i, x'_i) between them (as before), are also given a start hold s and a finish hold t . To get full points, they are required to climb the route using the shortest path from s to t , where the distance between two holds is specified by the route setter. Suppose the directed graph that corresponds to a route is $G = (V, E)$ where the non-negative number $\ell(e)$ is the length of $e \in E$. The way Kris chooses to set competition routes is by revisiting the ones he competed on in the past (and setting s and t to be the start and end holds he had used). For one of the routes, he noticed that the existing shortest path distance between s and t in G is too tiring, and he proposed to add exactly one move (one edge to the graph) to improve the situation. The candidate edges from which he had to choose is given by $E' = \{e_1, e_2, \dots, e_k\}$ and you can assume that $E \cap E' = \emptyset$. The length of the e_i is $\alpha_i \geq 0$. Your goal is to help out Kris (he is too old now for these computations, after all) and figure out which of these k edges will result in the most reduction in the shortest path distance from s to t . Describe an algorithm for this problem that runs in time $O(m + n \log n + k)$, where $m = |E|$ and $n = |V|$.

(Note that one can easily solve this problem, in $O(k(m + n \log n))$ time, by running Dijkstra's algorithm k times, one for each G_i where G_i is the graph obtained by adding e_i to G .)

3 (40 PTS.) Only two negative length edges.

You are given a directed graph $G = (V, E)$ where each edge e has a length/cost c_e and you want to find shortest path distances from a given node s to all the nodes in V . Suppose there are at most two edges $f_1 = (u, v)$ and $f_2 = (w, z)$ that have negative length and the rest have non-negative lengths. The Bellman-Ford algorithm for shortest paths with negative length edges takes $O(nm)$ time where $n = |V|$ and $m = |E|$. Show that you can take advantage of the fact that there are only at most two negative length edges to find shortest path distances from s in $O(n \log n + m)$ time — effectively this is the running time for running Dijkstra's algorithm. Your algorithm should output the following: *either* that the graph has a negative length cycle reachable from s , *or* the shortest path distances from s to all the nodes $v \in V$.

Hint: Solve first the case that you have only a single negative edge.

(For fun and for no credit [and definitely not for the exam], think about how to solve this algorithm for the case when there are k negative edges. You want an algorithm that is faster than Bellman-Ford if k is sufficiently small.)

60.1.4. Homework 3

HW 3 (due Monday, at noon, February 11, 2018)

CS 473: Fundamental Algorithms, Spring 2013

1 (50 PTS.) The vampire strike back.

Several years back vampires took over Champaign, IL (this story did not get sufficient coverage in the lamestream media). It turns out that there are several kinds of vampires. The only way to decide if two vampires V_1 and V_2 are of the same kind, is to ask them to bite each other (`biteEachOther(V_1, V_2)`) — if they both get sick they are of different kinds, otherwise, they are of the same kind. This check takes constant time. Specifically, `biteEachOther(V_1, V_2)` returns true if V_1 and V_2 are of the same kind, and false otherwise.

As is usual in such cases, there is one kind that is dominant and form the majority of the Vampires. Given the n vampires V_1, \dots, V_n living in Champaign, describe an algorithm that uses the `biteEachOther` operation and discovers all the vampires that belongs to the majority type. Formally, $n \geq 4$, and there are at least 3 different kinds of vampires, and you have to discover all the vampires that belong to the kind that has at least $n/2$ members. You have to describe a deterministic algorithm that solves this problem in $O(n \log n)$ time (a faster algorithm is possible, but it is hard). An algorithm that runs in $O(n^2)$ time (or slower) would get at most 25% of the points, but you might want to first verify you know how to do it with this running time bound.

2 (50 PTS.) Liberty towers.

A German mathematician developed a new variant of the Towers of Hanoi game, known in the US literature as the “Liberty Towers” game^①. Here, there are n disks placed on the first peg, and there are $k \geq 3$ pegs, numbered from 1 to k . You are allowed to move disks only on adjacent pegs (i.e., for peg i to peg $i + 1$, or vice versa). Naturally, you are not allowed to put a bigger disk on a smaller disk. Your mission is to move all the disks from the first peg to the last peg.

^①From Wikipedia: “During World War I, due to concerns the American public would reject a product with a German name, American sauerkraut makers relabeled their product as “Liberty cabbage” for the duration of the war.”

- 2.A. (15 PTS.) Describe a recursive algorithm for the case $k = 3$. How many moves does your algorithm do?
- 2.B. (15 PTS.) Describe a recursive algorithm for the case $k = n+1$. How many moves does your algorithm do? (The fewer, the better, naturally. For partial credit, the number of moves has to be at least polynomial in n .)
- 2.C. (20 PTS.) Describe a recursive algorithm for the general case. How many moves does your algorithm do? (The fewer, the better, naturally.) In particular, how many moves does your algorithm do for $n = k^2$? A good solution should yield a solution that is as good as your solution for the (A) and (B) parts.

This question is somewhat more open ended – we have no specific solution at mind – do your best – but don’t spend the rest of your life on this problem.

60.1.5. Homework 4

HW 4 (due Monday, at noon, February 25, 2018)

CS 473: Fundamental Algorithms, Spring 2013

1 (30 PTS.) Goodbye, and thanks for all the negative cycles.

You are given a directed graph G with (possibly negative) weights on the edges, and a source vertex s .

- 1.A. (20 PTS.) Show how to modify Bellman-Ford so that it outputs a negative cycle it had found in the graph reachable from the source s . **Prove** that your algorithm indeed outputs a negative cycle in the graph.
- 1.B. (10 PTS.) Describe an algorithm that computes for all the vertices in the given graph their distance from s .

Notice, that your algorithm needs to correctly handle vertices in G whose distance from s is $-\infty$ (there is a walk from s to such a vertex that includes a negative cycle).

For full credit, the running time of your algorithm must be $O(mn)$. (You do not have to use (A) to do this part.)

2 (30 PTS.) Longest common palindrome.

You are given two strings S and T of length n . Describe an algorithm, as fast as possible, that output the longest palindrome that appears, as a substring (a substring here might not necessarily a contiguous in the original string), in both S and T . For example, for

$$S = \underline{a}o\underline{b}r\underline{a}c\underline{a}d\underline{a}b\underline{o}r\underline{a} \quad T = \underline{a}h\underline{o}m\underline{a}l\underline{o} \text{ graphica,}$$

the longest common palindrome, seems to be “aoaoa”.

3 (40 PTS.) Recovering a message.

You are given a binary string S of length n that was transmitted using a new transmitted. Unfortunately, there is noise in the received string - new fake bits were added to it. Because of the way the message was encoded, you know that certain strings s_1, \dots, s_t can not appear consecutively in the original string.

Compute the longest substring (not necessarily consecutive) in S that does not contain any of the forbidden strings (consecutively). Formally, each of the forbidden strings can not appear as (consecutive) substrings of the output string. For example, consider the input string 11000011100000111000000111, and the forbidden substrings $s_1 = 00$ and $s_2 = 11$. Clearly, the longest legal output string here is 1010101.

(Hint: You might want to use material you learned in CS 373 in solving this question.)

- 3.A.** (20 PTS.) (This part is significantly harder than the other part – you might want to solve the other part first.) Describe an algorithm for the case that t is small (i.e., think about t as being a constant [hint: Solve the case $t = 1$ and $t = 2$ first]), but the strings s_1, \dots, s_t might be arbitrarily long. How fast is your algorithm? (Faster is better, naturally.)

For full credit for this part, it is enough if you solve the problem for $t = 1$ (the problem turned out to be harder than expected).

- 3.B.** (20 PTS.) Describe an algorithm for the case that t might be large, but the strings s_1, \dots, s_t are of length at most ℓ , and ℓ is relatively small (i.e., think about ℓ as being, say, 5). How fast is your algorithm? (Faster is better, naturally.)

60.1.6. Homework 5

HW 5 (due Monday, at noon, March 4, 2018)

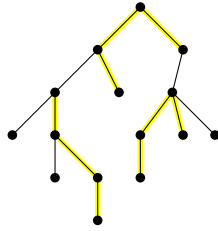
CS 473: Fundamental Algorithms, Spring 2013

1 (40 PTS.) Simultaneous Climbs.

One day, Kris (whom you all know and love from past climbing competitions) got tired of climbing in a gym and decided to take a very large group of climber friends (after, all he is a popular guy) outside to climb. The climbing area where they went, had a huge wide boulder, not very tall, with various marked hand and foot holds. Kris took a look and quickly figured out an "allowed" set of moves that his group of friends would do so that they get from one hold to another. He also figured out the difficulty of each individual move and assigned a grade (weight) to it. The higher the weight, the harder the move. Let $G = (V, E)$ be the (undirected) graph with a vertex for each hold and an edge between two holds (u, v) if v can be reached from u (and vice versa) by one of the moves that Kris decided. For an edge $(u, v) \in E$, we have a weight $w(uv)$ associated with (u, v) which represents the difficulty that he assigned to that particular move (it is always positive, negative weights would mean that climbers can defy gravity).

A *k -climb* is a sequence where a climber performs k moves in sequence. In graph G it is represented by a simple path with exactly k edges in it. Two k -climbs are disjoint if they do not share any vertex. A collection M of k -climbs is a *k -climb packing* if all pairs of climbs of M are disjoint (for $k = 1$ the set M is a matching in the graph). The total weight of a k -climb packing is the total weight of the edges used by the climbers.

Kris and his friends decided to play a game (they are all very good climbers), where as many climbers as possible are simultaneously on the wall and each climber needs to perform a set of k moves in sequence. In other words, they are interested in the problem of computing the maximum weight k -climb packing in G . In general, this problem seems hard.



Describe an efficient algorithm (i.e., provide pseudo-code, etc), as fast as possible, for computing the maximum weight k -climb packing when G is a rooted tree (fortunately for the tree case this is much easier) or a forest (collection of rooted trees).

Your algorithm should be recursive and use memoization to achieve efficiency. (You can not assume G is a binary tree - a node might have arbitrary number of children.) What is the running time of your algorithm as function of $n = |V(G)|$ and k ?

For an example, the figure shows a tree with a possible 3-climb packing.

2 (40 PTS.) Help Fellow Climber Out and ARC Scheduling.

2.A. (20 PTS.) Since Kris is a professional climber, the moves he had in mind for his friends to climb, unfortunately, did not result to the graph G from the previous question being a tree (or a collection of trees). In order for his friends to have fun on the boulder, you need to help Kris (who hasn't brushed up his algorithms in a long time) reduce his set of moves (edges in G) so that G ends up being a tree or a forest but also that the total difficulty of moves he ignores is as small as possible (he doesn't want to be too easy on his friends). Formally, for the graph $G = (V, E)$ that appears in the previous question, describe an algorithm that removes the smallest weight subset of edges such that the remaining graph is a tree or a forest.

2.B. (20 PTS.) Consider a variant of interval scheduling except now the intervals are arcs on a circle. The goal is to find the maximum number of arcs that do not overlap. More formally, let C be the circle on the plane centered at the origin with unit radius. Let A_1, \dots, A_n be a collection of arcs on the circle where an arc A_i is specified by two angles $\alpha_i \in [0, 2\pi]$ and $\beta_i \in [0, 2\pi]$: the arc starts at the point on the circle C with angle α_i and goes counter-clockwise till the point on C at angle β_i (the end points are part of the arc). Two arcs overlap if they share a point on the circle. Describe an algorithm to find the maximum number of non-overlapping arcs in the given set of arcs.

3 (20 PTS.) Process these words.

In a word processor the goal of “pretty-printing” is to take text with a ragged right margin, like this

```

I guess it takes two
to progress
from an apprentice to a
legitimate surfer. Two digits
in front of the
size in feet of the
wave one needs to take, two double overhead waves that holds
one under after
the wipeout, and two equally sized pieces that
one's previously intact board comes up
on the surface as.

```

and turn it into text whose right margin is as “even” as possible, like this

I guess it takes two to progress
from an apprentice to a legitimate
surfer. Two digits in front of
the size in feet of the wave one
needs to take, two double overhead
waves that holds one under after
the wipeout, and two equally sized
pieces that one's previously intact
board comes up on the surface as.

To make this precise enough for us to start thinking about how to write a pretty-printer for text, we need to figure out what it means for the right margins to be “even”. So suppose our text consists of a sequence of *words*, $W = \{w_1, w_2, \dots, w_n\}$, where w_i consists of c_i characters. We have a maximum line length of L . We will assume we have a fixed-width font and ignore issues of punctuation or hyphenation.

A *formatting* of W consists of an ordered partition of the words in W into *lines*. In the words assigned to a single line, there should be a space after each word except the last; and so if w_j, w_{j+1}, \dots, w_k are assigned to one line, then we should have

$$\left[\sum_{i=j}^{k-1} (c_i + 1) \right] + c_k \leq L.$$

We will call an assignment of words to a line *valid* if it satisfies this inequality. The difference between the left-hand side and the right-hand side will be called the *slack* of the line—that is, the number of spaces left at the right margin.

Given a partition of a set of words W , the *penalty* of the formatting is the sum of the *squares* of the slacks of all lines (including the last line). Give an efficient algorithm to find a partition of a set of words W into valid lines, so that the penalty of the formatting becomes minimized.

60.1.7. Homework 6

HW 6 (due Monday, at noon, March 11, 2018)

CS 473: Fundamental Algorithms, Spring 2013

1 (30 PTS.) The climbing vampires of Champaign.

Every year the vampires of Champaign elect their leader using a somewhat bizarre process. They all go and travel to Peru and try to climb Siula Grande^①. The rule of the game is that they all start from the base camp and climb for 5 hours, and then they stop (it is a competition, so each vampire probably reached a different location in the mountain). After that, the base station start calling the vampires one after the other, and ask them for their location, and more significantly how high are they in the mountain (the vampires have each a unique ID between 1 and n which they can use to call them). The vampire that is highest on the mountain is the new leader.

To avoid a vacuum in leadership, the leader is being updated as the results come in. Specifically, if after contacting the first i vampires the last vampire contacted is the highest, then the base station sends n messages to all the vampires telling them that this vampire is the new leader, where n is the number of vampires participating.

^①See http://en.wikipedia.org/wiki/Touching_the_Void_%28film%29.

- 1.A. (10 PTS.) Given that the given vampires are v_1, \dots, v_n , describe a strategy that minimizes the overall number of messages sent. How many messages does your scheme send in the worst case? If you decide on a randomized strategy, how many messages does your scheme send in expectation? The smaller the number of messages, the better. Prove your answer.
- 1.B. (10 PTS.) How many times does the leader change in the worst case (and also in expectation if your scheme is randomized) under your scheme (the fewer the better). Prove your answer.
- 1.C. (10 PTS.) A new scheme was suggested that minimizes the number of messages sent: Whenever a new leader is discovered, you announce it only to the vampires already contacted. Describe a scheme that orders the vampires, such that the number of messages sent is minimized. What is the number of messages sent by your scheme (either in the worst case, and in expectation if your scheme is randomized). The fewer the better.

For this question, if you have two bounds x and y , then if $x < y$ then x is preferable to y , even if x holds only in expectation, and y is a worst case bound.

2 (40 PTS.) Collapse and shrink.

- 2.A. (5 PTS.) Consider the procedure that receives as input an undirected weighted graph G with n vertices and m edges, and weight x , and outputs the graph $G_{<x}$ that results from removing all the edges in G that have weight larger than (or equal to) x . Describe (shortly – no need for pseudo code) an algorithm for computing $G_{<x}$. How fast is your algorithm?

The graph $G_{<x}$ might not be connected – how would you compute its connected components?

- 2.B. (5 PTS.) Consider the procedure that receives as input an undirected weighted graph G , and a partition \mathcal{V} of the vertices of G into k disjoint sets V_1, \dots, V_k . The *meta graph* $G(\mathcal{V})$ of G induced by \mathcal{V} is a graph having k vertices, v_1, \dots, v_k , where $v_i v_j$ has an edge if and only if, there is an edge between some vertex of V_i and some vertex of V_j . The weight of such an edge $v_i v_j$ is the minimum weight of any edge between vertices in V_i and vertices in V_j .

Describe an algorithm, as fast as possible, for computing the meta-graph $G(\mathcal{V})$. You are not allowed to use hashing for this question, but you can use that **RadixSort** works in linear time (see wikipedia if you do not know **RadixSort**). How fast is your algorithm?

- 2.C. (10 PTS.) Consider the randomized algorithm that starts with a graph G with m edges and n vertices. Initially it sets $G_0 = G$. In the i th iteration, it checks if G_{i-1} is a single edge. If so, it stops and outputs the weight of this edge. Otherwise, it randomly choose an edge $e_i \in E(G_{i-1})$. It then computes the graph $H_i = (G_{i-1})_{<w(e_i)}$, as described above.

- If the graph H_i is connected then it sets $G_i = H_i$ and continues to the next iteration.
- Otherwise, H_i is not connected, then it computes the connected components of H_i , and their partition \mathcal{V}_i of the vertices of G_{i-1} (the vertices of each connected component are a set in this partition). Next, it sets G_i to be the meta-graph $G_{i-1}(\mathcal{V}_i)$.

Let m_i be the number of edges of the graph G_i . Prove that if you know the value of m_{i-1} , then $\mathbb{E}[m_i] \leq (7/8)m_{i-1}$ (a better constant is probably true). Conclude that $\mathbb{E}[m_i] \leq (7/8)^i m$.

- 2.D. (15 PTS.) What is the expected running time of the algorithm describe above? **Prove** your answer. (The better your bound is, the better it is.)
- 2.E. (5 PTS.) What does the above algorithm compute, as far as the original graph G is concerned?

3 (30 PTS.) Selection revisited.

You are given two arrays of numbers $X[1 \dots n]$ and $Y[1 \dots m]$, where n is smaller than m .

- 3.A. (20 PTS.) Given a number k , and assuming both X and Y are sorted (say in increasing order), describe an algorithm, as fast as possible, for finding the k smallest number in the set $X \cup Y$ (assume all the numbers in X and Y are distinct).

- 3.B. (10 PTS.) Solve the same problem for the case that X is not sorted, but Y is sorted.
-

60.1.8. Homework 7

HW 7 (due Wednesday, at noon, March 27, 2018)

CS 473: Fundamental Algorithms, Spring 2013

1 (40 PTS.) Surf's up.

The Illinois (Vampire) Surfing Association decided to send a few teams to the surfing competition that happens every year in Kauai. It is decided that each team of surfers will satisfy the following: for every surfer in the Association, either she has to be in the team or one of her friends has to be in the team. Such a team is called *valid*. The president of the Association would like to send as many teams as possible to Kauai in order to maximize the chances of some Illinois team to win. His goal is to find the maximum number (*surfout* number) of mutually disjoint valid teams that can compete.

Let $G = (V, E)$ be an undirected graph where V consists of all the Illinois surfers and an edge (u, v) indicates that surfer u is friends with surfer v . Let δ be the degree of a minimum degree node in G . It is easy to see that the surfout number of G is at most $(\delta + 1)$ since each valid team has to contain u or some neighbor of u where u is a node with degree δ . In this problem we will see that the surfout number of an association of n surfers where the representing graph G has minimum degree δ is at least as large as $\lceil \frac{\delta+1}{c \ln n} \rceil$ for some sufficient large universal constant c . Note that this guarantees to send only 1 valid team in the competition if $\delta < c \ln n$ (the entire group of surfers can be chosen as the team). Let $k = \lceil \frac{\delta+1}{c \ln n} \rceil$. Consider the following randomized algorithm. To each surfer u independently give a team shirt with number $g(u)$ written on it, that is chosen uniformly at random from the numbers $\{1, 2, \dots, k\}$.

- 1.A. (20 PTS.) For a fixed surfer v and a fixed number i show that with probability at least $1 - 1/n^2$ there is a surfer with shirt number i that is either v or a neighbor of v . Choose c sufficiently large to ensure this.
- 1.B. (10 PTS.) Using the above show that for a fixed number i the set of surfers that are given shirts with number i form a valid surf team for G with probability at least $1 - 1/n$.
- 1.C. (10 PTS.) Using the above two parts argue that the surfout number of G is at least k .

2 (30 PTS.) Random walk.

Consider a full binary tree of height h . You start from the root, and at every stage you flip a coin and go the left subtree with probability half (if you get a head), and to the right subtree with probability half (if you get a tail). You arrive to a leaf, and let's assume you took k turns to the left (and $h - k$ turns to the right) traversing from the root to this leaf. Then the value written in this leaf is α^k , where $\alpha < 1$ some parameter.

Let X_h be the random variable that is the returned value.

- 2.A. (10 PTS.) Prove that $\mathbb{E}[X_h] = (\frac{1+\alpha}{2})^h$ by stating a recursive formula on this value, and solving this recurrence. Alternatively, you can prove this by a direct calculation.
- 2.B. (10 PTS.) Consider flipping a fair coin h times independently and interpret them as a path in the above tree. Let \mathcal{E} be the event that we get at most $h/4$ heads in these coin flips. Argue that \mathcal{E} happens if and only if $X_h \geq \alpha^{h/4}$.

- 2.C.** (10 PTS.) Markov's inequality states that for a positive random variable X we have that $\mathbb{P}[X \geq t] \leq \mathbb{E}[X/t]$. Let Y be the number of heads when flipping a fair coin h times. Using Markov's inequality, (A) and (B) prove that

$$\mathbb{P}[\text{Out of } h \text{ coin flips getting at most } h/4 \text{ heads}] \leq \left(\frac{1 + \alpha}{2\alpha^{1/4}}\right)^h.$$

In particular, by picking the appropriate value of α , prove that

$$\mathbb{P}[\text{Out of } h \text{ coin flips getting at most } h/4 \text{ heads}] \leq 0.88^h.$$

What is your value of α ?

3 (30 PTS.) Conditional probabilities and expectations.

Assume there are two random variable X and Y , and you know the value of Y (say it is y). The *conditional probability* of X given Y , written as $\mathbb{P}[X \mid Y]$, is the probability of X getting the value x , given that you know that $Y = y$. Formally, it is

$$\mathbb{P}[X = x \mid Y = y] = \frac{\mathbb{P}[X = x \cap Y = y]}{\mathbb{P}[Y = y]}.$$

The *conditional expectation* of X given Y , written as $\mathbb{E}[X \mid Y = y]$ is the expected value of X if you know that $Y = y$. Formally, it is the function

$$f(y) = \mathbb{E}[X \mid Y = y] = \sum_{x \in \Omega} x \mathbb{P}[X = x \mid Y = y].$$

- 3.A.** (2 PTS.) Prove that if X and Y are independent then $\mathbb{P}[X = x \mid Y = y] = \mathbb{P}[X = x]$.
- 3.B.** (2 PTS.) Let X_i be the number of elements in **QuickSelect** in the i th recursive call, when starting with $X_0 = n$ elements. Prove that $\mathbb{E}[X_i \mid X_{i-1}] \leq (3/4)X_{i-1}$.
- 3.C.** (2 PTS.) Prove that for any discrete random variables X and Y it holds $\mathbb{E}[\mathbb{E}[X|Y]] = \mathbb{E}[X]$.
- 3.D.** (10 PTS.) Prove that, in expectation, the i th recursive call made by **QuickSelect** has at most $(3/4)^i n$ elements in the sub-array it is being called on.
- 3.E.** (4 PTS.) Let X be a random variable that can take on only non-negative values. Assume that $\mathbb{E}[X] = \mu$, where $\mu > 0$ is a real number (for example, μ might be 0.01). Prove that $\mathbb{P}[X \geq 1] \leq \mu$.
- 3.F.** (10 PTS.) Using (D) and (E) prove that with probability $\geq 1 - 1/n^{10}$ the depth of the recursion of **QuickSelect** when executed on an array with n elements is bounded by $M = c \lg n$, where c is some sufficiently larger constant (figure out the value of c for which your claim holds!).
(Hint: Consider the random variable which is the size of the subproblem that **QuickSelect** handles if it reaches the problem in depth M , and 0 if **QuickSelect** does not reach depth M in the recursion.)

HW 8 (due Monday, at noon, April 8, 2018)

CS 473: Fundamental Algorithms, Spring 2013

1 (30 PTS.) Kris and the Climbing Vampires of CS @ Illinois

After Kris (your well-known professional climber) got tired of living in Colorado, he moved to Champaign, Illinois. Once the climbing community of this little town found out that Kris was living here, they felt honored and elected him president of “CS @ Illinois Climbing Club”, also known as “The Club”. The Computer Science Department at UIUC has n (semi-professional, vampire) climbers, who are members of The Club. In order for them to retain their membership, they need to participate in various competitions yearly. There are m competitions each year and the j 'th competition needs k_j participants from The Club. Kris, trying to fulfil his president duties, asked each member to volunteer to participate in a few competitions. Let $S_i \subseteq \{1, 2, \dots, m\}$ be the set of competitions that a computer scientist vampire climber i has volunteered for. A competition assignment consists of sets S'_1, S'_2, \dots, S'_n where $S'_i \subseteq \{1, 2, \dots, m\}$ is the set of competitions that club member i will participate in. A *valid* competition assignment has to satisfy two constraints: (i) for each club member i , $S'_i \subseteq S_i$, that is each member is only participating in competitions that he/she has volunteered for, and (ii) each competition j has k_j club members assigned to it, or in other words j occurs in at least k_j of the sets S'_1, S'_2, \dots, S'_n . Kris noticed that often there is no valid competition assignment because computer scientist vampire climbers naturally are inclined to volunteer for as few competitions as possible (stemming from the lazy nature of computer scientists and vampires). To overcome this, the definition of a valid assignment is relaxed as follows. Let ℓ be some integer. An assignment S'_1, S'_2, \dots, S'_n is now said to be valid if (i) $|S'_i - S_i| \leq \ell$ and (ii) each competition j has k_j club members participating at it. The new condition (i) means that a member i may participate up to ℓ competitions not on the list S_i that he/she volunteered for. Describe an algorithm to check if there is a valid competition assignment with the relaxed definition.

2 (40 PTS.) Augmenting Paths in Residual Networks.

You are given an integral instance G of network flow. Let C be the value of the maximum flow in G .

- 2.A. (8 PTS.) Given a flow f in G , and its residual network G_f , describe how to compute, as fast as possible, the highest capacity augmenting path flow from s to t . Prove the correctness of your algorithm.
- 2.B. (8 PTS.) Prove, that if the maximum flow in G_f has value T , then the augmenting path you found in (A) has capacity at least T/m .
- 2.C. (8 PTS.) Consider the algorithm that starts with the empty flow f , and repeatedly applies (A) to G_f (recomputing it after each iteration) until s and t are disconnected. Prove that this algorithm computes the maximum flow in G .
- 2.D. (8 PTS.) Consider the algorithm from (C), and the flow g it computes after m iterations. Prove that $|g| \geq C/10$ (here 10 is not tight).
- 2.E. (8 PTS.) Give a bound, as tight as possible, on the running time of your algorithm, as a function of n , m , and C .

3 (30 PTS.) Edge-Disjoint Paths.

You are given a directed graph $G = (V, E)$ and a natural number k .

- 3.A. We can define a relation $\rightarrow_{G,k}$ on pairs of vertices of G as follows. If $x, y \in V$ we say that $x \rightarrow_{G,k} y$ if there exist k mutually edge disjoint paths from x to y in G . Is it true that for every G and every

$k \geq 0$ the relation $\rightarrow_{G,k}$ is transitive? That is, is it always the case that if $x \rightarrow_{G,k} y$ and $y \rightarrow_{G,k} z$ then we have $x \rightarrow_{G,k} z$? Give a proof or a counterexample.

- 3.B.** Suppose that for each node v of G the number of edges into v is equal to the number of edges out of v . Let x, y be two nodes and suppose there exist k mutually edge-disjoint paths from x to y . Does it follow that there exist k mutually disjoint paths from y to x ? Give a proof or a counterexample.

60.1.10. Homework 9

HW 9 (due Monday, at noon, April 15, 2018)

CS 473: Fundamental Algorithms, Spring 2013

1 (50 PTS.) We want a proof!

The following question is long, but not very hard, and is intended to make sure you understand the following problems, and the basic concepts needed for proving NP-Completeness.

For each of the following problems, you are given an instance of the problem of size n . Imagine that the answer to this given instance is “yes”. Imagine that you need to convince somebody that indeed the answer to the given instance is yes – to this end, describe:

- (I) The format of the proof that the instance is correct.
- (II) A bound on the length of the proof (its have to be of polynomial length in the input size).
- (III) An efficient algorithm (as fast as possible [it has to be polynomial tie]) for verifying, given the instance and the proof, that indeed the given instance is indeed positive.

We solve the first such question, so that you understand what we want.^①

1.A. (0 PTS.)

Shortest Path

Instance: A weighted undirected graph G , vertices s and t and a threshold w .

Question: Is there a path between s and t in G of length at most w ?

Solution: A “proof” in this case would be a path π in G (i.e., a sequence of at most n vertices) connecting s to t , such that its total weight is at most w . The algorithm to verify this solution, would verify that all the edges in the path are indeed in the graph, the path starts at s and ends at t , and that the total weight of the edges of the path is at most w . The proof has length $O(n)$ in this case, and the verification algorithm runs in $O(n^2)$ time. if we assume graph is given to us using an adjacency lists.

1.B. (5 PTS.)

Independent Set

Instance: A graph G , integer k

Question: Is there an independent set in G of size k ?

^①We trust that the reader can by now readily translate all the following questions to questions about climbing vampires from Champaign. The reader can do this translation in their spare time for their own amusement.

1.C. (5 PTS.)

3Colorable

Instance: A graph G .

Question: Is there a coloring of G using three colors?

1.D. (5 PTS.)

TSP

Instance: A weighted undirected graph G , and a threshold w .

Question: Is there a TSP tour of G of weight at most w ?

1.E. (5 PTS.)

Vertex Cover

Instance: A graph G , integer k

Question: Is there a vertex cover in G of size k ?

1.F. (5 PTS.)

Subset Sum

Instance: S - set of positive integers, t : - an integer number (target).

Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

1.G. (5 PTS.)

3DM

Instance: X, Y, Z sets of n elements, and T a set of triples, such that $(a, b, c) \in T \subseteq X \times Y \times Z$.

Question: Is there a subset $S \subseteq T$ of n disjoint triples, s.t. every element of $X \cup Y \cup Z$ is covered exactly once.?

1.H. (5 PTS.)

Partition

Instance: A set S of n numbers.

Question: Is there a subset $T \subseteq S$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$?

1.I. (5 PTS.)

SET COVER

Instance: (S, \mathcal{F}, k) :

S : A set of n elements

\mathcal{F} : A family of subsets of S , s.t. $\bigcup_{X \in \mathcal{F}} X = S$.

k : A positive integer.

Question: Are there k sets $S_1, \dots, S_k \in \mathcal{F}$ that cover S . Formally, $\bigcup_i S_i = S$?

1.J. (5 PTS.)

CYCLE HATER.

Instance: An undirected graph $G = (V, E)$, and an integer $k > 0$.

Question: Is there a subset $X \subseteq V$ of at most k vertices, such that all cycles in G contain at least one vertices of X .

1.K. (5 PTS.)

CYCLE LOVER.

Instance: An undirected graph $G = (V, E)$, and an integer $k > 0$.

Question: Is there a subset $X \subseteq V$ of at most k vertices, such that all cycles in G contain at least two vertices of X .

2 (50 PTS.) Beware of Greeks bearing gifts.^②

The woodland deity, the brother of the *induction fairy*, came to visit you on labor day, and left you with two black boxes.

2.A. (25 PTS.) The first black-box can solve **Partition** in polynomial time (note that this black box solves the decision problem). Let S be a given set of n integer numbers. Describe a polynomial time algorithm that computes, using the black box, a partition of S if such a solution exists. Namely, your algorithm should output a subset $T \subseteq S$, such that

$$\sum_{s \in T} s = \sum_{s \in S \setminus T} s.$$

2.B. (25 PTS.) The first box was a black box for solving **Subgraph Isomorphism**.

Subgraph Isomorphism

Instance: Two graphs, $G = (V_1, E_1)$ and $H = (V_2, E_2)$.

Question: Does G contain a subgraph *isomorphic* to H , that is, a subset $V \subseteq V_1$ and a subset $E \subseteq E_1$ such that $|V| = |V_2|$, $|E| = |E_2|$, and there exists a one-to-one function $f : V_2 \rightarrow V$ satisfying $\{u, v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E$?

Show how to use this black box, to compute the subgraph isomorphism (i.e., you are given G and H , and you have to output f) in polynomial time. (You can assume that a call to this black box, takes polynomial time.)

60.1.11. Homework 10

HW 10 (due Monday, at noon, April 22, 2018)

CS 473: Fundamental Algorithms, Spring 2013

^②The expression “beware of Greeks bearing gifts” is based on Virgil’s Aeneid: “Quidquid id est, timeo Danaos et dona ferentes”, which means literally “Whatever it is, I fear Greeks even when they bring gifts.”

1 (45 PTS.) Coolest path

After Kris moved to Illinois, he started dating a computer scientist vampire climber girl Oinoe that he met at the climbing gym. Kris is a multi-dimensional personality and apart from climbing he also enjoys and excels at snowboarding. To his pleasant surprise, Oinoe was also a decent snowboarder and so they decided to take a trip to the backcountry mountains of New Zealand. They packed a few days worth of supplies, got their snowboarding gear on and hired a helicopter from the mountain base (hereafter referred to as the Base) to drop them off at the peak of a far away mountain called Death Peak. Kris and Oinoe's goal was to make their way from Death Peak back to the Base. The mountain was covered in powder and there was nobody else on it. The only thing that would reassure them that they were on a path back to the Base were some red poles planted in the snow at various parts of the mountain, called Stations. We can view the mountain as an undirected graph $G = (V, E)$ where each node is a Station and an edge (u, v) indicated that one can travel directly from station u to station v by snowboard (Kris and Oinoe carried with them a new kind of snowboard which was enhanced by a motor and allowed them to travel through flat parts of the mountain easily). The Death Peak is represented by a node s and the Base by a node t . Each edge e has a length $l_e \geq 0$ which represents distance from one Station to another. Also, some edges represent paths that are higher risk than others, in the sense that they are more avalanche-prone, have more tree-wells or obstacles along the way. So each edge e also has an integer risk $r_e \geq 0$, indicating the expected amount of damage in their health or equipment, if one traverses this edge.

It would be safest to travel by traversing the ridge of the mountain till they reach the end of the Sierra and then go downhill a very easy slope, but that would take them many days and they will run out of food. It would be fastest to just go down the steepest slope from Death Peak to the base of the mountain but that is very dangerous to create an avalanche. In general, for every path p from s to t , we define its total length to be the sum of the lengths of all its edges and its total risk to be the sum of the risks of all its edges.

Kris and Oinoe are looking for a complex type of shortest path in that graph that they name the *Coolest Path*: they need to get from s to t along a path whose total length and total risk is reasonably small. In concrete terms, the problem they want to solve is the following: given a graph with lengths and risks as above and integers L and R , is there a path from s to t whose total length is at most L and whose total risk is at most R ?

Show that the Coolest Path problem is NP-Complete.

2 (30 PTS.) Brooklyn is learning how to speak

Your friend's pre-school age daughter Brooklyn has recently learned to spell some simple words. To help encourage this, her parents got her a colorful set of refrigerator magnets featuring the letters of the alphabet (some number of copies of each letter), and the last time you saw her, the two of you spent a while arranging the magnets to spell out words that she knows.

Somehow with you and Brooklyn, things end up getting more elaborate than originally planned, and soon the two of you were trying to spell out words so as to use up all the magnets in the full set—that is, picking words that she knows how to spell, so that they were all spelled out, each magnet was participating in the spelling of exactly one word. Multiple copies of words are okay here.

This turned out to be pretty difficult, and it was only later that you realized a plausible reason for this. Suppose we consider a general version of the problem of *Using Up All the Refrigerator Magnets*, where we replace the English alphabet by an arbitrary collection of symbols, and we model Brooklyn's vocabulary as an arbitrary set of strings over this collection of symbols. The goal is the same. Prove that the problem of *Using Up All the Refrigerator Magnets* is NP-Complete.

3 (25 PTS.) Path Selection

Consider the following problem. You are managing a communication network, modeled by a directed

graph $G = (V, E)$. There are c users who are interested in making use of this network. User i (for each $i = 1, 2, \dots, c$) issues a *request* to reserve a specific path P_i in G on which to transmit data.

You are interested in accepting as many of these path requests as possible, subject to the following restriction: if you accept both P_i and P_j , then P_i and P_j can not share any nodes.

Thus the *Path Selection Problem* asks: Given a directed graph $G = (V, E)$, a set of requests P_1, \dots, P_c -each of which must be a path in G - and a number k , is it possible to select at least k of the paths so that no two of the selected paths share any nodes?

Find a polynomial time algorithm for Path Selection or show that the problem is NP-Complete.

60.2. Midterm 1

CS 473: Fundamental Algorithms, Spring 2013

Midterm 1: February 19, 2013

12:30-13:45 section in Everitt Lab 151

14:00-15:15 section in Loomis 151

1 RECURRENCES. (10 PTS.)

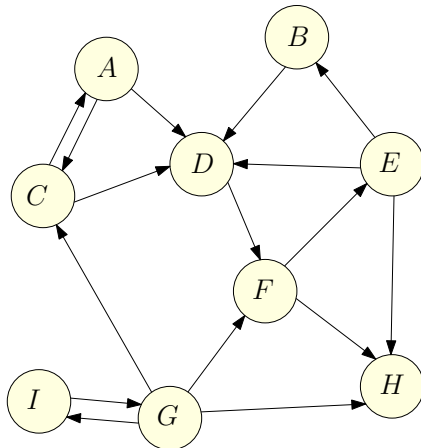
Give a tight asymptotic bound for the following recurrences. No justification necessary.

1.A. $A(n) = A(\sqrt{n}) + A(n/2) + A(n/3) + n$, for $n \geq 2$ and $A(1) = 1$.

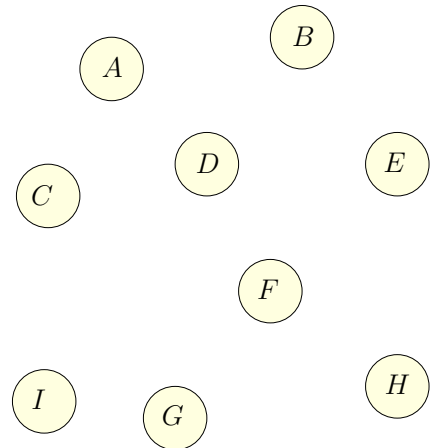
1.B. $B(n) = 4B(n/3) + \log \log n$, for $n \geq 2$ and $B(n) = 1$ for $0 \leq n < 2$.

2 SHORT QUESTIONS. (15 PTS.)

- 2.A. (5 PTS.) For the following directed graph, indicate what are the pre/post numbers for each vertex, when the DFS starts from the vertex A. To make things unique, assume that the DFS visits the neighbors of a vertex in alphabetical increasing order. Also, draw the resulting DFS tree/forest on the right side.

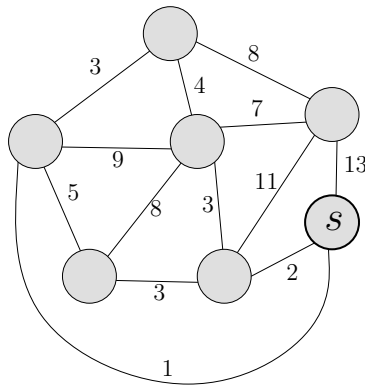


Vertex	pre #	post #
A		
B		
C		
D		
E		
F		
G		
H		
I		



- 2.B. (5 PTS.) List the strong connected components in the above graph.

- 2.C. (5 PTS.) For the graph below (yes, it is undirected) draw the shortest path tree rooted at s , and for each node indicate its distance from s .



3 SORTING UNIMODAL SEQUENCES. (25 PTS.)

A sequence of distinct numbers x_1, x_2, \dots, x_n is increasing if $x_i < x_{i+1}$, for $1 \leq i < n$. Similarly it is decreasing if $x_i > x_{i+1}$ for $1 \leq i < n$. It is **unimodal** if there is an index p , $1 \leq p \leq n$ such that either:

- (I) The subsequence x_1, x_2, \dots, x_p is an increasing sequence and the subsequence x_p, x_{p+1}, \dots, x_n is a decreasing sequence.
- (II) The subsequence x_1, x_2, \dots, x_p is a decreasing sequence and the subsequence x_p, x_{p+1}, \dots, x_n is an increasing sequence.

Observe that increasing (or decreasing) sequences are also unimodal. For example,

$$-10, -3, 2, 5, 25, 13, 12, -1$$

is a unimodal sequence with a maximum at 25. Similarly, the sequence

$$10, 3, -20, -13, -12, 1$$

is a unimodal sequence with a minimum at -20 .

Suppose you are given a sequence of n distinct numbers stored in an array X ($X[i]$ holds x_i) and are told that it is a unimodal sequence. Describe an algorithm that sorts the sequence in increasing order and runs in time $O(n)$.

4 FOREVER ALONE VAMPIRE. (25 PTS.)

There are n vampires in Champaign^①. You know for each vampire all the vampires that he/she/it had bitten (you can assume that, overall, there are m bites known). A vampire v is **popular** if there is a sequence of *distinct* vampires v_1, \dots, v_k , such that v had bitten v_1 , v_1 had bitten v_2 , ..., v_{k-1} had bitten v_k , and v_k had bitten v (note, that k can be 1).

Describe an algorithm, as fast as possible, that outputs all the lonely vampires in Champaign (a vampire is lonely if it is not popular, naturally).

5 SHORTEST CYCLE. (25 PTS.)

You are given a directed weighted graph G (the weights are positive). Design an algorithm that finds the shortest cycle in G . How fast is your algorithm? (The faster, the better.)

6 COMPUTE A GOOD VERTEX. (25 PTS.)

Given an unweighted rooted tree T with n nodes, the distance between two vertices u and v in T is the number of edges in the unique path between u and v in T . The **price** of a vertex v is the total sum of the distances of v to all the vertices of T .

^①If you live in Urbana then you are safe.

- 6.A. Describe an algorithm, as fast as possible, for computing the price of the root of T . How fast is your algorithm? (The faster, the better.)
- 6.B. In a tree T , a *separator* is a node v such that if we remove it T breaks into a collection of trees, each one of them having at most $n/2$ vertices. Describe, an algorithm, as fast as possible, that computes a separator in T (such a separator always exists). How fast is your algorithm? (The faster, the better.)

60.3. Midterm 2

CS 473: Fundamental Algorithms, Spring 2013

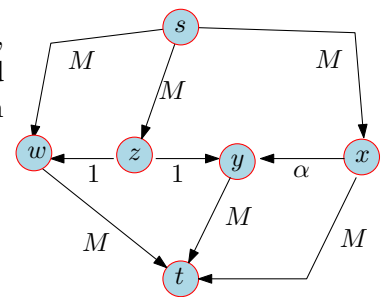
Midterm 2: April 2, 2013

12:30-13:45 section in Noyes Laboratory 217

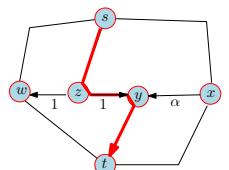
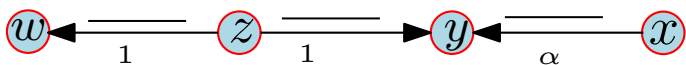
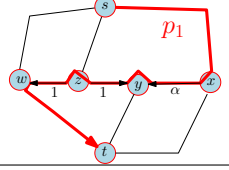
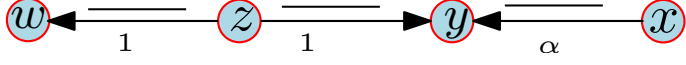
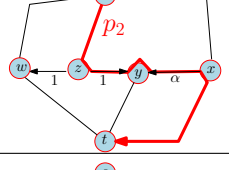
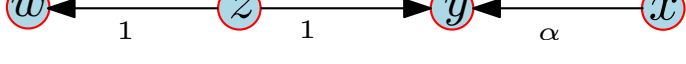
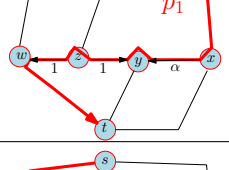
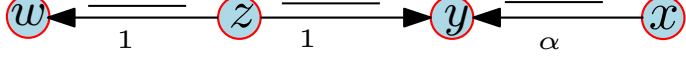
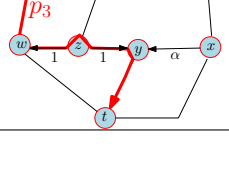
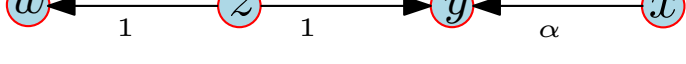
14:00-15:15 section in Mumford Hall 103

1 FLOW, FLOW, FLOW.

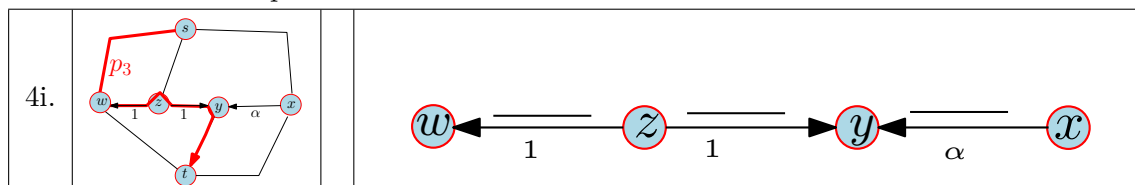
Consider the network flow on the right. Here M is a large positive integer, and $\alpha = (\sqrt{5} - 1)/2$. One can verify that (i) $\alpha < 1$, (ii) $\alpha^2 = 1 - \alpha$, and (iii) $1 - \alpha < \alpha$. We will be interested in running `mtdFordFulkerson` on this graph.



- 1.A. (2 PTS.) What is the value of the maximum flow in this network?
- 1.B. (3 PTS.) Prove that $\alpha^i - \alpha^{i+1} = \alpha^{i+2}$, for all $i \geq 0$.
- 1.C. (10 PTS.) We next specify the sequence of augmenting paths used by the algorithm. In the table below, specify the residual capacities for the designated edges after each step (the residual graph would have also other edges which we do not care about). We start here from the 0 flow. Please simplify the numbers involved as much as possible, using the relation from (B). Hint: All the numbers you have to write below are either 0, 1 or α^i , for some i . You want to be careful about the calculations - it is easy to make mistakes here.

step	augmenting path	amount pushed	residual capacity
0.		1	
1.		α	
2.		α	
3.		α^2	
4.		α^2	

- 1.D. (5 PTS.) Imagine that we now repeat steps 1,2,3,4 above (i.e., we augment along p_1, p_2, p_1, p_3 repeatedly. What would be the residual capacities in the graph after the i th such repetition, starting with the results in the step 4 above?



- 1.E. (5 PTS.) How many iterations would it take this algorithm to terminate, if we repeat the steps in (D)?

2 TAKE THESE INTERVALS. (25 PTS.)

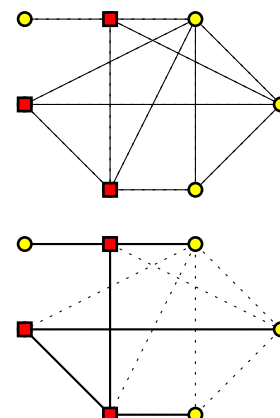
You are given a set of n intervals I_1, \dots, I_n , where $I_j = [x_j, y_j]$, for $j = 1, \dots, n$. Describe an algorithm that colors the intervals with a minimum number of colors. Here, two intervals that intersect must be assigned different colors. What is the running time of your algorithm? (Faster is better – your algorithm must have polynomial running time.)

Prove the correctness of your algorithm.

3 SPANNING IT ROBUSTLY. (25 PTS.)

You are given an undirected weighted graph $G = (V, E)$ with n vertices and m edges (think about it as describing a network). We are interested in computing a spanning tree for G . However, you are given a set U of “untrustable” vertices – for example, on the graph depicted on the right, the “circle” nodes are untrustable. Such an untrustable vertex might be suddenly deleted from the graph (and all the edges adjacent to it). You

want to build a spanning tree for G , such that even if this happens, the remaining spanning tree, is still a spanning tree for the remaining graph. We will refer to such a spanning tree as a **robust spanning tree**. For example, the figure on the right depicts one possible robust spanning tree for the example shown above.



- 3.A. (10 PTS.) Describe an algorithm, as efficient as possible, for computing a robust spanning tree if it exists. What is the running time of your algorithm? (Faster is better.)
- 3.B. (15 PTS.) Describe an algorithm, as efficient as possible, for computing the cheapest robust spanning tree, if such a tree exists. What is the running time of your algorithm?

4 INDEPENDENT VAMPIRES.

As you know by now, there are n vampires in Champaign, IL. Furthermore, you know all the m pairs of vampires that know each other. The climbing vampire club of Champaign had decided to organize a rock climbing trip to the Netherlands, and they would like to choose as many vampires as possible to this trip, under the constraint that no two vampires in the group know each other.

To this end, the club had ordered the vampires in a random order v_1, \dots, v_n . In the i th iteration, the vampire v_i is invited to the trip, if none of the vampires it knows are already invited to the trip.

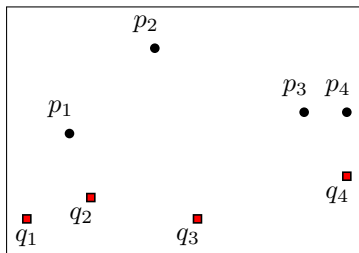
- 4.A. (5 PTS.) What is the probability of a vampire that knows all the other $n - 1$ vampires to be invited to the trip. Explain your answer.

- 4.B. (5 PTS.) Give a lower bound on the probability of a vampire that knows k other vampires to be invited to the trip. (Your lower bound should be a function of k only – computing the exact probability is hard here.) [Hint: Think about (A).]
- 4.C. (10 PTS.) Assume that a vampire $v \in V$ knows $d(v)$ other vampires, where V denotes the set of vampires. Give a closed form simple formula which is a lower bound on the expected number of vampires that are going to be in the trip. (Hint: Use (B).)
- 4.D. (5 PTS.) Given a lower bound on the expected number of vampires going on the trip, if every vampire knows exactly k other vampires? (Your lower bound should be a function of n and k .)

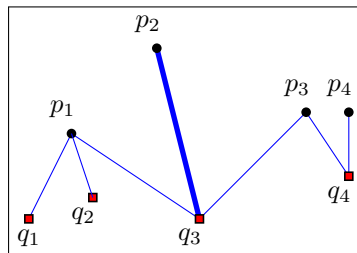
5 DISTANCE BETWEEN POINT SETS. (25 PTS.)

You are given two sequences of points in the plane: p_1, \dots, p_n and q_1, \dots, q_n . You start with two players standing on p_1 and q_1 , respectively. At each step, exactly one of the players can move to the next point on its sequence (or stay where it is, but the other player must move then). In the end of the process, the players need to be both located at p_n and q_n , respectively. During this process, if the two players are located at p_i and q_j then their *distance* is $\|p_i - q_j\|$.

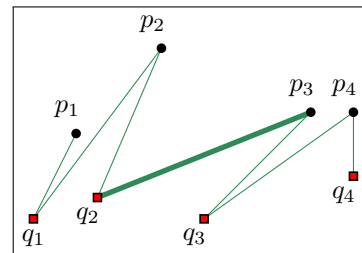
Given a valid solution to this problem (i.e., the sequence of moves), the *value* of the solution is the maximum distance of the two players (from each other) at any point in time during the process. See the following example:



The input.



One possible solution: $(p_1, q_1) \Rightarrow (p_1, q_2) \Rightarrow (p_1, q_3) \Rightarrow (p_2, q_3) \Rightarrow (p_3, q_3) \Rightarrow (p_3, q_4) \Rightarrow (p_4, q_4)$. The value of this solution is the distance between p_2 and q_3 .



Another possible solution: $(p_1, q_1) \Rightarrow (p_2, q_1) \Rightarrow (p_2, q_2) \Rightarrow (p_3, q_2) \Rightarrow (p_3, q_3) \Rightarrow (p_4, q_3) \Rightarrow (p_4, q_4)$. The value of this solution is the distance between p_3 and q_2 . The other solution is slightly better, as its value is smaller.

Describe an algorithm, as fast as possible, that computes the optimal solution (i.e., the solution with minimum value). Describe how to modify your algorithm so it outputs the sequence of moves realizing the optimal solution. What is the running time of your algorithm?

60.4. Final

CS 473: Fundamental Algorithms, Spring 2013

Final Exam: 7pm-10pm, May 3, 2018

David Kinley Hall, Room 114

1 Multiple choice. (8 PTS.)

For each of the questions below choose the most appropriate answer. No **IDK** credit for this question!

- 1.A. Given a graph G . Deciding if there is an independent set X in G , such that $G \setminus X$ (i.e., the graph G after we remove the vertices of X from it) is bipartite can be solved in polynomial time.

False: True: Answer depends on whether $P = NP$:

- 1.B. Consider any two problems X and Y both of them in **NPC**. There always exists a polynomial time reduction from X to Y .

False: True: Answer depends on whether $P = NP$:

- 1.C. Given a graph represented using adjacency lists, it can be converted into matrix representation in linear time in the size of the graph (i.e., linear in the number of vertices and edges of the graph).

False: True: Answer depends on whether $P = NP$:

- 1.D. Given a **2SAT** formula F , there is always an assignment to its variables that satisfies at least $(7/8)m$ of its clauses.

False: True: Answer depends on whether $P = NP$:

- 1.E. Given a graph G , deciding if contains a clique made out of 165 vertices is **NP-COMplete**. False: True: Answer depends on whether $P = NP$:
-

- 1.F. Given a network flow G with lower bounds and capacities on the edges (say all numbers are integers that are smaller than n). Assume f and g are two different maximum flows in G that complies with the lower bounds and capacity constraints. Then, the flow $0.7f + 0.3g$ is always a valid maximum flow in G . False: True:
-

- 1.G. Given a directed graph G with positive weights on the edges, and a number k , finding if there is simple path in G from s to t (two given vertices of G) with weight $\geq k$, can be done in polynomial time.

False: True: Answer depends on whether $P = NP$:

- 1.H. Given a directed graph G with (positive or negative) weights on its edges, computing the shortest walk from s to t in G can be done in polynomial time.

False: True: Answer depends on whether $P = NP$:

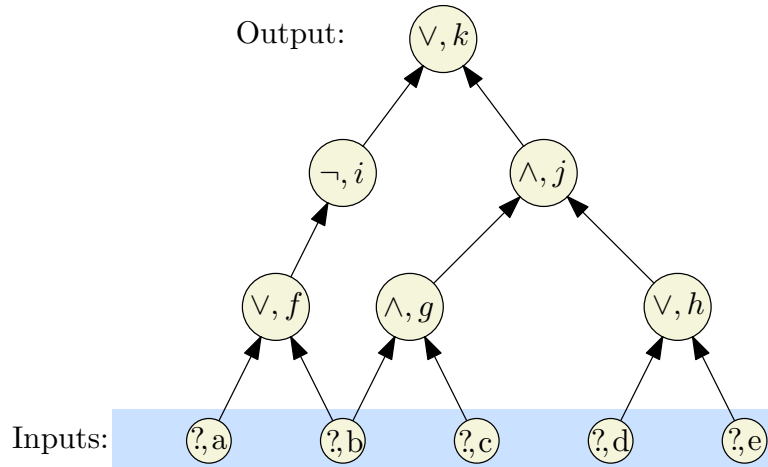
2 Short Questions. (16 PTS.)

- 2.A. (8 PTS.) Give a tight asymptotic bound for each of the following recurrences.

(I) (4 PTS.) $A(n) = A(n - 3 \lceil \log n \rceil) + A(\lceil \log n \rceil) + \log n$, for $n > 2$
and $A(1) = A(2) = 1$.

(II) (4 PTS.) $B(n) = 12B(\lfloor n/4 \rfloor) + B(\lfloor n/2 \rfloor) + n^2$, for $n > 10$ and $B(i) = 1$ for $1 \leq i \leq 10$.

- 2.B. (8 PTS.) Convert the following boolean circuit (i.e., an instance of **Circuit-SAT**) into a CNF formula (i.e., an instance of SAT) such that the resulting formula is satisfiable if and only if the circuit sat instance is satisfiable. Use $x_a, x_b, x_c, x_d, \dots$ as the variable names for the corresponding gates in the drawing. (You may need additional variables.) Note, that a node (\wedge, g) in the figure below denotes an and gate, where g is its label.



3 Balancing vampires. (24 PTS.)

Sadly, there are n vampires p_1, \dots, p_n in Champaign. The i th vampire has a score $w_i \geq 0$ describing how well it can climb mountains. You want to divide the vampires into two teams, and you want the division of teams to be as fair as possible. The score of a team is the sum of the scores of all the vampires in that team. We want to minimize the differences of the scores of the two teams. Assume that for all i , $w_i \leq W$.

- 3.A. (12 PTS.) Given integers $\alpha, \beta \geq 0$, and T_α, T_β , such that $\alpha + \beta = n$, describe an algorithm, as fast as possible, to compute the partition into two teams, such that the first team has α players of total score T_α , and the second team has β players with total score T_β . What is the running time of your algorithm? (For any credit, it has to be polynomial in n and W .)
(To simplify things, you can solve the decision version problem first, and describe shortly how to modify it to yield the desired partition.)
- 3.B. (4 PTS.) Describe an algorithm, as fast as possible, to compute the scores of the two teams in an optimal division that is as balanced as possible, when requiring that the two teams have exactly the same number of players (assume n is even). What is the running time of your algorithm?
- 3.C. (8 PTS.) State **formally** the decision version of the problem in (B), and prove that it is **NP-COMplete**. (There are several possible solutions for this part – pick the one you find most natural. Note, that the teams must have the same number of players.)

4 MAX Cut and MAX 2SAT. (13 PTS.)

The **Max CUT** problem is the following:

MAX Cut

Instance: Undirected graph G with n vertices and m edges, and an integer k .
Question: Is there an undirected cut in G that cuts at least k edges?

MAX 2SAT

Instance: A 2CNF formula F , and an integer k .

Question: Is there a truth assignment in F that satisfies at least k clauses.

You are given that **MAX Cut** is **NP-COMplete**. Prove that **MAX 2SAT** is **NP-COMplete** by a reduction to/from **MAX Cut** (be sure to do the reduction in the right direction! [and please do not ask us what is the right direction – this is part of the problem]).

Hint: Think about how to encode a cut, by associating a boolean variable with each vertex of the graph. It might be a good idea to verify your answer by considering a graph with two vertices and a single edge between them and checking all possibilities for this case.

5 Billboards are forever.

(From discussion 5.) (13 PTS.)

Consider a stretch of Interstate-57 that is m miles long. We are given an ordered list of mile markers, x_1, x_2, \dots, x_n in the range 0 to m , at each of which we are allowed to construct billboards (suppose they are given as an array $X[1 \dots n]$). Suppose we can construct billboards for free, and that we are given an array $R[1 \dots n]$, where $R[i]$ is the revenue we would receive by constructing a billboard at location $X[i]$. Given that state law requires billboards to be at least 5 miles apart, describe an algorithm, as fast as possible, to compute the maximum revenue we can acquire by constructing billboards.

What is the running time of your algorithm? (For full credit, your algorithm has to be as fast as possible.)

6 Best edge ever. (13 PTS.)

(Similar to homework problem.)

You are given a directed graph G with n vertices and m edges. For every edge $e \in E(G)$, there is an associated weight $w(e) \in \mathbb{R}$. For a path (not necessarily simple) π in G , its **quality** is $W(\pi) = \max_{e \in \pi} w(e)$. We are interested in computing the highest quality walk in G between two given vertices (say s and t). Either **prove** that computing such a walk is **NP-HARD**, or alternatively, provide an algorithm (and **prove** its correctness) for this problem (the algorithm has to be as fast as possible – what is the running time of your algorithm?).

7 Dominate this. (13 PTS.)

(Similar problems were covered in class/homework/discussion.)

You are given a set of intervals $\mathcal{I} = \{I_1, \dots, I_n\}$ on the real line (assume all with distinct endpoints) – they are given in arbitrary order (i.e., you can not assume anything on the ordering). Consider the problem of finding a set of intervals $\mathcal{K} \subseteq \mathcal{I}$, as small as possible, that dominates all the other intervals. Formally, \mathcal{K} **dominates** \mathcal{I} , if for every interval $I \in \mathcal{I}$, there is an interval $K \in \mathcal{K}$, such that I intersects K .

Describe an algorithm (as fast as possible) that computes such a minimal dominating set of \mathcal{I} . What is the running time of your algorithm? Prove the correctness of your algorithm.

8 Network flow. (13 PTS.)

(Similar problems were covered in class/homework/discussion.)

You are given a network flow G (with integer capacities on the edges, and a source s and a sink t), and a maximum flow f on it (you can assume f is integral). You want increase the maximum flow in G by one unit by applying a single augmenting path to f . Naturally, to be able to do that, you must increase the capacity of some of the edges of G . In particular, for every edge $e \in E(G)$, there is an associated cost $\text{cost}(e)$ of increasing its capacity by one unit. Describe an algorithm, that computes (as fast as possible), the cheapest collection of edges of G , such that if we increase the capacity on each of these edges by 1, then one can find an augmenting path to f that increases its flow by one unit. How fast is your algorithm?

Provide an argument that explains why your algorithm is correct.

Chapter 61

Fall 2014: CS 573 – Graduate algorithms

61.1. Homeworks

61.1.1. Homework 0

1 (60 PTS.) The numbers dance.

1.A. (30 PTS.) The input is a multiset X of n positive integer numbers. Consider the following famous algorithm:

```
PlayItBen( $X$ ):  
  while  $X$  contains more than two elements do  
    two distinct elements  $x_1, x_2$  are chosen arbitrarily from  $X$ ,  
    such that  $x_1 \leq x_2$   
    if  $x_1 = x_2$  or  $x_1 + 1 = x_2$  then  
       $X \leftarrow (X \setminus \{x_1, x_2\}) \cup \{x_1 + x_2\}$   
    else  
       $X \leftarrow (X \setminus \{x_1, x_2\}) \cup \{x_1 + 1, x_2 - 1\}$ 
```

Prove (maybe using induction, but you do not have to) that **PlayItBen** always terminates.

1.B. (30 PTS.) (Harder.) Let $N = \sum_{x \in X} x$, and let $n = |X|$. Provide an upper bound, as tight as possible, using n and N on the running time of **PlayItBen**.

2 (40 PTS.) Random walk.

A *random walk* is a walk on a graph G , generated by starting from a vertex $v_0 = v \in V(G)$, and in the i -th stage, for $i > 0$, randomly selecting one of the neighbors of v_{i-1} and setting v_i to be this vertex. A walk v_0, v_1, \dots, v_m is of length m .

2.A. (20 PTS.) For a vertex $u \in V(G)$, let $P_u(m, v)$ be the probability that a random walk of length m , starting from u , visits v (i.e., $v_i = v$ for some i).

Prove that a graph G with n vertices is connected, if and only if, for any two vertices $u, v \in V(G)$, we have $P_u(n-1, v) > 0$.

2.B. (20 PTS.) Prove that a graph G with n vertices is connected if and only if for any pair of vertices $u, v \in V(G)$, we have $\lim_{m \rightarrow \infty} P_u(m, v) = 1$.

61.1.2. Homework 1

1 (10 PTS.) Poly time subroutines can lead to exponential algorithms.

Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

2 (70 PTS.) Beware of algorithms carrying oracles.

Consider the following optimization problems, and for each one of them do the following:

- (I) (2 PTS.) State the natural decision problem corresponding to this optimization problem.
- (II) (3 PTS.) Either: (A) prove that this decision problem is **NP-COMplete** by showing a reduction from one of the **NP-COMplete** problems seen in class (if you already seen this problem in class state “seen in class” and move on with your life). (B) Alternatively, provide an efficient algorithm to solve this problem.
- (III) (5 PTS.) Assume that you are given an algorithm that can solve the decision problem in polynomial time. Show how to solve the original optimization problem using this algorithm in polynomial time.

Prove that the following problems are **NP-COMplete**.

2.A. (10 PTS.)

NOT SET COVER

Instance: Collection C of subsets of a finite set S .

Target: Compute the maximum k , and the sets S_1, \dots, S_k in C , such that $S \not\subseteq \cup_{i=1}^k S_i$.

2.B. (10 PTS.)

MAX BIN PACKING

Instance: Finite set U of items, size $s(u) \in \mathbb{Z}^+$ for $u \in U$, an integer bin capacity B .

Target: Compute the maximum k , and a partition of U into disjoint sets U_1, \dots, U_k , such that the sum of the sizes of the items inside each U_i is B or more.

2.C. (10 PTS.)

DOUBLE HITTING SET

Instance: A *ground set* $U = \{1, \dots, n\}$, and a set $\mathcal{F} = \{U_1, \dots, U_m\}$ of subsets of U .

Target: Find the smallest set $S' \subseteq U$, such that S' hits all the sets of \mathcal{F} at least twice. Specifically, $S' \subseteq U$ is a *double hitting set* if for all $U_i \in \mathcal{F}$, we have that S' contains at least two element of U_i .

2.D. (10 PTS.)

Min Leaf Spanning Tree

Instance: Graph $G = (V, E)$.

Target: Compute the spanning tree \mathcal{T} in G where the number of vertices in \mathcal{T} of degree one is minimized.

2.E. (10 PTS.)

Cover by paths (edge disjoint).

Instance: Graph $G = (V, E)$.

Target: Compute the minimum number k of paths π_1, \dots, π_k that are edge disjoint, and their union cover all the edges in G .

2.F. (10 PTS.)

Cover by paths (vertex disjoint).

Instance: Graph $G = (V, E)$.

Target: Compute the minimum number k of paths π_1, \dots, π_k that are vertex disjoint, and their union cover all the vertices in G .

2.G. (10 PTS.)

Partition graph into not so bad, and maybe even good, sets (PGINSBMEGS).

Instance: Graph $G = (V, E)$ and k .

Target: Compute the partition of V into k sets V_1, \dots, V_k , such that the number of edges uv of G that have distinct indices i and j , such that $u \in V_i$, and $v \in V_j$ is maximized.

3 (20 PTS.) Independence.

Let $G = (V, E)$ be an undirected graph over n vertices. Assume that you are given a numbering $\pi : V \rightarrow \{1, \dots, n\}$ (i.e., every vertex have a unique number), such that for any edge $ij \in E$, we have $|\pi(i) - \pi(j)| \leq 20$.

Either prove that it is **NP-HARD** to find the largest independent set in G , or provide a polynomial time algorithm.

61.1.3. Homework 2

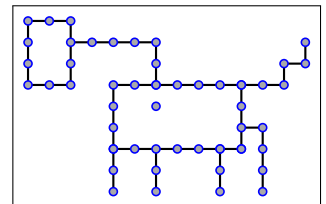
1 (40 PTS.) Breakable graphs.

In the following, let c_1, c_2 be some absolute, sufficiently large constants. Consider a graph $G = (V, E)$ with n vertices. A subset $Y \subseteq V$ is an *isolator* if:

- (I) In the graph $G \setminus Y$ (i.e., the induced graph on $V \setminus Y$) is disconnected, and every connected component of this graph has at most $(3/4)n$ vertices.
- (II) $|Y| \leq c_2 |n|^{2/3}$ (i.e., Y is significantly smaller than n).

1.A. (10 PTS.)

A *grid* graph is a graph where the vertices are points (x, y) in the plane, where x, y are integer numbers, and two vertices (x, y) and (x', y') can be connected only if $|x - x'| + |y - y'| = 1$. See picture on the right. Prove that there is always an isolator in such a graph. Your proof must be self contained, elementary and *short*.



(Hint: Start thinking about the case where the vertices of G are contained in $\llbracket N \rrbracket \times \llbracket N \rrbracket$, where $N = 4 \lceil n^{2/3} \rceil$ and $\llbracket N \rrbracket = \{1, \dots, N\}$. Then solve the case the vertices of G are contained in $\llbracket N \rrbracket \times \llbracket n \rrbracket$, and finally solve the general case where the vertices are contained in $\llbracket n \rrbracket \times \llbracket n \rrbracket$.)

- 1.B.** (10 PTS.) A graph is *breakable*, if for any subset $X \subseteq V$, of size at least c_1 , we have that there is an isolator in the induced graph G_X , and furthermore the isolator can be computed in polynomial time in X .

Prove that in a breakable graph, there is always a vertex of constant degree.

(This part is pretty hard, so do not be surprised if you can not do this part. If can not do this part, just assume it is correct, and continue to the next part of the question.)

- 1.C.** (10 PTS.) Given a breakable graph G , provide a polynomial time constant approximation algorithm for the largest independent set in G .
- 1.D.** (10 PTS.) For an arbitrary fixed $\varepsilon > 0$, provide a polynomial time algorithm that computes $(1 - \varepsilon)$ -approximation to the largest independent set in a breakable graph. What is the running time of your algorithm? Prove both the bound on the running time of your algorithm, and the quality of approximation it provides.
- (This part is also hard, do not be surprised if you can not do this part.)

2 (30 PTS.) Greedy algorithm does not work for coloring. Really.

Let G be a graph defined over n vertices, and let the vertices be ordered: v_1, \dots, v_n . Let G_i be the induced subgraph of G on v_1, \dots, v_i . Formally, $G_i = (V_i, E_i)$, where $V_i = \{v_1, \dots, v_i\}$ and

$$E_i = \left\{ uv \in E \mid u, v \in V_i \text{ and } uv \in E(G) \right\}.$$

The greedy coloring algorithm, colors the vertices, one by one, according to their ordering. Let k_i denote the number of colors the algorithm uses to color the first i vertices.

In the i th iteration, the algorithm considers v_i in the graph G_i . If all the neighbors of v_i in G_i are using all the k_{i-1} colors used to color G_{i-1} , the algorithm introduces a new color (i.e., $k_i = k_{i-1} + 1$) and assigns it to v_i . Otherwise, it assign v_i one of the colors $1, \dots, k_{i-1}$ (i.e., $k_i = k_{i-1}$).

Give an example of a graph G with n vertices, and an ordering of its vertices, such that even if G can be colored using $O(1)$ (in fact, it is possible to do this with two) colors, the greedy algorithm would color it with $\Omega(n)$ colors. (Hint: consider an ordering where the first two vertices are not connected.)

3 (30 PTS.) Maximum Clique

Let $G = (V, E)$ be an undirected graph. For any $k \geq 1$, define $G^{(k)}$ to be the undirected graph $(V^{(k)}, E^{(k)})$, where $V^{(k)} = V \times V \times \dots \times V$ is the set of all ordered k -tuples of vertices from V and $E^{(k)}$ is defined so that (v_1, v_2, \dots, v_k) is adjacent to (w_1, w_2, \dots, w_k) if and only if for each i (for $i = 1, \dots, k$) either vertex v_i is adjacent to w_i in G , or else $v_i = w_i$.

- 3.A.** (10 PTS.) Prove that the size of the maximum clique in $G^{(k)}$ is equal to the k th power of the size of the maximum clique in G . That is, if the largest clique in G has size α , then the largest clique in $G^{(k)}$ is α^k , and vice versa.
- 3.B.** (10 PTS.) Show an algorithm that is given a clique of size β in $G^{(k)}$ and outputs a clique of size $\lceil \beta^{1/k} \rceil$ in G .
- 3.C.** (5 PTS.) Argue that if there is an c -approximation algorithm for maximum clique (i.e., it returns in polynomial time a clique of size $\geq \text{opt}/c$) then there is a polynomial time $c^{1/k}$ -approximation algorithm for maximum clique, for any constant k . What is the running time of your algorithm, if the running time of the original algorithm is $T(n)$. (Hint: use (A) and (B).)
- 3.D.** (5 PTS.) Prove that if there is a constant approximation algorithm for finding a maximum-size clique, then there is a polynomial time approximation scheme for the problem.^①

^①Can one prove that there is FPTAS in this case? I do not know.

CS 573: Graduate algorithms, Fall 2015
Homework 3, due Monday, November 3, 23:59:59, 2018
Version 1.01
Required Problems
1 (20 PTS.) Euclidean three dimensional matching.

You are given three disjoint sets R, G, B of n points in the plane. Our purposes is to output a set \mathcal{T} of n disjoint triplets of points $(r_i, g_i, b_i) \in R \times G \times B$, such that all the points in $R \cup G \cup B$ are included in exactly one such triplet, and furthermore, the price function

$$f(\mathcal{T}) = \max_{i=1}^n (\|r_i - g_i\| + \|g_i - b_i\| + \|b_i - r_i\|)$$

is minimized (i.e., you are minimizing the maximum perimeter of the triangles you choose), where $\|p - q\|$ denotes the Euclidean distance between p and q . Provide a polynomial time constant approximation algorithm for this problem. **Prove** the correctness and the quality of approximation of your algorithm. The better the constant of approximation in your algorithm, the better your solution is.

(Hint: You would need to use network flow somewhere.)

2 (20 PTS.) Can you hear me?

You are given a set P of n points in the plane (i.e., location of n clients with phones), and a set Q of m points (i.e., base stations). The i th base station b_i , can serve at most α_i clients, and each client has to be in distance at most r_i from it, for $i = 1, \dots, m$. Describe a polynomial time algorithm that computes for each client which base station it should use, and no base station is assigned more clients that it can use. What is the running time of your algorithm?

3 (20 PTS.) Unique Cut.

3.A. (10 PTS.) Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and nonnegative edge capacities $\{c_e\}$. Give a polynomial-time algorithm to decide whether G has a *unique* minimum s - t cut (i.e., an s - t of capacity strictly less than that of all other s - t cuts).

3.B. (10 PTS.) THE GOOD, THE BAD, AND THE MIDDLE.

Suppose you're looking at a flow network G with source s and sink t , and you want to be able to express something like the following intuitive notion: Some nodes are clearly on the "source side" of the main bottlenecks; some nodes are clearly on the "sink side" of the main bottlenecks; and some nodes are in the middle. However, G can have many minimum cuts, so we have to be careful in how we try making this idea precise.

Here's one way to divide the nodes of G into three categories of this sort.

- We say a node v is *upstream* if, for all minimum s - t cuts (A, B) , we have $v \in A$ – that is, v lies on the source side of every minimum cut.

- We say a node v is **downstream** if, for all minimum s - t cuts (A, B) , we have $v \in B$ – that is, v lies on the sink side of every minimum cut.
- We say a node v is **central** if it is neither upstream nor downstream; there is at least one minimum s - t cut (A, B) for which $v \in A$, and at least one minimum s - t cut (A', B') for which $v \in B'$.

Give an algorithm that takes a flow network G and classifies each of its nodes as being upstream, downstream, or central. The running time of your algorithm should be within a constant factor of the time required to compute a *single* maximum flow.

4 (20 PTS.) Prove infeasibility.

You are trying to solve a circulation problem, but it is not feasible. The problem has demands, but no capacity limits on the edges. More formally, there is a graph $G = (V, E)$, and demands d_v for each node v (satisfying $\sum_{v \in V} d_v = 0$), and the problem is to decide if there is a flow f such that $f(e) \geq 0$ and $f^{in}(v) - f^{out}(v) = d_v$ for all nodes $v \in V$. Note that this problem can be solved via the circulation algorithm by setting $c_e = +\infty$ for all edges $e \in E$. (Alternately, it is enough to set c_e to be an extremely large number for each edge – say, larger than the total of all positive demands d_v in the graph.)

You want to fix up the graph to make the problem feasible, so it would be very useful to know why the problem is not feasible as it stands now. On a closer look, you see that there is a subset U of nodes such that there is no edge into U , and yet $\sum_{v \in U} d_v > 0$. You quickly realize that the existence of such a set immediately implies that the flow cannot exist: The set U has a positive total demand, and so needs incoming flow, and yet U has no edges into it. In trying to evaluate how far the problem is from being solvable, you wonder how big the demand of a set with no incoming edges can be.

Give a polynomial-time algorithm to find a subset $S \subset V$ of nodes such that there is no edge into S and for which $\sum_{v \in S} d_v$ is as large as possible subject to this condition.

(Hint: Think about strong connected components, and how to use them in this case.)

5 (20 PTS.) Maximum Flow By Scaling

Let $G = (V, E)$ be a flow network with source s , sink t , and an integer capacity $c(u, v)$ on each edge $(u, v) \in E$. Let $C = \max_{(u, v) \in E} c(u, v)$.

- 5.A.** (3 PTS.) Argue that a minimum cut of G has capacity at most $C|E|$.
- 5.B.** (3 PTS.) For a given number K , show that an augmenting path of capacity at least K can be found in $O(E)$ time, if such a path exists.

The following modification of FORD-FULKERSON-METHOD can be used to compute a maximum flow in G .

```

maxFlow-By-Scaling( $G, s, t$ )
1   $C \leftarrow \max_{(u, v) \in E} c(u, v)$ 
2  initialize flow  $f$  to 0
3   $K \leftarrow 2^{\lceil \lg C \rceil}$ 
4  while  $K \geq 1$  do {
5      while (there exists an augmenting path  $p$  of
              capacity at least  $K$ ) do {
6          augment flow  $f$  along  $p$ 
              }
7       $K \leftarrow K/2$ 
      }
8  return  $f$ 

```

- 5.C. (3 PTS.) Argue that **maxFlow-By-Scaling** returns a maximum flow.
 - 5.D. (3 PTS.) Show that the capacity of a minimum cut of the residual graph G_f is at most $2K|E|$ each time line 4 is executed.
 - 5.E. (4 PTS.) Argue that the inner **while** loop of lines 5-6 is executed $O(|E|)$ times for each value of K .
 - 5.F. (4 PTS.) Conclude that **maxFlow-By-Scaling** can be implemented so that it runs in $O(E^2 \lg C)$ time.
-

61.1.5. Homework 4

CS 573: Graduate algorithms, Fall 2015

Homework 4, due Monday, November 17, 23:59:59, 2018

Version 1.0

1 (20 PTS.) Slack form

Let L be a linear program given in slack form, with n nonbasic variables N , and m basic variables B . Let N' and B' be a different partition of $N \cup B$, such that $|N' \cup B'| = |N \cup B|$. Show a polynomial time algorithm that computes an equivalent slack form that has N' as the nonbasic variables and B' as the basic variables. How fast is your algorithm?

2 (20 PTS.) Tedious Computations

Provide *detailed* solutions for the following problems, showing each pivoting stage separately.

2.A. (5 PTS.)

maximize $6x_1 + 3x_2 + 5x_3 + 2x_4$
 subject to
 $x_1 + x_2 + x_3 + x_4 = 1$
 $x_1, x_2, x_3, x_4 \geq 0$.

2.B. (5 PTS.)

maximize $2x_1 + 4x_2$
 subject to
 $2x_1 + x_2 \leq 4$
 $2x_1 + 3x_2 \leq 3$
 $4x_1 + x_2 \leq 5$
 $x_1 + 5x_2 \leq 1$
 $x_1, x_2 \geq 0$.

2.C. (5 PTS.)

maximize $6x_1 + 2x_2 + 8x_3 + 9x_4$
 subject to
 $2x_1 + x_2 + x_3 + 3x_4 \leq 5$
 $x_1 + 3x_2 + x_3 + 2x_4 \leq 3$
 $x_1, x_2, x_3, x_4 \geq 0$.

2.D. (5 PTS.)

minimize $2x_{12} + 8x_{13} + 3x_{14} + 2x_{23} + 7x_{24} + 3x_{34}$
 subject to
 $x_{12} + x_{13} + x_{14} \geq 1$

$$\begin{aligned}
-x_{12} + x_{23} + x_{24} &= 0 \\
-x_{13} - x_{23} + x_{34} &= 0 \\
x_{14} + x_{24} + x_{34} &\leq 1 \\
x_{12}, x_{13}, \dots, x_{34} &\geq 0.
\end{aligned}$$

3 (20 PTS.) Linear Programming for a Graph

In the *minimum-cost multicommodity-flow problem*, we are given a directed graph $G = (V, E)$, in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$ and a cost $\alpha(u, v)$. As in the multicommodity-flow problem (Chapter 29.2, CLRS), we are given k different commodities, K_1, K_2, \dots, K_k , where commodity i is specified by the triple $K_i = (s_i, t_i, d_i)$. Here s_i is the source of commodity i , t_i is the sink of commodity i , and d_i is the demand, which is the desired flow value for commodity i from s_i to t_i . We define a flow for commodity i , denoted by f_i , (so that $f_i(u, v)$ is the flow of commodity i from vertex u to vertex v) to be a real-valued function that satisfies the flow-conservation, skew-symmetry, and capacity constraints. We now define $f(u, v)$, the *aggregate flow* to be sum of the various commodity flows, so that $f(u, v) = \sum_{i=1}^k f_i(u, v)$. The aggregate flow on edge (u, v) must be no more than the capacity of edge (u, v) .

The cost of a flow is $\sum_{u,v \in V} f(u, v)\alpha(u, v)$, and the goal is to find the feasible flow of minimum cost. Express this problem as a linear program.

4 (20 PTS.) Linear programming

4.A. (10 PTS.) Show the following problem in NP-hard.

Integer Linear Programming

Instance: A linear program in standard form, in which A and B contain only integers.
Question: Is there a solution for the linear program, in which the x must take integer values?

4.B. (5 PTS.) A steel company must decide how to allocate next week's time on a rolling mill, which is a machine that takes unfinished slabs of steel as input and produce either of two semi-finished products: bands and coils. The mill's two products come off the rolling line at different rates:

Bands 200 tons/hr
Coils 140 tons/hr.

They also produce different profits:

Bands \$ 25/ton
Coils \$ 30/ton.

Based on current booked orders, the following upper bounds are placed on the amount of each product to produce:

Bands 6000 tons
Coils 4000 tons.

Given that there are 40 hours of production time available this week, the problem is to decide how many tons of bands and how many tons of coils should be produced to yield the greatest profit. Formulate this problem as a linear programming problem. Can you solve this problem by inspection?

4.C. (5 PTS.) A small airline, Ivy Air, flies between three cities: Ithaca (a small town in upstate New York), Newark (an eyesore in beautiful New Jersey), and Boston (a yuppie town in Massachusetts). They offer several flights but, for this problem, let us focus on the Friday afternoon flight that departs from Ithaca, stops in Newark, and continues to Boston. There are three types of passengers:

1. Those traveling from Ithaca to Newark (god only knows why).

2. Those traveling from Newark to Boston (a very good idea).
3. Those traveling from Ithaca to Boston (it depends on who you know).

The aircraft is a small commuter plane that seats 30 passengers. The airline offers three fare classes:

1. Y class: full coach.
2. B class: nonrefundable.
3. M class: nonrefundable, 3-week advanced purchase.

Ticket prices, which are largely determined by external influences (i.e., competitors), have been set and advertised as follows:

	Ithaca-Newark	Newark-Boston	Ithaca-Boston
Y	300	160	360
B	220	130	280
M	100	80	140

Based on past experience, demand forecasters at Ivy Air have determined the following upper bounds on the number of potential customers in each of the 9 possible origin-destination/fare-class combinations:

	Ithaca-Newark	Newark-Boston	Ithaca-Boston
Y	4	8	3
B	8	13	10
M	22	20	18

The goal is to decide how many tickets from each of the 9 origin/destination/fare-class combinations to sell. The constraints are that the plane cannot be overbooked on either the two legs of the flight and that the number of tickets made available cannot exceed the forecasted maximum demand. The objective is to maximize the revenue. Formulate this problem as a linear programming problem.

5 (20 PTS.) Some duality required.

5.A. (5 PTS.) What is the dual of the following LP?

$$\begin{aligned}
 &\text{maximize} && x_1 - 2x_2 \\
 &\text{subject to} && x_1 + 2x_2 - x_3 + x_4 \geq 0 \\
 &&& 4x_1 + 3x_2 + 4x_3 - 2x_4 \leq 3 \\
 &&& -x_1 - x_2 + 2x_3 + x_4 = 1 \\
 &&& x_2, x_3 \geq 0
 \end{aligned}$$

5.B. (7 PTS.) Solve the above LP in detail, providing the state of the LP after each pivot step. What is the value of the target function of your LP?

5.C. (8 PTS.) Solve the dual of the above LP in detail, providing the state of the LP after each pivot step.

6 (20 PTS.) Strong duality.

Consider a directed graph G with source vertex s and target vertex t and associated costs $c_e \geq 0$ on the edges. Let \mathcal{P} denote the set of all the directed (simple) paths from s to t in G .

Consider the following (very large) integer program:

$$\begin{aligned}
 &\text{minimize} && \sum_{e \in E(G)} c_e x_e \\
 &\text{subject to} && x_e \in \{0, 1\} \quad \forall e \in E(G) \\
 &&& \sum_{e \in \pi} x_e \geq 1 \quad \forall \pi \in \mathcal{P}.
 \end{aligned}$$

- 6.A. (5 PTS.) What does this IP compute?
- 6.B. (5 PTS.) Write down the relaxation of this IP into a linear program.
- 6.C. (5 PTS.) Write down the dual of the LP from (B). What is the interpretation of this new LP? What is it computing for the graph G (prove your answer)?
- 6.D. (5 PTS.) The strong duality theorem states the following.

Theorem 61.1.1. *If the primal LP problem has an optimal solution $x^* = (x_1^*, \dots, x_n^*)$ then the dual also has an optimal solution, $y^* = (y_1^*, \dots, y_m^*)$, such that*

$$\sum_j c_j x_j^* = \sum_i b_i y_i^*.$$

In the context of (A)-(C) what result is implied by this theorem if we apply it to the primal LP and its dual above? (For this, you can assume that the optimal solution to the LP of (B) is integral – which is not quite true – things are slightly more complicated than that.)

61.1.6. Homework 5

1 (20 PTS.) Sorting networks stuff

- 1.A. (2 PTS.) Prove that an n -input sorting network must contain at least one comparator between the i th and $(i + 1)$ st lines for all $i = 1, 2, \dots, n - 1$.
- 1.B. (10 PTS.) Prove that in a sorting network for n inputs, there must be at least $\Omega(n \log n)$ gates. For full credit, your answer should be short, and self contained (i.e., no reduction please).
[As an exercise, you should think why your proof does not imply that a regular sorting algorithm takes $\Omega(n \log n)$ time in the worst case.]
- 1.C. (3 PTS.)
Suppose that we have $2n$ elements $\langle a_1, a_2, \dots, a_{2n} \rangle$ and wish to partition them into the n smallest and the n largest. Prove that we can do this in constant additional depth after separately sorting $\langle a_1, a_2, \dots, a_n \rangle$ and $\langle a_{n+1}, a_{n+2}, \dots, a_{2n} \rangle$.
- 1.D. (5 PTS.)
Let $S(k)$ be the depth of a sorting network with k inputs, and let $M(k)$ be the depth of a merging network with $2k$ inputs. Suppose that we have a sequence of n numbers to be sorted and we know that every number is within k positions of its correct position in the sorted order, which means that we need to move each number at most $(k - 1)$ positions to sort the inputs. For example, in the sequence 3 2 1 4 5 8 7 6 9, every number is within 3 positions of its correct position. But in sequence 3 2 1 4 5 9 8 7 6, the number 9 and 6 are outside 3 positions of its correct position.
Show that we can sort the n numbers in depth $S(k) + 2M(k)$. (You need to prove your answer is correct.)

2 (10 PTS.) Computing Polynomials Quickly

In the following, assume that given two polynomials $p(x), q(x)$ of degree at most n , one can compute the polynomial remainder of $p(x) \bmod q(x)$ in $O(n \log n)$ time. The *remainder* of $r(x) = p(x) \bmod q(x)$ is the unique polynomial of degree smaller than this of $q(x)$, such that $p(x) = q(x) * d(x) + r(x)$, where $d(x)$ is a polynomial.

Let $p(x) = \sum_{i=0}^{n-1} a_i x^i$ be a given polynomial.

- 2.A.** (2 PTS.) Prove that $p(x) \bmod (x - z) = p(z)$, for all z .
- 2.B.** (2 PTS.) We want to evaluate $p(\cdot)$ on the points x_0, x_1, \dots, x_{n-1} . Let

$$P_{ij}(x) = \prod_{k=i}^j (x - x_k)$$

and

$$Q_{ij}(x) = p(x) \bmod P_{ij}(x).$$

Observe that the degree of Q_{ij} is at most $j - i$.

Prove that, for all x , $Q_{kk}(x) = p(x_k)$ and $Q_{0,n-1}(x) = p(x)$.

- 2.C.** (4 PTS.) Prove that for $i \leq k \leq j$, we have

$$\forall x \quad Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$$

and

$$\forall x \quad Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x).$$

- 2.D.** (4 PTS.) Given an $O(n \log^2 n)$ time algorithm to evaluate $p(x_0), \dots, p(x_{n-1})$. Here x_0, \dots, x_{n-1} are n given real numbers.

3 (20 PTS.) Linear time Union-Find.

- 3.A.** (2 PTS.) With path compression and union by rank, during the lifetime of a Union-Find data-structure, how many elements would have rank equal to $\lfloor \lg n - 5 \rfloor$, where there are n elements stored in the data-structure?
- 3.B.** (2 PTS.) Same question, for rank $\lfloor (\lg n)/2 \rfloor$.
- 3.C.** (4 PTS.) Prove that in a set of n elements, a sequence of n consecutive FIND operations take $O(n)$ time in total.
- 3.D.** (4 PTS.) Write a non-recursive version of FIND with path compression.
- 3.E.** (4 PTS.) Show that any sequence of m MAKESET, FIND, and UNION operations, where all the UNION operations appear before any of the FIND operations, takes only $O(m)$ time if both path compression and union by rank are used.
- 3.F.** (4 PTS.) What happens in the same situation if only the path compression is used?

4 (10 PTS.) Naive.

We wish to compress a sequence of independent, identically distributed random variables X_1, X_2, \dots . Each X_j takes on one of n values. The i th value occurs with probability p_i , where $p_1 \geq p_2 \geq \dots \geq p_n$. The result is compressed as follows. Set

$$T_i = \sum_{j=1}^{i-1} p_j,$$

and let the i th codeword be the first $\lceil \lg(1/p_i) \rceil$ bits (in the binary representation) of T_i . Start with an empty string, and consider X_j in order. If X_j takes on the i th value, append the i th codeword to the end of the string.

- 4.A.** Show that no codeword is the prefix of any other codeword.
- 4.B.** Let Z be the average number of bits appended for each random variable X_j . Show that

$$\mathbb{H}(X_j) \leq Z \leq \mathbb{H}(X_j) + 1.$$

5 (20 PTS.) Codification.

Arithmetic coding is a standard compression method. In the case when the string to be compressed is a sequence of biased coin flips, it can be described as follows. Suppose that we have a sequence of bits $X = (X_1, X_2, \dots, X_n)$, where each X_i is independently 0 with probability p and 1 with probability $1 - p$. The sequences can be ordered lexicographically, so for $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_n)$, we say that $x < y$ if $x_i = 0$ and $y_i = 1$ in the first coordinate i such that $x_i \neq y_i$. If $z(x)$ is the number of zeroes in the string x , then define $p(x) = p^{z(x)}(1 - p)^{n - z(x)}$ and

$$q(x) = \sum_{y < x} p(y).$$

- 5.A.** Suppose we are given $X = (X_1, X_2, \dots, X_n)$. Explain how to compute $q(X)$ in time $O(n)$ (assume that any reasonable operation on real numbers takes constant time).
- 5.B.** Argue that the intervals $[q(x), q(x) + p(x))$ are disjoint subintervals of $[0, 1)$.
- 5.C.** Given (A) and (B), the sequence X can be represented by any point in the interval $I(X) = [q(X), q(X) + p(X))$. Show that we can choose a codeword in $I(X)$ with $\lceil \lg(1/p(X)) \rceil + 1$ binary digits to represent X in such a way that no codeword is the prefix of any other codeword.
- 5.D.** Given a codeword chosen as in (C), explain how to decompress it to determine the corresponding sequence (X_1, X_2, \dots, X_n) .
- 5.E.** (Extra credit.) Using the Chernoff inequality, argue that $\lg(1/p(X))$ is close to $n\mathbb{H}(p)$ with high probability. Thus, this approach yields an effective compression scheme.

6 (20 PTS.) Entropy stuff.

6.A. (5 PTS.) *Maximizing Entropy*

Consider an n -sided die, where the i th face comes up with probability p_i . Show that the entropy of a die roll is maximized when each face comes up with equal probability $1/n$.

6.B. (5 PTS.) *Extraction to the limit,*

We have shown that we can extract, on average, at least $\lceil \lg m \rceil - 1$ independent, unbiased bits from a number chosen uniformly at random from $\{0, \dots, m - 1\}$. It follows that if we have k numbers chosen independently and uniformly at random from $\{0, \dots, m - 1\}$ then we can extract, on average, at least $k \lceil \lg m \rceil - k$ independent, unbiased bits from them. Give a better procedure that extracts, on average, at least $k \lceil \lg m \rceil - 1$ independent, unbiased bits from these numbers.

- 6.C.** (2 PTS.) Assume you have a (valid) prefix code with n codewords, where the i th codeword is made out of ℓ_i bits. Prove that

$$\sum_{i=1}^n \frac{1}{2^{\ell_i}} \leq 1.$$

- 6.D.** (2 PTS.) Let $S = \sum_{i=1}^{10} 1/i^2$. Consider a random variable X such that $\mathbb{P}[X = i] = 1/(Si^2)$, for $i = 1, \dots, 10$. Compute $\mathbb{H}(X)$.
- 6.E.** (2 PTS.) Let $S = \sum_{i=1}^{10} 1/i^3$. Consider a random variable X such that $\mathbb{P}[X = i] = 1/(Si^3)$, for $i = 1, \dots, 10$. Compute $\mathbb{H}(X)$.
- 6.F.** (2 PTS.) Let $S(\alpha) = \sum_{i=1}^{10} 1/i^\alpha$, for $\alpha > 1$. Consider a random variable X such that $\mathbb{P}[X = i] = 1/(S(\alpha)i^\alpha)$, for $i = 1, \dots, 10$. Prove that $\mathbb{H}(X)$ is either increasing or decreasing as a function of α (you can assume that α is an integer).

6.G. (2 PTS.) The *conditional entropy* $\mathbb{H}(Y|X)$ is defined by

$$\mathbb{H}(Y|X) = \sum_{x,y} \mathbb{P}[(X = x) \cap (Y = y)] \lg \frac{1}{\mathbb{P}[Y = y|X = x]}.$$

If $Z = (X, Y)$, prove that

$$\mathbb{H}(Z) = \mathbb{H}(X) + \mathbb{H}(Y|X).$$

61.2. Midterm

1 COLORING GRAPHS. (25 PTS.)

For a set S , a **balanced coloring** ϕ assigns every element in S a label that is either -1 or $+1$. For a set $F \subseteq S$, its **balance** is $\phi(F) = \sum_{x \in F} \phi(x)$.

Prove that the following problem is **NP-COMplete**:

Balanced coloring of a set system.

Instance: (S, \mathcal{F}) : S is a set of n elements, and \mathcal{F} is a family of subsets of S .

Question: Is there a balanced coloring ϕ of S , such that for any set $F \in \mathcal{F}$, we have;

- If $|F|$ is even then $\phi(F) = 0$.
- If $|F|$ is odd then $\phi(F) \geq -1$.

(Hint: Think what a balanced coloring means for sets of size two and three.)

2 INDEPENDENCE IN A HYPERGRAPH. (25 PTS.)

Let (V, \mathcal{F}) be a set system, with $n = |V|$, and \mathcal{F} is a family (i.e., set) of m subsets of V . Here, every set $F \in \mathcal{F}$ is of size exactly three. A set of $S \subseteq V$ is **independent**, if for all $F \in \mathcal{F}$, not all the elements of F are contained in S (i.e., at most two elements of F are in S).

- 2.A.** (5 PTS.) Let $S \subseteq V$ be a random sample generated by picking every element of V into the sample, independently, with probability $1/t$, for some parameter t . A set $F \in \mathcal{F}$ is **bad** if all its elements are in S (i.e., $F \subseteq S$). What is the probability of a specific set $F \in \mathcal{F}$ to be bad?
- 2.B.** (5 PTS.) Let X be the random variable that is the number of bad sets in \mathcal{F} in relation to the random sample S . What is $\mu = \mathbb{E}[X]$?
- 2.C.** (5 PTS.) Prove that $\mathbb{P}[X \geq 2\mu] \leq 1/2$.
- 2.D.** (10 PTS.) Consider the algorithm that now fixes S to be an independent set as follows: scan all the bad sets, and for each such bad set $F \in \mathcal{F}$, randomly throw away one element from S such that F is no longer contained in S .

Verify that the resulting set S' is an independent set. Provide a lower bound, as good as possible, as a function of t , on the expected size of S' . What is the choice of t (as a function of n and m) for which the algorithm (in expectation) outputs the largest possible independent set? What is the expected size of the independent set in this case? (Bigger is better.)

3 FIRE STATIONS. (25 PTS.)

Let $C = \{c_1, \dots, c_n\}$ be the set of locations of n small towns living on the real axis (it is a straight road in the middle of nowhere, and these are the locations of the tiny towns starting from one of its endpoints). Being in America, we would like to build k fire stations to serve their gas needs. Specifically, for a set of locations $Y = \{y_1, \dots, y_k\}$ of the gas stations, the cost of this solution to the i th customer is the squared distance of c_i to its nearest neighbor in Y . Formally, it is $\text{price}(c_i, Y) = |c_i - \text{nn}(c_i, Y)|^2$, where $\text{nn}(c_i, Y)$ is the location of the nearest point to c_i in Y .

(This might seem strange, but the further the fire station is, the more damage caused by the fire before help shows up. This pricing model just try to capture this intuition.)

The **price** of the solution Y is $\text{price}(C, Y) = \sum_i \text{price}(c_i, Y)$.

Given C and k , provide a polynomial time algorithm (in n and k) that computes the price of the cheapest possible solution (i.e., the price of the optimal solution). What is the running time of your algorithm? [You can assume in your solution that $Y \subseteq C$.]

4 GREEDY HITTING SET. (25 PTS.)

Consider the following problem:

Hitting Set

Instance: (S, \mathcal{F}) :

S - a set of n elements

\mathcal{F} - a family of m subsets of S .

Question: Compute a set $S' \subseteq S$ such that S' contains as few elements as possible, and S' “hits” all the sets in \mathcal{F} . Formally, for all $F \in \mathcal{F}$, we have $|S' \cap F| \geq 1$.

The greedy algorithm **GreedyHit** computes a solution by repeatedly computing the element in S , that is contained in the largest number of sets of \mathcal{F} that are not hit yet, adding it to the current solution, and repeating this till all the sets of \mathcal{F} are hit. Let k be the number of elements in the optimal cover. Prove the following:

- 4.A.** (5 PTS.) In the beginning of the i th iteration, if there are β_i sets in \mathcal{F} not hit yet, then there is an element in S that hits at least β_i/k of these sets.
- 4.B.** (10 PTS.) Prove that for any i , we have that $\beta_{i+k} \leq \beta_i/c$, where $c > 1$ is some positive constant (what is the value of c - provide a reasonable lower bound).
- 4.C.** (8 PTS.) Using the above, provide an upper bound (as small as possible) on the number of iterations performed by **GreedyHit** before it stops.
- 4.D.** (2 PTS.) What is the quality of approximation provided by **GreedyHit**?

61.3. Final

1 STRONG DUALITY. (20 PTS.)

Consider a directed graph G with source vertex s and target vertex t and associated costs $\text{cost}(\cdot) \geq 0$ on the edges. Let \mathcal{P} denote the set of all the directed (simple) paths from s to t in G .

Consider the following (very large) integer program:

$$\begin{aligned} & \text{minimize} && \sum_{e \in E(G)} \text{cost}(e)x_e \\ & \text{subject to} && x_e \in \{0, 1\} \quad \forall e \in E(G) \\ & && \sum_{e \in \pi} x_e \geq 1 \quad \forall \pi \in \mathcal{P}. \end{aligned}$$

- 1.A. (5 PTS.) What does this IP compute?
1.B. (5 PTS.) Write down the relaxation of this IP into a linear program.
1.C. (5 PTS.) Write down the dual of the LP from (B). What is the interpretation of this new LP? What is it computing for the graph G (prove your answer)?
1.D. (5 PTS.) The strong duality theorem states the following.

Theorem 61.3.1. *If the primal LP problem has an optimal solution $x^* = (x_1^*, \dots, x_n^*)$ then the dual also has an optimal solution, $y^* = (y_1^*, \dots, y_m^*)$, such that*

$$\sum_j c_j x_j^* = \sum_i b_i y_i^*.$$

In the context of (A)–(C) what result is implied by this theorem if we apply it to the primal LP and its dual above? (For this, you can assume that the optimal solution to the LP of (B) is integral.)

2 SEQUENCES AND CONSEQUENCES. (20 PTS.)

Let $\mathcal{X} = \langle X_1, X_2, \dots, X_n \rangle$ be a sequence of n numbers generated by picking each X_i independently and uniformly from the range $\{1, \dots, n\}$.

- 2.A. (5 PTS.) What is the entropy of \mathcal{X} ?
2.B. (5 PTS.) Consider the sequence $\mathcal{Y} = \langle Y_1, \dots, Y_n \rangle$ that results from sorting the sequence \mathcal{X} in increasing order. For example, if $\mathcal{X} = \langle 4, 1, 4, 1 \rangle$ then $\mathcal{Y} = \langle 1, 1, 4, 4 \rangle$.
Describe an encoding scheme that takes the sequence \mathcal{Y} and encodes it as a sequence of $2n$ binary bits (you will lose points if your scheme uses more bits). Given this encoded sequence of bits, how do you recover the sequence \mathcal{Y} ? (Hint: Consider the differences sequence $Y_1, Y_2 - Y_1, Y_3 - Y_2, \dots, Y_n - Y_{n-1}$. And do **not** use Huffman's encoding.)
Demonstrate how your encoding scheme works for the sequence $\mathcal{Y} = \langle 1, 1, 4, 6, 6, 6 \rangle$.
2.C. (5 PTS.) Consider the set U of all sequences \mathcal{Y} that can be generated by the above process (i.e., it is the set of all monotonically non-decreasing sequences of length n using integer numbers in the range 1 to n). Provide (and prove) an upper bound on the number of elements in U . Your bound should be as small as possible. (Hint: Use (B).)
(Note, that we are not asking for the exact bound on the size of U , which is doable but harder.)
2.D. (5 PTS.) **Prove** an upper bound (as low as possible) on the entropy of \mathcal{Y} . (Proving a lower bound here seems quite hard and you do not have to do it.)

3 FIND k TH SMALLEST NUMBER. (20 PTS.)

This question asks you to design and analyze a *randomized incremental* algorithm to select the k th smallest element from a given set of n elements (from a universe with a linear order).

We assume the numbers are given to you one at a time, and your algorithm has only $O(k)$ space in its disposal that it can use (in particular, the algorithm can not just read all the input and only then compute the desired quantity). Specifically, in an *incremental* algorithm, the input consists of a sequence of elements x_1, x_2, \dots, x_n . After any prefix x_1, \dots, x_{i-1} has been read, the algorithm has computed the k th smallest element in x_1, \dots, x_{i-1} (which is undefined if $i \leq k$), or if appropriate, some other invariant from which the k th smallest element could be determined. This invariant is updated as the next element x_i is read.

We assume that before it is given to us, the input sequence has been randomly permuted, with each permutation equally likely. Note that this case is of interest in analyzing real world situations, where the input arrives as a stream, and we believe that this stream behaves like a random stream of numbers.

- 3.A.** (5 PTS.) Describe an efficient incremental algorithm for computing the k th smallest element (the more efficient it is, the better).
- 3.B.** (5 PTS.) How many comparisons does your algorithm perform in the worst case?
- 3.C.** (10 PTS.) Consider the problem of computing the k smallest numbers in a given stream. Describe an algorithm that outputs these k numbers in sorted order, assuming that k and n are provided in advance, the algorithm has $O(k)$ space, and the input is provided in a stream that is randomly permuted.

What is the expected number (over all permutations) of comparisons performed by your algorithm? For full credit, the expected number of comparisons performed by your algorithm should be as small as possible. **Prove** your answer.

4 STAB THESE RECTANGLES (IN THE BACK, IF POSSIBLE). (20 PTS.)

You are given a set $\mathcal{R} = \{R_1, \dots, R_n\}$ of n rectangles in the plane, and a set $P = \{p_1, \dots, p_n\}$ of n points in the plane. For every point $p \in P$, there is an associated weight $w_p > 0$. Your purpose in this problem is to select a minimum weight subset $X \subseteq P$, such that for any rectangle R of \mathcal{R} there is at least one point of X that is contained in R .

Under the assumption that no rectangle of \mathcal{R} contains more than k points, describe a polynomial time approximation algorithm for this problem. What is the approximation quality of your algorithm? (Naturally, your approximation algorithm should have the best possible approximation quality.) Prove your stated bound on the quality of approximation.

5 SORTING IN $\Omega(n \log n)$ TIME. (20 PTS.)

Prove that any sorting algorithm in the comparison model for sorting n numbers takes $\Omega(n \log n)$ time.

This question would be graded strictly – there is no partial credit for this question.

Chapter 62

Fall 2015: CS 473 – Theory II

62.1. Homeworks

62.1.1. Homework 0

HW 0 (due Monday, at noon, August 31, 2018)

CS 473: Theory II, Fall 2015

1 (50 PTS.) Some probability required.

Consider an undirected graph $G = (V, E)$ with n vertices and m edges, with $m \geq n$.

- 1.A. (10 PTS.) Let $p = n/(2m)$. For every vertex $v \in V$, pick it to be in the set X with probability p . What is the expected number of vertices in X ?
- 1.B. (10 PTS.) Let G_X be the *induced* graph by G on X . Formally, $V(G_X) = X$, and an edge $uv \in E$ is in $E(G_X)$ if and only if both u and v are in X . Provide and prove an exact bound for the expected number of edges in G_X . (Hint: Linearity of expectations.)
- 1.C. (10 PTS.) Set $Y \leftarrow X$. Next, for every edge $xy \in E(G_X)$, remove, say, the vertex x from the set Y (i.e., you remove at most one vertex for every edge of G_X). Let Z be the resulting set of vertices. Prove, that induced graph G_Z has no edges.
- 1.D. (10 PTS.) Provide a **lower** upper bound, as tight as possible, on the expected size of Z .
- 1.E. (10 PTS.) Prove, using the above (and only the above), that in any graph G with n vertices and m edges, there is always an independent set of size $\Omega(n^2/m)$. (Proving this argument formally is easy but surprisingly subtle - be careful.)

2 (50 PTS.) My point is shrinking. (50 PTS.)

- 2.A. (40 PTS.) Let $\mathbf{p} = (p_1, \dots, p_d)$ be a point in \mathbb{R}^d , with all the coordinate being positive integers. Consider the following fabulous algorithm.

```

shrink(p) :
  if  $p_1 = p_2 = \dots = p_d$  then return  $p_1$ 
   $f \leftarrow 1$ 
  for  $i = 1, \dots, d$  do
    if  $p_i$  is odd then  $f \leftarrow 0$ 
  if  $f = 1$  then
    return  $2 * \text{shrink}((p_1/2, p_2/2, \dots, p_d/2))$ .      (*)
  q  $\leftarrow$  p
  for  $i = 1, \dots, d$  do
    if  $p_i$  is even then
       $q_i \leftarrow q_i/2$ 
    return shrink(q)

 $\alpha \leftarrow \arg \min_i q_i$ 
 $\beta \leftarrow \arg \max_i q_i$ 
 $q_\beta = q_\beta - q_\alpha$ 
return shrink(q)                                          (**)

```

Here is an example of the execution of `play`((14, 2048, 1022)).

(14, 2048, 1022) \rightarrow^* (7, 1024, 511) \rightarrow (7, 512, 511) \rightarrow (7, 256, 511) \rightarrow (7, 128, 511) \rightarrow (7, 64, 511) \rightarrow
 (7, 32, 511) \rightarrow (7, 16, 511) \rightarrow (7, 8, 511) \rightarrow (7, 4, 511) \rightarrow (7, 2, 511) \rightarrow (7, 1, 511) \rightarrow
 (7, 1, 510) \rightarrow (7, 1, 255) \rightarrow (7, 1, 254) \rightarrow (7, 1, 127) \rightarrow (7, 1, 126) \rightarrow (7, 1, 63) \rightarrow (7, 1, 62) \rightarrow
 (7, 1, 31) \rightarrow (7, 1, 30) \rightarrow (7, 1, 15) \rightarrow (7, 1, 14) \rightarrow (7, 1, 7) \rightarrow (7, 1, 6) \rightarrow (7, 1, 3) \rightarrow (6, 1, 3) \rightarrow
 (3, 1, 3) \rightarrow (2, 1, 3) \rightarrow (1, 1, 3) \rightarrow (1, 1, 2) \rightarrow (1, 1, 1) \rightarrow Output: 2.

Prove (maybe using induction, but you do not have to) that `shrink` always terminates.

(Hint: Come up with an argument why in each step some non-trivial progress is being made. As a warm-up exercise, prove that the algorithm always terminates if the initial input has three numbers.)

- 2.B.** (5 PTS.) Assuming that the input \mathbf{p} is given using the (standard) binary representation, let N be the number of bits needed to represent \mathbf{p} . Provide a tight bound, to the value of N as a function of the values of p_1, \dots, p_d .
- 2.C.** (5 PTS.) Provide a bound, as tight as possible, on the running time of `shrink` as a function of N and d . (Hint: First analyze the algorithm when `(**)` never happened. Then, extend your analysis to the case that `(**)` does happen.)

62.1.2. Homework 1

HW 1 (due Monday, Noon, September 7, 2018)

CS 473: Theory II, Fall 2015

Collaboration Policy: For this homework, Problems 1–2 can be worked in groups of up to three students. Submission is online on moodle.

1 (50 PTS.) Reduction, deduction, induction, and abduction.

The following question is long, but not very hard, and is intended to make sure you understand the following problems, and the basic concepts needed for proving NP-Completeness.

All graphs in the following have n vertices and m edges.

For each of the following problems, you are given an instance of the problem of size n . Imagine that the answer to this given instance is “yes”, and that you need to convince somebody that indeed the answer to the given instance is **yes**. To this end, describe:

- (I) An algorithm for solving the given instance (not necessarily efficient). What is the running time of your algorithm?
- (II) The format of the certificate that the instance is correct.
- (III) A bound on the length of the certificate (its have to be of polynomial length in the input size).
- (IV) An efficient algorithm (as fast as possible [it has to be polynomial time]) for verifying, given the instance and the proof, that indeed the given instance is indeed **yes**. What is the running time of your algorithm?

We solve the first such question as an example.

(EXAMPLE)

Shortest Path

Instance: A weighted undirected graph G , vertices s and t and a threshold w .

Question: Is there a path between s and t in G of length at most w ?

Solution:

- (I) We seen in class the Dijkstra algorithm for solving the shortest path problem in $O(n \log n + m) = O(n^2)$ time. Given the shortest path, we can just compare its price to w , and return yes/no accordingly.
 - (II) A “proof” in this case would be a path π in G (i.e., a sequence of at most n vertices) connecting s to t , such that its total weight is at most w .
 - (III) The proof here is a list of $O(n)$ vertices, and can be encoded as a list of $O(n)$ integers. As such, its length is $O(n)$.
 - (IV) The verification algorithm for the given solution/proof, would verify that all the edges in the path are indeed in the graph, the path starts at s and ends at t , and that the total weight of the edges of the path is at most w . The proof has length $O(n)$ in this case, and the verification algorithm runs in $O(n^2)$ time, if we assume the graph is given to us using adjacency lists representation.
-

(A) (5 PTS.)

Semi-Independent Set

Instance: A graph G , integer k

Question: Is there a semi-independent set in G of size k ? A set $X \subseteq V(G)$ is semi-independent if no two vertices of X are connected by an edge, or a path of length 2.

(B) (5 PTS.)

3EdgeColorable

Instance: A graph G .

Question: Is there a coloring of the edges of G using three colors, such that no two edges of the same color are adjacent?

Subset Sum

Instance: S : Set of positive integers. t : An integer number (target).

Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

(C) (5 PTS.)

3DM

Instance: X, Y, Z sets of n elements, and T a set of triples, such that $(a, b, c) \in T \subseteq X \times Y \times Z$.

Question: Is there a subset $S \subseteq T$ of n disjoint triples, s.t. every element of $X \cup Y \cup Z$ is covered exactly once.?

(See https://en.wikipedia.org/wiki/3-dimensional_matching#Example for an example.)

(D) (10 PTS.)

SET COVER

Instance: (S, \mathcal{F}, k) :

S : A set of n elements

\mathcal{F} : A family of m subsets of S , s.t. $\bigcup_{X \in \mathcal{F}} X = S$.

k : A positive integer.

Question: Are there k sets $S_1, \dots, S_k \in \mathcal{F}$ that cover S . Formally, $\bigcup_i S_i = S$?

(E) (10 PTS.)

CYCLE HATER.

Instance: An undirected graph $G = (V, E)$, and an integer $k > 0$.

Question: Is there a subset $X \subseteq V$ of at least k vertices, such that no cycle in G contains any vertex of X .

(F) (10 PTS.)

Many Meta-Spiders.

Instance: An undirected graph $G = (V, E)$ and an integer k .

Question: Are there k vertex-disjoint meta-spiders that visits all the vertices of G ?

A *meta-spider* in a graph G is defined by two vertices u, v (i.e., the head and tail of the meta-spider), and a collection Π of simple paths all starting in v and ending at u , that are vertex disjoint (except for u and v). The vertex set of such a spider is all the vertices that the paths of Π visit (including, of course, u and v).

2 (50 PTS.) Beware of algorithms carrying oracles.

Consider the following optimization problems, and for each one of them do the following:

- (I) (2 PTS.) State the natural decision problem corresponding to this optimization problem.
- (II) (3 PTS.) Either: (A) prove that this decision problem is **NP-COMplete** by showing a reduction from one of the **NP-COMplete** problems seen in class (if you already seen this problem in class state “seen in class” and move on with your life). (B) Alternatively, provide an efficient algorithm to solve this problem.
- (III) (5 PTS.) (You have to do this part only if you proved that the given problem is **NPC**, since otherwise this is not interesting.) Assume that you are given an algorithm that can solve the **decision** problem in polynomial time. Show how to solve the original optimization problem using this algorithm in polynomial time, and output the solution that realizes this optimal solution.

An example for the desired solution and how it should look like is provided in the last page.

- (A) (10 PTS.)

NO COVER

Instance: Collection C of subsets of a finite set S .

Target: Compute the maximum k , and the sets S_1, \dots, S_k in C , such that $S \not\subseteq \cup_{i=1}^k S_i$.

- (B) (10 PTS.)

TRIPLE HITTING SET

Instance: A *ground set* $U = \{1, \dots, n\}$, and a set $\mathcal{F} = \{U_1, \dots, U_m\}$ of subsets of U .

Target: Find the smallest set $S' \subseteq U$, such that S' hits all the sets of \mathcal{F} at least three times. Specifically, $S' \subseteq U$ is a *triple hitting set* if for all $U_i \in \mathcal{F}$, we have that S' contains at least three elements of U_i .

(Hint: Think about the **NPC** problem **HITTING SET**.)

- (C) (15 PTS.)

Max Inner Spanning Tree

Instance: Graph $G = (V, E)$.

Target: Compute the spanning tree \mathcal{T} in G where the number of vertices in \mathcal{T} of degree two or more is maximized.

(Hint: Think about the **NPC** problem **Hamiltonian Path**.)

- (D) (15 PTS.)

Cover by paths (edge disjoint).

Instance: Graph $G = (V, E)$.

Target: Compute the minimum number k of paths (not necessarily simple) π_1, \dots, π_k that are edge disjoint, and their union cover all the edges in G .

(Hint: Think about the Eulerian path problem.)

Example for a solution for the second problem

You are given the following optimization problem:

Max 3SAT

Instance: A boolean 3CNF formula F with n variables and m clauses.

Target: Compute the assignment to the variables of F , such that the number of clauses of F that are satisfied is maximized.

Solution:

- (I) The corresponding decision problem:

SAT Partial

Instance: A boolean CNF formula F with n variables and m clauses, and a parameter k .

Question: Is there an assignment to the variables of F that at least k clauses of F are satisfied (i.e., evaluate to true).

- (II) The decision problem **SAT Partial** is **NPC**. Indeed, given a positive instance, a certifier would be the desired assignment, and this assignment can be verified in polynomial time, as such the problem is in **NP**. As for the completeness part, observe that there is an immediate reduction from **SAT** to this problem.
- (III) Let $\text{algSAT}\partial\text{Sol}$ be the given solver for **SAT Partial**. Let F be the given input for **Max SAT** (i.e., the formula with n variables x_1, \dots, x_n and m clauses).

Let $\text{algCount}(F')$ be a function that performs a binary search for the largest number k , such that $\text{algSAT}\partial\text{Sol}(F', k)$ returns true. This requires $O(\log m)$ calls to $\text{algSAT}\partial\text{Sol}$, given that F' has m clauses.

The new algorithm $\text{algMaxSAT}(F)$ is the following:

- (i) If F is an empty formula, then return.
- (ii) $k \leftarrow \text{algCount}(F)$,
- (iii) Temporarily set $x_1 = 0$, and compute the resulting formula $F_{x_1=0}$ (all appearances of x_1 disappear, and all clauses that contain \bar{x}_1 are satisfied. Let k_0 be the number of clauses satisfied by this.
- (iv) $r_0 \leftarrow \text{algCount}(F_{x_1=0})$.
- (v) If $k_0 + r_0 = k$ then print “ $x_1 = 0$ ”, compute the remaining optimal assignment by calling $\text{algMaxSAT}(F_{x_1=0})$, Return.
- (vi) Temporarily set $x_1 = 1$, and compute the resulting formula $F_{x_1=1}$ (all appearances of \bar{x}_1 disappear, and all clauses that contain x_1 are satisfied.
- (vii) print “ $x_1 = 1$ ”
- (viii) Call recursively $\text{algMaxSAT}(F_{x_1=0})$, Return.

The correctness of this algorithm is easy to see, so we do not elaborate. As for the running time, observe that algMaxSAT calls only a single recursive call (i.e., it is a tail recursion, which implies that the main body of this function is performed n times. Clearly, all other operations in this function, ignoring the time to call algCount takes $O(|F|)$ time, where $|F|$ is the length of F . algCount calls $O(\log m)$ calls to the oracle. As such, overall, the running time of this algorithm is $O(n|F| + Tn \log m)$, where T is the running time of the oracle.

62.1.3. Homework 2

HW 2 (due Monday, 6pm, September 14, 2018)

CS 473: Theory II, Fall 2015

1 (50 PTS.) Is your overlap in vain?

(Dynamic programming.)

Let \mathcal{I} be a set of n closed intervals on the real line (assume they all have distinct endpoints). A set of intervals $B \subseteq \mathcal{I}$ is **admissible** if no point on the real line is covered by more than 3 intervals of B . Let $C(B)$ be the set of all points on the real line that are covered by two or more intervals of B . The **profit** of B , denoted by $\phi(B)$, is total length of $C(B)$.

Describe an algorithm, as fast as possible, that outputs the subset of \mathcal{I} that maximizes the profit, and is admissible.

(Hint: Look on the class slides for dynamic programming.)

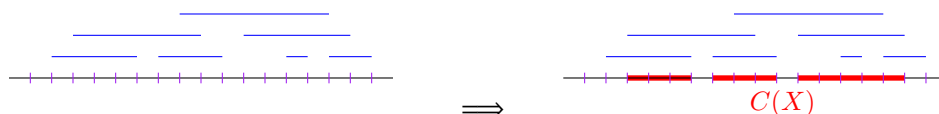


Figure 62.1: The set X , with profit $\|C(X)\| = 3 + 3 + 5 = 11$.

2 (50 PTS.) Diagonals and points. (Dynamic programming + DAGs and topological sort.^①)

- 2.A.** (10 PTS.) The **rank** of a vertex v in a DAG G , is the length of the longest path in DAG that starts in v . Describe a linear time algorithm (in the number of edges and vertices of G) that computes for all the vertices in G their rank.
- 2.B.** (10 PTS.) Prove that if two vertices $u, v \in V(G)$ have the same rank (again, G is a DAG), then the edges (u, v) and (v, u) are not in G .
- 2.C.** (10 PTS.) Using (B), prove that in any DAG G with n vertices, for any k , either there is a path of length k , or there is a set B of $\lfloor n/k \rfloor$ vertices in G that is **independent**; that is, there is no edge between any pair of vertices of B .
- 2.D.** (10 PTS.) Consider a set P of n points in the plane. The points of P are in general position – no two points have the same x or y coordinates. Consider a sequence S of points p_1, p_2, \dots, p_k of P , where $p_i = (x_i, y_i)$, for $i = 1, \dots, k$. The sequence S is **diagonal**, if either
 - for all $i = 1, \dots, k - 1$, we have $x_i < x_{i+1}$ and $y_i < y_{i+1}$, or
 - for all $i = 1, \dots, k - 1$, we have $x_i < x_{i+1}$ and $y_i > y_{i+1}$.
 Prove using (C) that there is always a diagonal of length $\lfloor \sqrt{n} \rfloor$ in P . Describe an algorithm, as fast as possible, that computes the longest diagonal in P .
- 2.E.** (10 PTS.) Using the algorithm of (D), describe a polynomial time algorithm that decomposes P into a set of $O(\sqrt{n})$ disjoint diagonals. Prove the correctness of your algorithm.

62.1.4. Homework 3

HW 3 (due Monday, 6pm, September 21, 2018)

CS 473: Theory II, Fall 2015

1 (50 PTS.) Packing heavy snakes on a tree.

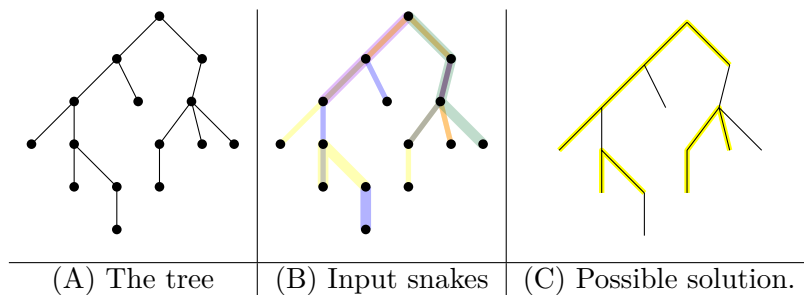
Let $G = (V, E)$ be a given rooted tree with n vertices. You are given also t snakes s_1, \dots, s_t , where a **snake** is just a simple path in the tree (with fixed vertices – a snake has only a single location where it can be

^①If you do not know what topological sort is, and how to compute it in linear time, then you need read on this stuff – you are suppose to know this, and you might be tested on this stuff.

placed). Every snake s_i has associated weight w_i , and your purpose is to pick the maximum weight subset S of snakes (of the given snakes) such that (i) the weight of the set is maximized, and (ii) no two snakes of S share a vertex. We refer to such a set S as a **packing** of snakes.

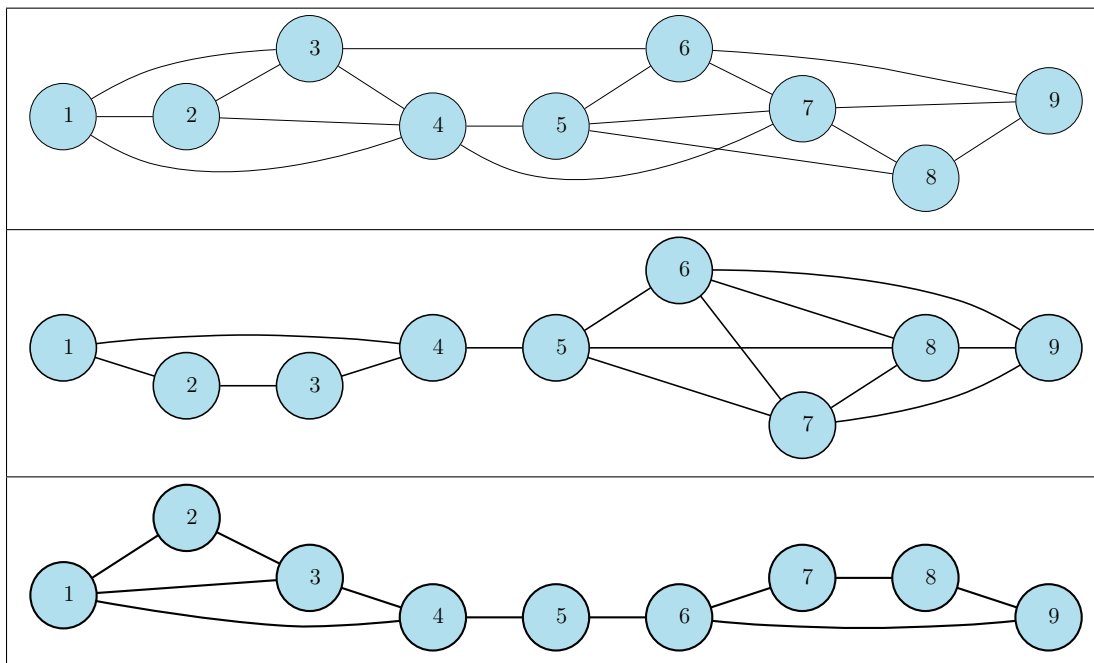
Describe an efficient algorithm (i.e., provide pseudo-code, etc), as fast as possible, for computing the maximum weight snake packing. (You can not assume G is a binary tree - a node might have arbitrary number of children.) What is the running time of your algorithm as function of n ?

For example, the following shows a tree with a possible snake packing.



2 (50 PTS.) Coloring silly graphs.

A graph G with $V(G) = \{1, \dots, n\}$ is k -silly, if for every edge $ij \in E(G)$, we have that $|i - j| \leq k$ (note, that it is not true that if $|i - j| \leq k$ then ij must be an edge in the graph!). Here are a few examples of a 3-silly graph:



Note, that only the last graph in the above example is 3-colorable.

Consider the decision problem **3COLORSillyGraph** of deciding if a given k -silly graphs is 3-colorable.

- 2.A.** (20 PTS.) Prove that **3COLORSillyGraph** is NP-COMplete.
- 2.B.** (30 PTS.) Provide an algorithm, as fast as possible, for solving **3COLORSillyGraph**. What is the dependency of the running time of your algorithm on the parameter k ?

In particular, for credit, your solution for this problem should be have polynomial time for k which is a constant. For full credit, the running time of your algorithm should be $O(f(k)n)$, where $f(k)$ is some function of k .

Hint: (A) Think about the vertices as ordered from left to right as above. Start with $k = 2$. Then, solve the problem for $k = 3, 4, \dots$. Hopefully, by the time you hit $k = 5$ you would be able to describe an algorithm for the general case.

62.1.5. Homework 4

HW 4 (due Monday, 6pm, October 5, 2018)

CS 473: Theory II, Fall 2015

1 (30 PTS.) TSP for k -silly graphs.

You are given the a graph G over the set of vertex $V = \llbracket n \rrbracket$, with the edge ij having weight $w(i, j) > 0$, for all $i < j$.

Given a parameter $k > 0$, describe an algorithm, as fast as possible, that computes the exact shortest TSP in G , assuming that the TSP can use an edge ij only if $|i - j| \leq k$. (For a fixed k the running time of your algorithm should be polynomial.)

[The recursive subproblem here is somewhat messy, but should be doable after the last homework. Figure it out, and the rest should be easy.]

2 (70 PTS.) Packing things.

2.A. (10 PTS.) Let \mathcal{I} be a given set of n closed intervals on the real line, and let $k > 0$ be a parameter. A **k -packing** of \mathcal{I} is a set of intervals $\mathcal{J} \subseteq \mathcal{I}$, such that no point is contained in more than k intervals of \mathcal{J} .

Describe an algorithm, as efficient as possible, that computes the largest subset of \mathcal{I} that is a k -packing. (For full credit your algorithm has to run in polynomial time in k and n .)

[Hint: Use a greedy algorithm and **prove** that it indeed outputs the optimal solution in this case. If you are unable to do the proof (which is a bit subtle) – no worries, you can still use the algorithm as a black box in the later parts of this problem.]

2.B. (30 PTS.) Let \mathcal{R} be a given set of axis-parallel rectangles in the plane, where the i th rectangle is of the form $[a_i, b_i] \times [c_i, d_i]$.

A **k -packing** is a set of rectangles $\mathcal{Q} \subseteq \mathcal{R}$, such that no point is contained in more than k rectangles of \mathcal{Q} .

Describe an approximation algorithm, as efficient as possible, that outputs a k -packing of \mathcal{R} of size $\geq \text{opt}/t$, where t is as small as possible and opt is the size of the largest k -packing of \mathcal{R} . What is the value of t of your algorithm in the worst case? What is the running time of your algorithm?

Provide a self contained proof of the approximation quality of your algorithm.

[Hint: See lecture slides.]

2.C. (30 PTS.) Let \mathcal{B} be a given set of axis-parallel boxes in three dimensions, where the i th box is of the form $[a_i, b_i] \times [c_i, d_i] \times [e_i, f_i]$.

A **k -packing** is a set of boxes $\mathcal{C} \subseteq \mathcal{B}$, such that no point is contained in more than k boxes of \mathcal{C} .

Describe an approximation algorithm, as efficient as possible, that outputs a k -packing of \mathcal{B} of size $\geq \text{opt}/t$, where t is as small as possible and opt is the size of the maximum k -packing of \mathcal{B} . What is the value of t of your algorithm in the worst case? What is the running time of your algorithm?

Provide a self contained proof of the approximation quality of your algorithm.

[Hint: Use (B).]

HW 5 (due Monday, 6pm, October 12, 2018)

Collaboration Policy: This homework can be worked in groups of up to three students. Submission is online on moodle.

1 (40 PTS.) Collapse and shrink.

1.A. (5 PTS.) You are given an undirected graph with n vertices and m edges, with positive weights on the edges (for simplicity, you can assume all the weights are distinct). Consider the procedure that given a weight x , outputs the graph $G_{<x}$ that results from removing all the edges in G that have weight larger than (or equal to) x . Describe (shortly – no need for pseudo code) an algorithm for computing $G_{<x}$. How fast is your algorithm?

The graph $G_{<x}$ might not be connected – how would you compute its connected components?

1.B. (5 PTS.) Consider the procedure that receives as input an undirected weighted graph G , and a partition \mathcal{V} of the vertices of G into k disjoint sets V_1, \dots, V_k . The *meta graph* $G(\mathcal{V})$ of G induced by \mathcal{V} is a graph having k vertices, v_1, \dots, v_k , where $v_i v_j$ has an edge if and only if, there is an edge between some vertex of V_i and some vertex of V_j . The weight of such an edge $v_i v_j$ is the minimum weight of any edge between vertices in V_i and vertices in V_j .

Describe an algorithm, as fast as possible, for computing the meta-graph $G(\mathcal{V})$. You are not allowed to use hashing for this question, but you can use that **RadixSort** works in linear time (see wikipedia if you do not know **RadixSort**). How fast is your algorithm?

1.C. (10 PTS.) Consider the randomized algorithm that starts with a graph G with m edges and n vertices. Initially it sets $G_0 = G$. In the i th iteration, it checks if G_{i-1} is a single edge. If so, it stops and outputs the weight of this edge. Otherwise, it randomly choose an edge $e_i \in E(G_{i-1})$. It then computes the graph $H_i = (G_{i-1})_{<w(e_i)}$, as described above.

- If the graph H_i is connected then it sets $G_i = H_i$ and continues to the next iteration.
- Otherwise, H_i is not connected, then it computes the connected components of H_i , and their partition \mathcal{V}_i of the vertices of G_{i-1} (the vertices of each connected component are a set in this partition). Next, it sets G_i to be the meta-graph $G_{i-1}(\mathcal{V}_i)$.

Let m_i be the number of edges of the graph G_i . Prove that if you know the value of m_{i-1} , then $\mathbb{E}[m_i] \leq (7/8)m_{i-1}$ (a better constant is probably true). Conclude that $\mathbb{E}[m_i] \leq (7/8)^i m$.

1.D. (15 PTS.) What is the expected running time of the algorithm describe above? **Prove** your answer. (The better your bound is, the better it is.)

1.E. (5 PTS.) What does the above algorithm computes, as far as the original graph G is concerned?

2 (30 PTS.) Majority tree

Consider a uniform rooted tree of height h (every leaf is at distance h from the root). The root, as well as any internal node, has 3 children. Each leaf has a boolean value associated with it. Each internal node returns the value returned by the majority of its children. The evaluation problem consists of determining the value of the root; at each step, an algorithm can choose one leaf whose value it wishes to read.

2.A. (15 PTS.) Show that for any deterministic algorithm, there is an instance (a set of boolean values for the leaves) that forces it to read all $n = 3^h$ leaves. (hint: Consider an adversary argument, where you provide the algorithm with the minimal amount of information as it request bits from you. In particular, one can devise such an adversary algorithm.).

- 2.B.** (10 PTS.) Consider the recursive randomized algorithm that evaluates two subtrees of the root chosen at random. If the values returned disagree, it proceeds to evaluate the third sub-tree. If they agree, it returns the value they agree on.

Write an explicit exact formula for the expected number of leaves being read, for a tree of height $h = 1$, and height $h = 2$.

- 2.C.** (5 PTS.) Using (B), prove that the expected number of leaves read by the algorithm on any instance is at most $n^{0.9}$.

3 (30 PTS.) Attack of the edge killers.

Let \mathcal{T}_h denote the full balanced binary tree with 2^h leaves. Due to eddies in the space-time continuum, every edge get deleted with probability half (independently) – let $\mathcal{T}'_h = \text{leftover}(\mathcal{T}_h)$ denote the remaining tree (rooted at the original root). The remaining tree \mathcal{T}'_h is *usable*, if there is a path from the root of the tree to one of the original leaves of the tree.

- 3.A.** (10 PTS.) Let ρ_1 be the probability that $\text{leftover}(\mathcal{T}_1)$ is a usable. What is the value of ρ_1 ? Prove your answer.
- 3.B.** (10 PTS.) Let ρ_h be the probability that a tree \mathcal{T}'_h is usable. Give a recursive formula for the value of ρ_h as a function of ρ_{h-1} .
- 3.C.** (10 PTS.) Prove, by induction, that $\rho_h \geq 1/(h + 1)$.

62.1.7. Homework 6

HW 6 (due Monday, 6pm, October 19, 2018)

CS 473: Theory II, Fall 2015

Collaboration Policy: This homework can be worked in groups of up to three students. Submission is online on moodle.

1 (50 PTS.) Collect the min-cut.

Consider the algorithm that given a graph $G = (V, E)$ with n vertices and m edges, starts with the empty forest $H_0 = (V, \emptyset)$ where every vertex is a single node tree.

In the i th epoch, the algorithm picks an edge e randomly and uniformly from the original set of edges E , and check:

- If the vertices of e belong to different trees of the forest H_{i-1} , then it is *useful*. If so the algorithm adds e to this graph, to form the new forest H_i , and continues to the next epoch.
- If the edge connects two vertices that are already in the same tree of H_{i-1} , then this edge is *useless*.

The algorithm continues in this fashion, till it computes H_{n-2} , which has exactly two trees.

- 1.A.** (5 PTS.) Let H_{i-1} be as above. Assume that the minimum cut in G is of size k , and H_{i-1} does not contain any edges of this min-cut. Let U_i be the set of edges of G that are useful as far as H_{i-1} is concerned. Prove, a lower bound, as large as possible, on U_i as a function of i, k, n and m .
- 1.B.** (5 PTS.) Provide a lower bound, as large as possible, on the probability that a random edge of G is useful for H_{i-1} . (Use (A).)

- 1.C. (10 PTS.) Let N_i be the number of edges sampled in the i th epoch till a useful edge for H_{i-1} is found. Prove an upper bound, as small as possible, on $\mathbb{E}[N_i]$. (Use (B).)
- 1.D. (10 PTS.) Consider the forest H_{n-2} . Prove a lower bound as large as possible, on the probability that H_{n-2} (with its two trees), represents a min-cut of G . Provide a full and self contained proof of this lower bound (i.e., you can not refer to the class notes/web/the universe for a proof of this).
- 1.E. (10 PTS.) The expected number of edges inspected by the above algorithm to construct H_{n-2} is

$$\alpha = \mathbb{E}[N_1 + N_2 + \dots + N_{n-2}].$$

Provide and prove an upper bound, as small as possible, on the value of α (as a function of k, n and m).

- 1.F. (10 PTS.) (Hard.) Provide a randomized algorithm, as fast as possible, that computes the connected components of H_{n-2} (and its associated vertex cut [i.e., it does not compute the edges in the cut themselves]). Prove a bound on the expected running time of your algorithm.

For full credit, the expected running time of your algorithm should be $O(\alpha)$, and should not use union-find and hashing^①. A slower algorithm would get half the points (and then this part is not that hard).

(Computing the edges of the cut themselves, once the vertex cut is known, can be easily done in $O(m)$ time.)

Note: You do not know what the value of k is!

2 (50 PTS.) Disjoint paths.

Let $G = (V, E)$ be a directed graph, with n vertices and m edges. Let s and t be two vertices in G . For the sake of simplicity, assume that there are no u, v such that (u, v) and (v, u) are in G .

A set of paths \mathcal{P} in G is **edge disjoint** if no two paths in \mathcal{P} share an edge.

- 2.A. (10 PTS.) Let \mathcal{P} be a set of k edge disjoint paths from s to t . Let π be a path from s to t (which is not in \mathcal{P}). Prove or disprove: There is a set \mathcal{P}' of k edge disjoint paths from s to t in G that contains π as one of the paths.
- 2.B. (10 PTS.) Let \mathcal{P} be a given set of edge disjoint paths from s to t . Let $E(\mathcal{P})$ be the set of edges used by the paths of \mathcal{P} . The **leftover graph** $G_{\mathcal{P}}$ is the graph where $(u, v) \in E(G_{\mathcal{P}})$ if $(u, v) \in E(G) \setminus E(\mathcal{P})$ or $(v, u) \in E(\mathcal{P})$ (note that the edge (u, v) is the reverse edge of (v, u)).

Describe how to compute the leftover graph in $O(m)$ time (no hashing please).

- 2.C. (5 PTS.) Let \mathcal{P} be a set of k edge disjoint paths from s to t . Let π be a path in $G_{\mathcal{P}}$ from s to t . Prove that there is a set of \mathcal{P}' of $k + 1$ edge disjoint paths from s to t in G . In particular, show how to compute \mathcal{P}' given \mathcal{P} and π in $O(m)$ time. (For credit, your solution should be self contained and not use min-cut max-flow theorem or network flow algorithms.)
- 2.D. (5 PTS.) The natural greedy algorithm for computing the maximum number of edge disjoint paths in G , works by starting from an empty set of paths \mathcal{P}_0 , then in the i th iteration, it finds a path π_i in the leftover graph $G_{\mathcal{P}_{i-1}}$ from s to t , and then compute a set of i edge-disjoint paths \mathcal{P}_i , by using the algorithm of (C) on \mathcal{P}_{i-1} and π_i .

Assume the algorithm stops in the $(k + 1)$ th iteration, because there is not path from s to t in $G_{\mathcal{P}_k}$. We want to prove that the k edge-disjoint paths computed (i.e., \mathcal{P}_k) is optimal, in the sense that there is no larger set of edge-disjoint paths from s to t in G .

To this end, let S be the set of vertices that are reachable from s in $G_{\mathcal{P}_k}$. Let $T = V(G) \setminus S$ (observe that $t \in T$). Prove, that every path in \mathcal{P}_k contains exactly one edge of

$$(S, T) = \{(u, v) \in E(G) \mid u \in S, v \in T\}.$$

^①Note that I see how hashing helps here.

(Hint: Prove first that no path of \mathcal{P}_k can use an edge of the “reverse” set (T, S) .)

- 2.E. (5 PTS.) Consider the setting of (D). Prove that $k = |\mathcal{P}_k| = |(S, T)|$.
 - 2.F. (5 PTS.) Consider any set X of edge-disjoint paths in G from s to t . Prove that any path π of X must contain at least one edge of (S, T) .
 - 2.G. (5 PTS.) Prove that the greedy algorithm described in (D) indeed computes the largest possible set of edge-disjoint paths from s to t in G .
 - 2.H. (5 PTS.) What is the running time of the algorithm in (D), if there are at most k edge-disjoint path in G ?
-

62.1.8. Homework 7

HW 7 (due Monday, 6pm, October 26, 2018)

CS 473: Theory II, Fall 2015

1 (100 PTS.) Much matching about nothing.

- 1.A. (20 PTS.) Read about Hall’s theorem and its proof (for example, on wikipedia). A graph is *regular* if every vertex has the same number of edges incident to it (and this number is non-zero). Prove that every regular bipartite graph has a perfect matching using Hall’s theorem. A matching is *perfect* if all vertices are incident on a matching edge.
 - 1.B. (20 PTS.) Prove that the edges of a k -regular bipartite graph (i.e., a graph where every vertex has degree k) can be colored using k colors, such that no two edges of the same color share a vertex. Describe an efficient algorithm for computing this coloring.
 - 1.C. (20 PTS.) Given a bipartite graph $G = (V, E)$, describe an algorithm, as efficient as possible, for computing a k -regular bipartite graph that is contained in G , and uses all the vertices of G . Naturally, the algorithm has to return the largest k for which such a graph exists, together with this regular subgraph. Prove the correctness of your algorithm.
(Hint: Modify the simple algorithm seen in class for computing maximum matching in bipartite graphs. Other natural algorithms do not seem to work for this problem.)
 - 1.D. (20 PTS.) You are given an algorithm `alg` that in $T(n, m)$ time, can return the largest cardinality matching in a graph with n vertices and m edges. You are given a complete graph G on n vertices, with distinct weights on the edges. Describe an algorithm, as fast as possible, that uses (and does relatively little else) `alg`, and computes the minimum weight w , such that if we remove from G all the edges heavier than w , then the remaining graph $G_{\leq w}$ contains a perfect matching.
 - 1.E. (20 PTS.) You are given a set of n clients, and m shops. A specific shop i can serve at most c_i clients. Every client, has the set of shops they are willing to shop in. Describe an algorithm, as efficient as possible, that decides for a client which shop to use, such that no shop exceeds its capacity [if such a solution exists, naturally]. (You are not allowed to use hashing or network flow in solving this problem.)
-

62.1.9. Homework 8

HW 8 (due Monday, 6pm, November 9, 2018)

CS 473: Theory II, Fall 2015

1 (25 PTS.) LP I

Let L be a linear program given in slack form, with n nonbasic variables N , and m basic variables B . Let N' and B' be a different partition of $N \cup B$, such that $|N' \cup B'| = |N \cup B|$. Show a polynomial time algorithm that computes an equivalent slack form that has N' as the nonbasic variables and B' as the basic variables. How fast is your algorithm?

2 (25 PTS.) Multi mini flowy.

In the *minimum-cost multicommodity-flow problem*, we are given a directed graph $G = (V, E)$, in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$ and a cost $\alpha(u, v)$. As in the multicommodity-flow problem (Chapter 29.2, CLRS), we are given k different commodities, K_1, K_2, \dots, K_k , where commodity i is specified by the triple $K_i = (s_i, t_i, d_i)$. Here s_i is the source of commodity i , t_i is the sink of commodity i , and d_i is the demand, which is the desired flow value for commodity i from s_i to t_i . We define a flow for commodity i , denoted by f_i , (so that $f_i(u, v)$ is the flow of commodity i from vertex u to vertex v) to be a real-valued function that satisfies the flow-conservation, skew-symmetry, and capacity constraints. We now define $f(u, v)$, the **aggregate flow** to be sum of the various commodity flows, so that $f(u, v) = \sum_{i=1}^k f_i(u, v)$. The aggregate flow on edge (u, v) must be no more than the capacity of edge (u, v) .

The cost of a flow is $\sum_{u,v \in V} f(u, v)\alpha(u, v)$, and the goal is to find the feasible flow of minimum cost. Express this problem as a linear program.

3 (25 PTS.) Some calculations required,

Provide *detailed* solutions for the following problems, showing each pivoting stage separately.

$$\text{maximize } 6x_1 + 8x_2 + 5x_3 + 9x_4$$

subject to

$$2x_1 + x_2 + x_3 + 3x_4 \leq 5$$

$$x_1 + 3x_2 + x_3 + 2x_4 \leq 3$$

$$x_1, x_2, x_3, x_4 \geq 0.$$

4 (25 PTS.) Some calculations required,

$$\text{minimize } 4x_{12} + 6x_{13} + 9x_{14} + 2x_{23} + 7x_{24} + 3x_{34}$$

subject to

$$x_{12} + x_{13} + x_{14} \geq 1$$

$$-x_{12} + x_{23} + x_{24} = 0$$

$$-x_{13} - x_{23} + x_{34} = 0$$

$$x_{14} + 3x_{24} + x_{34} \leq 1$$

$$x_{12}, x_{13}, \dots, x_{34} \geq 0.$$

62.1.10. Homework 9

HW 9 (due Monday, 6pm, November 16, 2018)

CS 473: Theory II, Fall 2015

1 (50 PTS.) Many cover problem.

Let (X, \mathcal{F}) be a set system with $n = |X|$ elements, and $m = |\mathcal{F}|$ sets. Furthermore, for every elements $u \in X$, there is a positive integer c_u . In the **ManyCover** problem, you need to find a minimum number of sets $\mathcal{K} \subseteq \mathcal{F}$, such that every element of $u \in X$ is covered at least c_u times. (You are not allowed to use the same set more than once in the cover \mathcal{K} .)

- 1.A. (10 PTS.) Let y_1, \dots, y_n be numbers in $[0, 1]$, such that $t = \sum_{i=1}^n y_i \geq 3$. Let Y_i be a random variable that is one with probability y_i (and zero otherwise), for all i . Prove, that $\mathbb{P}[t/2 \leq \sum_i Y_i \leq 3t/2] \geq 1 - f(t)$, where $f(t)$ is a function that goes to zero as t increases (the smaller the $f(t)$ is, the better your solution is).
- 1.B. (20 PTS.) Describe in detail a polynomial approximation algorithms that provides a $O(\log n)$ approximation to the optimal solution for this problem (as usual, you can assume that solving a polynomially sized LP takes polynomial time). (Hint: See the algorithm provided in class for set Cover.)
- 1.C. (20 PTS.) Provide a polynomial time algorithm, that provides a $O(1)$ approximation to the problem, if we know that $c_u \geq \log n$, for all $u \in X$.

2 (50 PTS.) Independent set via interference.

Let $G = (V, E)$ be a graph with n vertices, and m edges. Assume we have a feasible solution to the natural **independent set** for G :

$$\begin{array}{ll} \max & \sum_{v \in V} x_v \\ \text{s.t.} & x_v + x_u \leq 1 \quad \forall uv \in E \\ & x_u \geq 0 \quad \forall u \in V. \end{array}$$

This solution assigns the value \hat{x}_v to x_v , for all v . Furthermore, assume that $\alpha = \sum_{v \in V} \hat{x}_v$ and, importantly, $\sum_{uv \in E} \hat{x}_u \hat{x}_v \leq \alpha/8$.

- 2.A. (10 PTS.) Let S be a subset of the vertices of the graph being generated by picking (independently) each vertex $u \in V$ to be in S with probability \hat{x}_u . Prove, that with probability at least $9/10$, we have $|S| \geq \alpha/2$ (you can safely assume that $\alpha \geq n_0$, where n_0 is a sufficiently large constant).
- 2.B. (20 PTS.) Let G_S be the induced subgraph of G on S . Prove that $\mathbb{P}[|E(G_S)| \geq \alpha/4] \leq 1/2$.
- 2.C. (20 PTS.) Present an algorithm, as fast as possible, that outputs an independent set in G of size at least $c\alpha$, where $c > 0$ is some fixed constant. What is the running time of your algorithm? What is the value of c for your algorithm?

62.1.11. Homework 10

HW 10 (due Monday, 6pm, November 30, 2018)

CS 473: Theory II, Fall 2015

1 (50 PTS.) Adding numbers.

In the following we work with multisets. For an element x in a multiset X , we denote by $\#_X(x)$ its **multiplicity** in X . For a multiset S , we denote by $\text{set}(S)$ the set of distinct elements appearing in S . The

size of a multiset S is the number of distinct elements in S (i.e., $|\text{set}(S)|$). The **cardinality** of S , denoted by $\text{card}(S) = \sum_{s \in S} \#_S(s)$.^①

We use $\llbracket x : y \rrbracket = \{x, x+1, \dots, y\}$ to denote the set of integers in the interval $[x, y]$. Similarly, $\llbracket \mathbf{U} \rrbracket = \llbracket 1 : \mathbf{U} \rrbracket$.

Abusing notation, we denote by $S \cap \llbracket x : y \rrbracket$ the multiset resulting from removing from S all the elements that are outside $\llbracket x : y \rrbracket$. We denote that a multiset S has all its elements in the interval $\llbracket x : y \rrbracket$ by $S \subseteq \llbracket x : y \rrbracket$.

For two multisets X and Y , we denote by $X \oplus Y$ the multiset with the ground set being

$$\{x + y \mid x \in X \text{ and } y \in Y\},$$

where the multiplicity of $x + y$ is the number of different ways to get this sum (in particular, $x \in X$ and $y \in Y$ contribute the element $x + y$ with multiplicity $\#_X(x) \cdot \#_Y(y)$ to the resulting multiset).

For a multiset S of integers, let $\Sigma_S = \sum_{\alpha \in S} \#_S(\alpha) * \alpha$ denote the total **sum** of the elements of S . The multiset of all **subset sums** is

$$\sum(S) = \text{set}(\{\Sigma_T \mid T \subseteq S\}).$$

Here, we would be interested in the subset sums up to \mathbf{U} ; that is $\sum_{\leq \mathbf{U}}(S) = \sum(S) \cap \llbracket 0 : \mathbf{U} \rrbracket$. Formally, we want to compute $\text{set}(\sum_{\leq \mathbf{U}}(S))$.

- 1.A.** (5 PTS.) Let $S \subseteq \llbracket u \rrbracket$ be a multiset of cardinality n . Describe an algorithm that computes, in $O(n\mathbf{U})$ time, the set of subset sums up to \mathbf{U} ; that is, $\text{set}(\sum_{\leq \mathbf{U}}(S))$.
- 1.B.** (5 PTS.) Given two sets $S, T \subseteq \llbracket \mathbf{U} \rrbracket$, present an algorithm that computes $S \oplus T$ in $O(\mathbf{U} \log \mathbf{U})$ time.
- 1.C.** (5 PTS.) Given a multiset S of non-negative integers of cardinality n , present an algorithm that, in $O(n \log n)$ time, computes a multiset T , such that $\text{set}(\sum(S)) = \text{set}(\sum(T))$, and no number appears in T more than twice.
- 1.D.** (5 PTS.) Given a multiset $X \subseteq \llbracket \Delta \rrbracket$, of cardinality n , show how to compute the $\sum(X)$ in time $O(n\Delta \log(n\Delta) \log n)$.
- 1.E.** (5 PTS.) Given a multiset $S \subseteq \llbracket x : 2x \rrbracket$ of cardinality n' , which is at most $\lfloor \mathbf{U}/x \rfloor$, show how to compute all possible subset sums of S , in time $O(\mathbf{U} \log \mathbf{U} \log n')$.
- 1.F.** (5 PTS.) Given a multiset $S \subseteq \llbracket x : 2x \rrbracket$ of cardinality $n \geq \mathbf{U}/x$, show how to compute all possible subset sums of S that are at most \mathbf{U} (i.e., $\sum_{\leq \mathbf{U}}(S)$). The running time of your algorithm should be $O(nx \log^2 \mathbf{U})$. (Hint: Use **1.E.**)
- 1.G.** (10 PTS.) [Hard.] Given a set $S \subseteq \llbracket x : x + \ell \rrbracket$ of cardinality n , show an algorithm that compute all possible subset sums of multisets of cardinality at most t in S , in time $O(n\ell t \log(n\ell t) \log n)$.
- 1.H.** (10 PTS.) [Hard.] Given a multiset $S \subseteq \llbracket \mathbf{U} \rrbracket$ of cardinality n , show how to compute $\sum_{\leq \mathbf{U}}(S)$ in $O(\mathbf{U}\sqrt{n} \log^2 \mathbf{U})$ time. For credit, your solution has to be self contained, and use the above. (Hint: Partition the set S into intervals $I_i = \llbracket a_i : b_i \rrbracket$, for $i = 1, \dots, t$ (for some t), and compute the subset sums of the sets $S_i = S \cap \llbracket a_i : b_i \rrbracket$ using the above. Then combine them into subsets sums of all the numbers of S , again using the above. Observe that for S_i we care only about subsets sums involving at most \mathbf{U}/a_i terms (why?).)

In the above, you can assume that $n < \mathbf{U}$.

2 (25 PTS.) Sorting networks stuff

^①For more information about multisets, see wikipedia.

- 2.A.** (5 PTS.) Prove that an n -input sorting network must contain at least one comparator between the i th and $(i + 1)$ st lines for all $i = 1, 2, \dots, n - 1$.
- 2.B.** (10 PTS.) Prove that in a sorting network for n inputs, there must be at least $\Omega(n \log n)$ gates. For full credit, your answer should be short, and self contained (i.e., no reduction please).
[As an exercise, you should think why your proof does not imply that a regular sorting algorithm takes $\Omega(n \log n)$ time in the worst case.]
- 2.C.** (5 PTS.)
Suppose that we have $2n$ elements $\langle a_1, a_2, \dots, a_{2n} \rangle$ and wish to partition them into the n smallest and the n largest. Prove that we can do this in constant additional depth after separately sorting $\langle a_1, a_2, \dots, a_n \rangle$ and $\langle a_{n+1}, a_{n+2}, \dots, a_{2n} \rangle$.
- 2.D.** (5 PTS.)
Let $S(k)$ be the depth of a sorting network with k inputs, and let $M(k)$ be the depth of a merging network with $2k$ inputs. Suppose that we have a sequence of n numbers to be sorted and we know that every number is within k positions of its correct position in the sorted order, which means that we need to move each number at most $(k - 1)$ positions to sort the inputs. For example, in the sequence 3 2 1 4 5 8 7 6 9, every number is within 3 positions of its correct position. But in sequence 3 2 1 4 5 9 8 7 6, the number 9 and 6 are outside 3 positions of its correct position.
Show that we can sort the n numbers in depth $S(k) + 2M(k)$. (You need to prove your answer is correct.)

3 (25 PTS.) Computing Polynomials Quickly

In the following, assume that given two polynomials $p(x), q(x)$ of degree at most n , one can compute the polynomial remainder of $p(x) \bmod q(x)$ in $O(n \log n)$ time. The *remainder* of $r(x) = p(x) \bmod q(x)$ is the unique polynomial of degree smaller than this of $q(x)$, such that $p(x) = q(x) * d(x) + r(x)$, where $d(x)$ is a polynomial.

Let $p(x) = \sum_{i=0}^{n-1} a_i x^i$ be a given polynomial.

- 3.A.** (6 PTS.) Prove that $p(x) \bmod (x - z) = p(z)$, for all z .
- 3.B.** (6 PTS.) We want to evaluate $p(\cdot)$ on the points x_0, x_1, \dots, x_{n-1} . Let

$$P_{ij}(x) = \prod_{k=i}^j (x - x_k)$$

and

$$Q_{ij}(x) = p(x) \bmod P_{ij}(x).$$

Observe that the degree of Q_{ij} is at most $j - i$.

Prove that, for all x , $Q_{kk}(x) = p(x_k)$ and $Q_{0,n-1}(x) = p(x)$.

- 3.C.** (6 PTS.) Prove that for $i \leq k \leq j$, we have

$$\forall x \quad Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$$

and

$$\forall x \quad Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x).$$

- 3.D.** (7 PTS.) Given an $O(n \log^2 n)$ time algorithm to evaluate $p(x_0), \dots, p(x_{n-1})$. Here x_0, \dots, x_{n-1} are n given real numbers.

62.1.12. Homework 11

HW 11 (due Monday, 6pm, December 7, 2018)

CS 473: Theory II, Fall 2015

Collaboration Policy: For this homework, every student should do this homework on their own. Submissions should be done individually.

We will not provide a solution for this homework. Also, we do not provide the latex file for this homework.

1 (10 PTS.) Multiple choice

For each of the questions below choose the most appropriate answer. No **IDK** credit for this question!

- 1.A. Given a graph G . Deciding if there is an independent set X in G , such that $G \setminus X$ (i.e., the graph G after we remove the vertices of X from it) is bipartite can be solved in polynomial time.

False: True: Answer depends on whether $P = NP$:

- 1.B. Consider any two problems X and Y both of them in **NPC**. There always exists a polynomial time reduction from X to Y .

False: True: Answer depends on whether $P = NP$:

- 1.C. Given a graph represented using adjacency lists, it can be converted into matrix representation in linear time in the size of the graph (i.e., linear in the number of vertices and edges of the graph).

False: True: Answer depends on whether $P = NP$:

- 1.D. Given a **2SAT** formula F , there is always an assignment to its variables that satisfies at least $(7/8)m$ of its clauses. False: True: Answer depends on whether $P = NP$:

- 1.E. Given a graph G , deciding if contains a clique made out of 165 vertices is **NP-COMplete**. False: True: Answer depends on whether $P = NP$:

- 1.F. Given a directed graph G with positive weights on the edges, and a number k , finding if there is simple path in G from s to t (two given vertices of G) with weight $\geq k$, can be done in polynomial time.

False: True: Answer depends on whether $P = NP$:

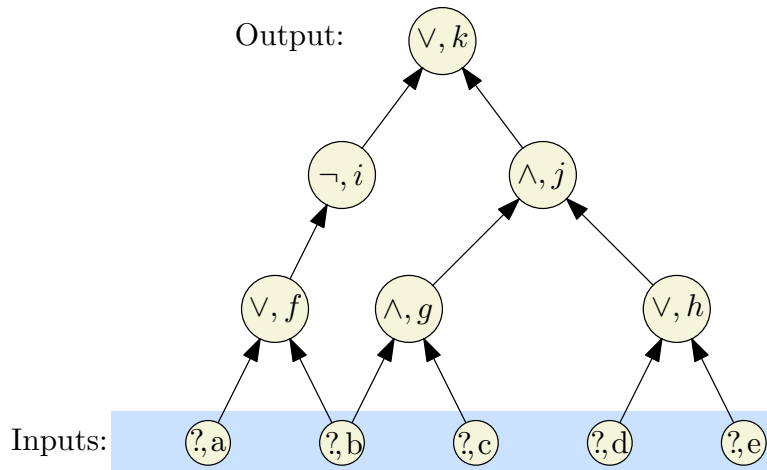
- 1.G. Given a directed graph G with (positive or negative) weights on its edges, computing the shortest walk from s to t in G can be done in polynomial time. False: True: Answer depends on whether $P = NP$:

2 (10 PTS.) Short Questions.

- 2.A. (8 PTS.) Give a tight asymptotic bound for each of the following recurrences.

2.A.i. (4 PTS.) $A(n) = A(n - 3 \lceil \log n \rceil) + A(\lceil \log n \rceil) + \log n$, for $n > 2$
and $A(1) = A(2) = 1$.

- 2.A.ii. (4 PTS.) $B(n) = 12B(\lfloor n/4 \rfloor) + B(\lfloor n/2 \rfloor) + n^2$, for $n > 10$ and $B(i) = 1$ for $1 \leq i \leq 10$.
- 2.B. (8 PTS.) Convert the following boolean circuit (i.e., an instance of **Circuit-SAT**) into a CNF formula (i.e., an instance of SAT) such that the resulting formula is satisfiable if and only if the circuit sat instance is satisfiable. Use $x_a, x_b, x_c, x_d, \dots$ as the variable names for the corresponding gates in the drawing. (You may need additional variables.) Note, that a node (\wedge, g) in the figure below denotes an and gate, where g is its label.



- 3 (20 PTS.) Balancing vampires.
- Sadly, there are n vampires p_1, \dots, p_n in Champaign. The i th vampire has a score $w_i \geq 0$ describing how well it can climb mountains. You want to divide the vampires into two teams, and you want the division of teams to be as fair as possible. The score of a team is the sum of the scores of all the vampires in that team. We want to minimize the differences of the scores of the two teams. Assume that for all i , $w_i \leq W$.
- 3.A. (10 PTS.) Given integers $\alpha, \beta \geq 0$, and T_α, T_β , such that $\alpha + \beta = n$, describe an algorithm, as fast as possible, to compute the partition into two teams, such that the first team has α players of total score T_α , and the second team has β players with total score T_β . What is the running time of your algorithm? (For any credit, it has to be polynomial in n and W .)
(To simplify things, you can solve the decision version problem first, and describe shortly how to modify it to yield the desired partition.)
- 3.B. (5 PTS.) Describe an algorithm, as fast as possible, to compute the scores of the two teams in an optimal division that is as balanced as possible, when requiring that the two teams have exactly the same number of players (assume n is even). What is the running time of your algorithm?
- 3.C. (5 PTS.) State **formally** the decision version of the problem in (B), and prove that it is **NP-COMplete**. (There are several possible solutions for this part – pick the one you find most natural. Note, that the teams must have the same number of players.)

- 4 (12 PTS.) MAX Cut and MAX 2SAT.

The **Max CUT** problem is the following:

MAX Cut

Instance: Undirected graph G with n vertices and m edges, and an integer k .

Question: Is there an undirected cut in G that cuts at least k edges?

MAX 2SAT

Instance: A 2CNF formula F , and an integer k .

Question: Is there a truth assignment in F that satisfies at least k clauses.

You are given that **MAX Cut** is **NP-COMplete**. Prove that **MAX 2SAT** is **NP-COMplete** by a reduction to/from **MAX Cut** (be sure to do the reduction in the right direction! [and please do not ask us what is the right direction – this is part of the problem]).

Hint: Think about how to encode a cut, by associating a boolean variable with each vertex of the graph. It might be a good idea to verify your answer by considering a graph with two vertices and a single edge between them and checking all possibilities for this case.

5 (10 PTS.) Billboards are forever.

Consider a stretch of Interstate-57 that is m miles long. We are given an ordered list of mile markers, x_1, x_2, \dots, x_n in the range 0 to m , at each of which we are allowed to construct billboards (suppose they are given as an array $X[1 \dots n]$). Suppose we can construct billboards for free, and that we are given an array $R[1 \dots n]$, where $R[i]$ is the revenue we would receive by constructing a billboard at location $X[i]$. Given that state law requires billboards to be at least 5 miles apart, describe an algorithm, as fast as possible, to compute the maximum revenue we can acquire by constructing billboards.

What is the running time of your algorithm? (For full credit, your algorithm has to be as fast as possible.)

6 (10 PTS.) Best edge ever.

You are given a directed graph G with n vertices and m edges. For every edge $e \in E(G)$, there is an associated weight $w(e) \in \mathbb{R}$. For a path (not necessarily simple) π in G , its **quality** is $W(\pi) = \max_{e \in \pi} w(e)$. We are interested in computing the highest quality walk in G between two given vertices (say s and t). Either **prove** that computing such a walk is **NP-HARD**, or alternatively, provide an algorithm (and **prove** its correctness) for this problem (the algorithm has to be as fast as possible – what is the running time of your algorithm?).

7 (10 PTS.) Dominate this.

You are given a set of intervals $\mathcal{I} = \{I_1, \dots, I_n\}$ on the real line (assume all with distinct endpoints) – they are given in arbitrary order (i.e., you can not assume anything on the ordering). Consider the problem of finding a set of intervals $\mathcal{K} \subseteq \mathcal{I}$, as small as possible, that dominates all the other intervals. Formally, \mathcal{K} **dominates** \mathcal{I} , if for every interval $I \in \mathcal{I}$, there is an interval $K \in \mathcal{K}$, such that I intersects K .

Describe an algorithm (as fast as possible) that computes such a minimal dominating set of \mathcal{I} . What is the running time of your algorithm? Prove the correctness of your algorithm.

62.2. Midterm

1 COLORING GRAPHS. (25 PTS.)

For a set S , a **balanced coloring** ϕ assigns every element in S a label that is either -1 or $+1$. For a set $F \subseteq S$, its **balance** is $\phi(F) = \sum_{x \in F} \phi(x)$.

Prove that the following problem is **NP-COMplete**:

Balanced coloring of a set system.

Instance: (S, \mathcal{F}) : S is a set of n elements, and \mathcal{F} is a family of subsets of S .

Question: Is there a balanced coloring ϕ of S , such that for any set $F \in \mathcal{F}$, we have;

- If $|F|$ is even then $\phi(F) = 0$.
- If $|F|$ is odd then $\phi(F) \geq -1$.

(Hint: Think what a balanced coloring means for sets of size two and three.)

2 INDEPENDENCE IN A HYPERGRAPH. (25 PTS.)

Let (V, \mathcal{F}) be a set system, with $n = |V|$, and \mathcal{F} is a family (i.e., set) of m subsets of V . Here, every set $F \in \mathcal{F}$ is of size exactly three. A set of $S \subseteq V$ is **independent**, if for all $F \in \mathcal{F}$, not all the elements of F are contained in S (i.e., at most two elements of F are in S).

- 2.A.** (5 PTS.) Let $S \subseteq V$ be a random sample generated by picking every element of V into the sample, independently, with probability $1/t$, for some parameter t . A set $F \in \mathcal{F}$ is **bad** if all its elements are in S (i.e., $F \subseteq S$). What is the probability of a specific set $F \in \mathcal{F}$ to be bad?
- 2.B.** (5 PTS.) Let X be the random variable that is the number of bad sets in \mathcal{F} in relation to the random sample S . What is $\mu = \mathbb{E}[X]$?
- 2.C.** (5 PTS.) Prove that $\mathbb{P}[X \geq 2\mu] \leq 1/2$.
- 2.D.** (10 PTS.) Consider the algorithm that now fixes S to be an independent set as follows: scan all the bad sets, and for each such bad set $F \in \mathcal{F}$, randomly throw away one element from S such that F is no longer contained in S .

Verify that the resulting set S' is an independent set. Provide a lower bound, as good as possible, as a function of t , on the expected size of S' . What is the choice of t (as a function of n and m) for which the algorithm (in expectation) outputs the largest possible independent set? What is the expected size of the independent set in this case? (Bigger is better.)

3 FIRE STATIONS. (25 PTS.)

Let $C = \{c_1, \dots, c_n\}$ be the set of locations of n small towns living on the real axis (it is a straight road in the middle of nowhere, and these are the locations of the tiny towns starting from one of its endpoints). Being in America, we would like to build k fire stations to serve their gas needs. Specifically, for a set of locations $Y = \{y_1, \dots, y_k\}$ of the gas stations, the cost of this solution to the i th customer is the squared distance of c_i to its nearest neighbor in Y . Formally, it is $\text{price}(c_i, Y) = |c_i - \text{nn}(c_i, Y)|^2$, where $\text{nn}(c_i, Y)$ is the location of the nearest point to c_i in Y .

(This might seem strange, but the further the fire station is, the more damage caused by the fire before help shows up. This pricing model just try to capture this intuition.)

The **price** of the solution Y is $\text{price}(C, Y) = \sum_i \text{price}(c_i, Y)$.

Given C and k , provide a polynomial time algorithm (in n and k) that computes the price of the cheapest possible solution (i.e., the price of the optimal solution). What is the running time of your algorithm? [You can assume in your solution that $Y \subseteq C$.]

4 GREEDY HITTING SET. (25 PTS.)

Consider the following problem:

Hitting Set

Instance: (S, \mathcal{F}) :

S - a set of n elements

\mathcal{F} - a family of m subsets of S .

Question: Compute a set $S' \subseteq S$ such that S' contains as few elements as possible, and S' “hits” all the sets in \mathcal{F} . Formally, for all $F \in \mathcal{F}$, we have $|S' \cap F| \geq 1$.

The greedy algorithm **GreedyHit** computes a solution by repeatedly computing the element in S , that is contained in the largest number of sets of \mathcal{F} that are not hit yet, adding it to the current solution, and repeating this till all the sets of \mathcal{F} are hit. Let k be the number of elements in the optimal cover. Prove the following:

- 4.A.** (5 PTS.) In the beginning of the i th iteration, if there are β_i sets in \mathcal{F} not hit yet, then there is an element in S that hits at least β_i/k of these sets.
- 4.B.** (10 PTS.) Prove that for any i , we have that $\beta_{i+k} \leq \beta_i/c$, where $c > 1$ is some positive constant (what is the value of c - provide a reasonable lower bound).
- 4.C.** (8 PTS.) Using the above, provide an upper bound (as small as possible) on the number of iterations performed by **GreedyHit** before it stops.
- 4.D.** (2 PTS.) What is the quality of approximation provided by **GreedyHit**?

62.3. Final

1 STRONG DUALITY. (20 PTS.)

Consider a directed graph G with source vertex s and target vertex t and associated costs $\text{cost}(\cdot) \geq 0$ on the edges. Let \mathcal{P} denote the set of all the directed (simple) paths from s to t in G .

Consider the following (very large) integer program:

$$\begin{aligned} & \text{minimize} && \sum_{e \in E(G)} \text{cost}(e)x_e \\ & \text{subject to} && x_e \in \{0, 1\} \quad \forall e \in E(G) \\ & && \sum_{e \in \pi} x_e \geq 1 \quad \forall \pi \in \mathcal{P}. \end{aligned}$$

- 1.A. (5 PTS.) What does this IP compute?
1.B. (5 PTS.) Write down the relaxation of this IP into a linear program.
1.C. (5 PTS.) Write down the dual of the LP from (B). What is the interpretation of this new LP? What is it computing for the graph G (prove your answer)?
1.D. (5 PTS.) The strong duality theorem states the following.

Theorem 62.3.1. *If the primal LP problem has an optimal solution $x^* = (x_1^*, \dots, x_n^*)$ then the dual also has an optimal solution, $y^* = (y_1^*, \dots, y_m^*)$, such that*

$$\sum_j c_j x_j^* = \sum_i b_i y_i^*.$$

In the context of (A)–(C) what result is implied by this theorem if we apply it to the primal LP and its dual above? (For this, you can assume that the optimal solution to the LP of (B) is integral.)

2 SEQUENCES AND CONSEQUENCES. (20 PTS.)

Let $\mathcal{X} = \langle X_1, X_2, \dots, X_n \rangle$ be a sequence of n numbers generated by picking each X_i independently and uniformly from the range $\{1, \dots, n\}$.

- 2.A. (5 PTS.) What is the entropy of \mathcal{X} ?
2.B. (5 PTS.) Consider the sequence $\mathcal{Y} = \langle Y_1, \dots, Y_n \rangle$ that results from sorting the sequence \mathcal{X} in increasing order. For example, if $\mathcal{X} = \langle 4, 1, 4, 1 \rangle$ then $\mathcal{Y} = \langle 1, 1, 4, 4 \rangle$.
Describe an encoding scheme that takes the sequence \mathcal{Y} and encodes it as a sequence of $2n$ binary bits (you will lose points if your scheme uses more bits). Given this encoded sequence of bits, how do you recover the sequence \mathcal{Y} ? (Hint: Consider the differences sequence $Y_1, Y_2 - Y_1, Y_3 - Y_2, \dots, Y_n - Y_{n-1}$. And do **not** use Huffman's encoding.)
Demonstrate how your encoding scheme works for the sequence $\mathcal{Y} = \langle 1, 1, 4, 6, 6, 6 \rangle$.
2.C. (5 PTS.) Consider the set U of all sequences \mathcal{Y} that can be generated by the above process (i.e., it is the set of all monotonically non-decreasing sequences of length n using integer numbers in the range 1 to n). Provide (and prove) an upper bound on the number of elements in U . Your bound should be as small as possible. (Hint: Use (B).)
(Note, that we are not asking for the exact bound on the size of U , which is doable but harder.)
2.D. (5 PTS.) **Prove** an upper bound (as low as possible) on the entropy of \mathcal{Y} . (Proving a lower bound here seems quite hard and you do not have to do it.)

3 FIND k TH SMALLEST NUMBER. (20 PTS.)

This question asks you to design and analyze a *randomized incremental* algorithm to select the k th smallest element from a given set of n elements (from a universe with a linear order).

We assume the numbers are given to you one at a time, and your algorithm has only $O(k)$ space in its disposal that it can use (in particular, the algorithm can not just read all the input and only then compute the desired quantity). Specifically, in an *incremental* algorithm, the input consists of a sequence of elements x_1, x_2, \dots, x_n . After any prefix x_1, \dots, x_{i-1} has been read, the algorithm has computed the k th smallest element in x_1, \dots, x_{i-1} (which is undefined if $i \leq k$), or if appropriate, some other invariant from which the k th smallest element could be determined. This invariant is updated as the next element x_i is read.

We assume that before it is given to us, the input sequence has been randomly permuted, with each permutation equally likely. Note that this case is of interest in analyzing real world situations, where the input arrives as a stream, and we believe that this stream behaves like a random stream of numbers.

- 3.A.** (5 PTS.) Describe an efficient incremental algorithm for computing the k th smallest element (the more efficient it is, the better).
- 3.B.** (5 PTS.) How many comparisons does your algorithm perform in the worst case?
- 3.C.** (10 PTS.) Consider the problem of computing the k smallest numbers in a given stream. Describe an algorithm that outputs these k numbers in sorted order, assuming that k and n are provided in advance, the algorithm has $O(k)$ space, and the input is provided in a stream that is randomly permuted.

What is the expected number (over all permutations) of comparisons performed by your algorithm? For full credit, the expected number of comparisons performed by your algorithm should be as small as possible. **Prove** your answer.

4 STAB THESE RECTANGLES (IN THE BACK, IF POSSIBLE). (20 PTS.)

You are given a set $\mathcal{R} = \{R_1, \dots, R_n\}$ of n rectangles in the plane, and a set $P = \{p_1, \dots, p_n\}$ of n points in the plane. For every point $p \in P$, there is an associated weight $w_p > 0$. Your purpose in this problem is to select a minimum weight subset $X \subseteq P$, such that for any rectangle R of \mathcal{R} there is at least one point of X that is contained in R .

Under the assumption that no rectangle of \mathcal{R} contains more than k points, describe a polynomial time approximation algorithm for this problem. What is the approximation quality of your algorithm? (Naturally, your approximation algorithm should have the best possible approximation quality.) Prove your stated bound on the quality of approximation.

5 SORTING IN $\Omega(n \log n)$ TIME. (20 PTS.)

Prove that any sorting algorithm in the comparison model for sorting n numbers takes $\Omega(n \log n)$ time.

This question would be graded strictly – there is no partial credit for this question.

Bibliography

- [ACG⁺99] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and approximation*. Springer-Verlag, Berlin, 1999.
- [AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proc. 15th Annu. ACM Sympos. Theory Comput.* (STOC), pages 1–9, 1983.
- [AN04] N. Alon and A. Naor. Approximating the cut-norm via grothendieck’s inequality. In *Proc. 36th Annu. ACM Sympos. Theory Comput.* (STOC), pages 72–80, 2004.
- [ASS96] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs*. MIT Press, 1996.
- [BV04] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge, 2004.
- [Chr76] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.
- [CKX10] J. Chen, I. A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theor. Comput. Sci.*, 411(40-42):3736–3756, 2010.
- [CS00] S. Cho and S. Sahni. A new weight balanced binary search tree. *Int. J. Found. Comput. Sci.*, 11(3):485–513, 2000.
- [DLHT13] György Dósa, Rongheng Li, Xin Han, and Zsolt Tuza. Tight absolute bound for first fit decreasing bin-packing: $FFD(l) \leq 11/9OPT(L) + 6/9$. *Theo. Comp. Sci.*, 510:13–61, 2013.
- [EHS14] D. Eppstein, S. Har-Peled, and A. Sidiropoulos. On the greedy permutation and counting distances. manuscript, 2014.
- [ERH12] A. Ene, B. Raichel, and S. Har-Peled. Fast clustering with lower bounds: No customer too far, no shop too small. In submission. <http://sarielhp.org/papers/12/lbc/>, 2012.
- [ES95] J. Erickson and R. Seidel. Better lower bounds on detecting affine and spherical degeneracies. *Discrete Comput. Geom.*, 13:41–57, 1995.
- [FG88] T. Feder and D. H. Greene. Optimal algorithms for approximate clustering. In *Proc. 20th Annu. ACM Sympos. Theory Comput.* (STOC), pages 434–444, 1988.
- [FG95] U. Feige and M. Goemans. Approximating the value of two power proof systems, with applications to max 2sat and max dicut. In *ISTCS '95: Proceedings of the 3rd Israel Symposium on the Theory of Computing Systems (ISTCS'95)*, page 182, Washington, DC, USA, 1995. IEEE Computer Society.
- [GJ90] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.
- [GLS93] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics*. Springer-Verlag, Berlin Heidelberg, 2nd edition, 1993.
- [GO95] A. Gajentaan and M. H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Comput. Geom. Theory Appl.*, 5:165–185, 1995.

- [Gre69] W.R. Greg. *Why are Women Redundant?* Trübner, 1869.
- [GRSS95] M. Golin, R. Raman, C. Schwarz, and M. Smid. Simple randomized algorithms for closest pair problems. *Nordic J. Comput.*, 2:3–27, 1995.
- [Grü03] B. Grünbaum. *Convex Polytopes*. Springer, 2nd edition, may 2003. Prepared by V. Kaibel, V. Klee, and G. Ziegler.
- [GT89] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. Assoc. Comput. Mach.*, 36(4):873–886, 1989.
- [GW95] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. Assoc. Comput. Mach.*, 42(6):1115–1145, nov 1995.
- [Har04] S. Har-Peled. Clustering motion. *Discrete Comput. Geom.*, 31(4):545–565, 2004.
- [Hås01a] J. Håstad. Some optimal inapproximability results. *J. Assoc. Comput. Mach.*, 48(4):798–859, jul 2001.
- [Hås01b] J. Håstad. Some optimal inapproximability results. *J. Assoc. Comput. Mach.*, 48(4):798–859, 2001.
- [HK73] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2:225—231, 1973.
- [HR13] S. Har-Peled and B. Raichel. Net and prune: A linear time algorithm for Euclidean distance problems. In *Proc. 45th Annu. ACM Sympos. Theory Comput. (STOC)*, pages 605–614, New York, NY, USA, 2013. ACM.
- [Kar78] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Math.*, 23:309–311, 1978.
- [KKMO04] S. Khot, G. Kindler, E. Mossel, and R. O’Donnell. Optimal inapproximability results for max cut and other 2-variable csps. In *Proc. 45th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 146–154, 2004. To appear in SICOMP.
- [Meg84] N. Megiddo. Linear programming in linear time when the dimension is fixed. *J. Assoc. Comput. Mach.*, 31:114–127, 1984.
- [MOO05] E. Mossel, R. O’Donnell, and K. Oleszkiewicz. Noise stability of functions with low influences invariance and optimality. In *Proc. 46th Annu. IEEE Sympos. Found. Comput. Sci. (FOCS)*, pages 21–30, 2005.
- [MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, UK, 1995.
- [MS09] M. Mendel and C. Schwob. Fast c-k-r partitions of sparse graphs. *Chicago J. Theor. Comput. Sci.*, 2009, 2009.
- [MU05] M. Mitzenmacher and U. Upfal. *Probability and Computing – randomized algorithms and probabilistic analysis*. Cambridge, 2005.
- [Rab76] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, Orlando, FL, USA, 1976.
- [SA96] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

- [Sch04] A. Schrijver. *Combinatorial Optimization : Polyhedra and Efficiency (Algorithms and Combinatorics)*. Springer, July 2004.
- [Sei93] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, volume 10 of *Algorithms and Combinatorics*, pages 37–68. Springer-Verlag, 1993.
- [Smi00] M. Smid. Closest-point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 877–935. Elsevier, Amsterdam, The Netherlands, 2000.
- [Ste12] E. Steinlight. Why novels are redundant: Sensation fiction and the overpopulation of literature. *ELH*, 79(2):501–535, 2012.
- [Tar85] É. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 1985.
- [WG75] H. W. Watson and F. Galton. On the probability of the extinction of families. *J. Anthropol. Inst. Great Britain*, 4:138–144, 1875.

Index

- (k, n) decoding function, 196
- (k, n) encoding function, 196
- 0/1-flow, 104
- 2-universal, 79
- k -climb, 390
- k -climb packing, 390
- k -packing, 436
- „, 418, 441
- FPTAS, 59
- PTAS, 59
- 3CNF, 17, 18

- acceptable, 131
- active, 227
- admissible, 434
- algorithm
 - Alg, 43, 51, 67
 - alg, 440
 - algExtManyPaths, 205
 - algFPVertexCover, 44, 45
 - algMatching_{HK}, 208
 - algSlowMatch, 203, 208
 - ApproxSubsetSum, 60–62
 - ApproxVertexCover, 44
 - AprxKCenter, 57, 58
 - Alg, 51
 - BarleyMow, 297, 298
 - Bellman-Ford, 210, 211
 - BFS, 100, 203, 205, 207, 211, 212, 214
 - biteEachOther, 388
 - BitonicSorter, 167, 168
 - compare, 215, 217
 - CompBestTreeI, 37
 - CompBestTreeI, 37
 - CompBestTree, 37
 - CompBestTreeI, 37
 - CompBestTreeMemoize, 37
 - Contract, 91, 92
 - decrease-key, 225
 - delete-min, 225
 - DFS, 194, 206–208, 402

 - ed, 34
 - edM, 34
 - Edmonds-Karp, 100, 101
 - edDP, 34
 - edM, 34
 - EnumBinomCoeffAlg, 194
 - ExactSubsetSum, 60
 - Ext, 192, 195
 - FastCut, 91–93
 - FastEuclid, 243
 - FastExp, 33
 - FastExp, 33
 - Feasible, 140, 141
 - FFTAlg, 159, 160, 162
 - FibDP, 32
 - FibI, 32
 - FibR, 32
 - find, 170–175
 - FibI, 32
 - FlipCleaner, 168
 - FlipCoins , 67
 - FlipCoins, 67
 - Ford-Fulkerson, 125
 - fpVertexCoverInner, 45
 - GreedyHit, 425, 449
 - GreedySetCover, 52–54
 - GreedyVertexCover, 42, 43
 - GreedySetCover, 53
 - Half-Cleaner, 167
 - IsMatch, 39
 - Lookup, 78
 - lookup, 78
 - LPStartSolution, 140, 141
 - makeSet, 170–172
 - MatchNutsAndBolts, 66, 67, 71
 - maxFlow-By-Scaling, 416, 417
 - Merger, 168, 169
 - MinCut, 90, 91, 93
 - MinCutRep, 91, 93
 - mtdFordFulkerson, 98–100, 404

- Partition, 294
- Partitions, 30
- PartitionsI, 30
- PartitionsI, 30
- PartitionsI_C, 30
- PartitionS_C, 30
- perceptron, 179–182
- play, 429
- PlayItBen, 411
- QuickSelect, 68, 69, 395
- QuickSort, 66–68, 70–72, 75, 220, 221, 228
- RadixSort, 393, 437
- RandBit, 67
- Random, 294
- randomized, 51
- Relax, 211
- RotateLeft, 76
- RotateRight, 76
- RotateUp, 76
- shrink, 429
- Simplex, 136, 137, 140–143, 150–152
- SimplexInner, 141
- SlowEuclid, 243
- SolveSubsetSum, 58
- Sorter, 169
- Trim, 60, 61
- union, 170–172
- WeirdEuclid, 243
- WGreedySetCover, 54, 55
- alternating BFS, 211
- alternating cycle, 202
- alternating path, 202
- amortize, 80
- amplification, 91
- approximation
 - algorithm
 - minimization, 43
 - maximization problem, 51
- Arithmetic coding, 285, 422
- augmenting, 202
- augmenting path, 97
- average optimal cost, 54
- average price, 54
- backwards analysis, 219
- bad, 81, 424, 448
- balance, 424, 448
- balanced coloring, 424, 448
- basic solution, 143
- basis, 131
- Bin Packing
 - next fit, 62
- bin packing
 - first fit, 62
- binary code, 185
- binary entropy, 190
- binary symmetric channel, 195
- bitonic sequence, 166
- black-box access, 225
- Bland’s rule, 142
- blossom, 213
- breakable, 414
- capacity, 94, 98
- cardinality, 443
- cell, 221
- chaining, 78
- Chernoff inequality, 73
- Cholesky decomposition, 178
- circulation, 106
- classification, 179
- classifying, 179
- clause gadget, 23
- clique, 19
- clustering
 - k -center, 57
 - price, 56
- CNF, 17, 18, 409, 433, 446
- collapsible, 158
- collide, 78
- coloring, 22
- comparison network, 165
 - depth, 165
 - depth wire, 165
 - half-cleaner, 167
 - sorting network, 165
- Compute, 81
- conditional entropy, 286, 423
- conditional probability, 88
- cone, 131
- configurations, 35
- congestion, 155
- congruent modulo p , 80
- contraction
 - edge, 88
- convex, 37
- convex hull, 128
- convex polygon, 37
- convex programming, 175
- convolution, 162

- cost, 119
- critical, 223
- cuckoo hashing, 78
- cut, 88, 98
 - minimum, 88
- cuts, 88
- Cycle
 - directed, 116
 - Eulerian, 48
- DAG, 35, 165, 207, 386, 387, 434
- decision problem, 15
- diagonal, 37, 434
- digraph, 116
- directed, 94
- directed acyclic graph, 35
- directed cut, 98
- directed Cycle
 - average cost, 116
- disjoint paths
 - edge, 104
- distance
 - point from set, 56
- divides, 80
- dominates, 410, 447
- dot product, 179
- double hitting set, 412
- Dynamic, 78
- dynamic programming, 31
- edge, 131
- edge cover, 147
- edge disjoint, 439
 - paths, 104
- edit distance, 33
- entering, 141
- entropy, 189, 190
 - binary, 190
- Eulerian, 48
- Eulerian cycle, 24, 48
- expectation, 64
 - conditional, 65
 - linearity, 64
- facility location problem, 56
- family, 79
- Fast Fourier Transform, 158
- feasible, 112
 - LP
 - solution, 129
- feasible permutations, 216
- finite metric, 225
- fixed parameter tractable, 46, 256
- Flow
 - capacity, 94
 - Integrality theorem, 100
 - value, 95
- flow, 95
 - 0/1-flow, 104
 - circulation
 - minimum-cost, 119
 - valid, 119
 - cost, 118
 - efficient, 125
 - min cost
 - reduced cost, 122
 - minimum-cost, 118
 - residual graph, 96
 - residual network, 96
- flow network, 94
- flower, 213
- flow
 - flow across a cut, 98
- Fold, 81
- forces, 386
- Ford-Fulkerson method, 98
- FPTAS, 59
- fully polynomial time approximation scheme, 59
- gadget, 22
 - color generating, 22
- greedy algorithms, 42
- grid, 221, 413
- ground set, 52, 412, 432
- Hamiltonian cycle, 24
- Helly's theorem, 129
- high probability, 67
- Huffman coding, 187
- hypergraph, 52
- image segmentation problem, 110
- imaginary, 163
- independent, 424, 434, 448
- independent set, 21, 147
- induced, 428
- induced subgraph, 19, 44
- induction fairy, 399
- inverse, 80
- isolator, 413
- kernel technique, 183

- labeled, 180
- LCA, 213
- leader, 172
- learning, 179
- leaving, 142
- leftover graph, 439
- length, 202
- lexicographically, 150
- linear classifier h , 180
- linear program
 - dual, 144
- Linear programming
 - standard form, 136
- linear programming, 129, 129, 131–134
 - constraint, 134
 - dual, 143
 - dual program, 145
 - entering variable, 141
 - instance, 129–134
 - pivoting, 139
 - primal, 143
 - primal problem, 144
 - primal program, 145
 - slack form, 137
 - slack variable, 136
 - target function, 129
 - unbounded, 129
 - variable
 - basic, 137
 - non-basic, 137
 - vertex, 130
- Linearity of expectations, 51
- linearization, 183
- load factor, 78
- longest ascending subsequence, 39
- LP, 129, 130, 133–156
- LPI, 129
- lucky, 227

- Macaroni sort, 164
- Markov's Inequality, 72
- matched, 202
- matching, 46, 103, 201
 - bridge, 211
 - free edge, 202
 - free vertex, 202
 - matching edge, 202
 - maximum matching, 103
 - perfect, 46, 103
- maximal, 46, 201
 - maximization problem, 50
 - maximum cardinality matching, 201
 - maximum cut problem, 176
 - maximum flow, 95
 - maximum matching, 46
 - maximum size matching, 201
 - maximum weight matching, 201
 - maximum-weight matching, 46
 - memoization, 30
 - merge sort, 169
 - meta graph, 393, 437
 - meta-spider, 431
 - metric, 225
 - metric space, 225
 - min-weight perfect matching, 46
 - mincut, 88
 - minimum average cost cycle, 116
 - minimum cut, 98
 - mirror, 386
 - multiplicity, 442

 - neighborhood, 221
 - net, 225
 - network, 94

 - objective function, 134

 - packing, 147, 435
 - argument, 221
 - partition number, 29
 - path
 - augmenting, 209
 - perfect, 201, 440
 - pivoting, 142
 - point-value pairs, 157
 - polar form, 163
 - polyhedron, 129
 - polynomial, 157
 - polynomial reductions, 16
 - polynomial time approximation scheme, 59
 - polytope, 129
 - popular, 403
 - positive semidefinite, 178
 - potential, 118
 - prefix code, 185
 - prefix-free, 185
 - price, 403, 424, 448
 - prime, 81
 - probabilistic method, 150, 200
 - probability, 154
 - conditional, 64

Problem

- k*-CENTER, 253, 327, 346, 355
- 2SAT
 - 2SAT Max, 55
- 3Colorable, 22, 398
- 3DM, 27, 398
- 3DM, 431
- 3EdgeColorable, 431
- 3SAT
 - 2SAT Max, 55
 - 3SAT Max, 50, 55
- Approx Partition, 352
- Balanced coloring of a set system., 424, 448
- BIN PACKING, 247, 250, 252, 325, 342, 344, 353
- bin packing
 - min, 62
- Circuit Satisfiability, 14
- Clique, 20
- clustering
 - k*-center, 57
- Cover by paths (edge disjoint)., 413, 432
- Cover by paths (vertex disjoint)., 413
- CYCLE HATER., 399, 431
- CYCLE LOVER., 399
- DOUBLE HITTING SET, 412
- Hamiltonian Cycle, 25
- Hamiltonian Path, 125, 249, 341
- HITTING SET, 247, 249, 250, 253, 326, 341, 342, 345, 354
- Hitting Set, 425, 449
- Independent Set, 21, 397
- Integer Linear Programming, 280, 418
- KNAPSACK, 251, 343
- Knapsack, 377
- LARGEST COMMON SUBGRAPH, 247, 343
- Largest Subset, 375
- Many Meta-Spiders., 431
- MAX 2SAT, 410, 447
- Max 3SAT, 433
- MAX BIN PACKING, 412
- MAX Cut, 409, 446
- Max Degree Spanning Tree, 249, 341
- Max Inner Spanning Tree, 432
- MAX SAT, 253, 327, 346, 355
- MaxClique, 19
- Min Edge Coloring, 377
- Min Leaf Spanning Tree, 412
- MINIMUM SET COVER, 247, 249, 250, 252, 326, 341, 342, 345, 353
- Minimum Test Collection, 246
- MULTIPROCESSOR SCHEDULING, 251, 343
- NO COVER, 432
- NOT SET COVER, 412
- Not-3SAT, 249, 340
- PARTITION, 250, 342
- Partition, 28, 398
- Partition graph into not so bad, and maybe even good, sets (PGINSBMEGS)., 413
- SAT Partial, 433
- Semi-Independent Set, 430
- SET COVER, 28, 398, 431
- Set Cover, 52, 154
- Shortest Path, 397, 430
- Subgraph embedding, 376
- SUBGRAPH ISOMORPHISM, 247, 343
- Subgraph Isomorphism, 246, 399
- SUBSET SUM, 250, 341
- Subset Sum, 26, 58, 248, 398, 431
- subset sum
 - approximation, 59
 - optimization, 59
- TILING, 247, 251, 252, 326, 342, 344, 353
- TRIPLE HITTING SET, 432
- TSP, 26, 398
 - Min, 47
 - With the triangle inequality, 47, 48
- TSP Path, 41
- TSP: Traveling Salesperson Problem, 40
- Vec Subset Sum, 26
- VERTEX COVER, 251, 325, 343, 352, 381
- Vertex Cover, 21, 398
 - Min, 42–46
 - Minimization, 42
- Weight Set Cover, 54
- problem
 - 2CNF-SAT, 374
 - 2SAT, 55, 248, 408, 445
 - 3CNF, 26
 - 3COLORABLE, 254
 - 3COLORSillyGraph, 435
 - 3Colorable, 22, 24
 - 3DM, 27
 - 3SAT, 17, 18, 20, 22, 26, 27, 50, 51, 248, 378
 - 3SUM, 219
 - A, 20
 - AFWLB, 125
 - Bin Packing, 62
 - Circuit Satisfiability, 15–17
 - Circuit-SAT, 409, 446
 - Clique, 21

co-NP, 378
 CSAT, 17, 18
 EDGE COVER, 254
 EXACT-COVER-BY-3-SETS, 248
 formula satisfiability, 16, 17
 Halting, 378
 Hamiltonian Cycle, 25, 47
 Hamiltonian Path, 125, 432
 Hamiltonian path, 125
 HITTING SET, 432
 Independent Set, 21, 22, 255
 independent set, 442
 Knapsack, 376
 Largest Subset, 375
 LONGEST PATH, 254
 ManyCover, 442
 MAX 2SAT, 410, 447
 Max 3SAT, 50, 51
 Max 5SAT, 377
 MAX CUT, 176, 254
 MAX Cut, 410, 447
 Max CUT, 409, 446
 MAX SAT, 253
 Max SAT, 433
 MAX SPANNING TREE, 254
 MaxClique, 19, 20
 MIN CUT, 254
 Min Edge Coloring, 377, 378
 MIN EQUIVALENT DIGRAPH, 254
 MIN SPANNING TREE, 254
 minimization, 42
 Not-3SAT, 249, 341
 NP, 374, 378
 NPC, 26
 Partition, 28, 62, 63, 246, 340, 399
 PRIMALITY, 255
 PROB, 59
 reduction, 246
 SAT, 16–18, 433
 SAT Partial, 433
 Set Cover, 43, 52, 54, 147, 154
 SetCover, 54
 SHORTEST PATH, 254
 Sorting Nuts and Bolts, 65
 Subgraph embedding, 376
 Subgraph Isomorphism, 246, 399
 Subset Sum, 26–28, 58–60, 248
 TRANSITIVE REDUCTION, 254
 TRAVELING SALESMAN PROBLEM, 254
 TSP, 26, 47, 48
 TSP Path, 41
 TSP REVISIT, 376
 Uniqueness, 218
 uniqueness, 224
 VACATION TOUR PROBLEM, 254
 Vec Subset Sum, 26–28
 VERTEX COVER, 254
 Vertex Cover, 22, 25, 42, 152, 255, 273, 349
 VertexCover, 273, 349
 VertexCover, 43
 Weighted Vertex Cover, 152, 153
 X, 20
 profit, 112, 434
 PTAS, 59
 quality, 410, 447
 Quick Sort
 lucky, 70
 quotation
 – A Scanner Darkly, Philip K. Dick, 195
 – Defeat, Arkady and Boris Strugatsky, 193
 – The first world war, John Keegan., 29
 – The roots of heaven, Romain Gary, 36
 – Romain Gary, The talent scout., 190
 A confederacy of Dunces, John Kennedy Toole, 13
 quotient, 80
 random variable, 64
 random variables
 independent, 64
 rank, 66, 68, 434
 real, 163
 rearrangeable, 260
 reduced cost, 122
 regular, 440
 relaxation, 153
 remainder, 80, 268, 420, 444
 residual capacity, 96, 97
 residual graph, 119
 robust spanning tree, 406
 rounding, 153
 running-time
 amortized, 170
 expected, 66
 see, 52
 separator, 404
 set system, 52
 shortcutting, 48
 sidelength, 221
 significant, 386

- simplex, 131
- sink, 94, 95
- size, 78, 88, 443
- slack form, 136
- snake, 434
- sorting network
 - bitonic sorter, 167
 - running time, 165
 - size, 165
 - zero-one principle, 165
- source, 94, 95
- Static, 78
- stem, 213
- streaming, 233
- strong duality theorem, 145
- subset sums, 443
- sum, 443
- Swapping, 260

- target function, 129
- theorem
 - Helly's, 129
- training, 179
- transportation problem, 127
- treap, 75
- tree
 - code trees, 185
 - prefix tree, 185
- triangle inequality, 48
- triple hitting set, 432
- TSP, 40, 41, 436

- unimodal, 403
- union bound, 70
- union-find
 - block of a node, 174
 - jump, 174
 - internal, 174
 - path compression, 171
 - rank, 171
 - union by rank, 171
- Unique Games Conjecture, 46
- unsupervised learning, 56
- upper envelope, 228
- usable, 438
- useful, 438
- useless, 438
- uselessly-connected, 386

- value, 95, 407
- Vandermonde, 160

- variable gadget, 23
- vertex, 229
- vertex cover, 21
- visibility polygon, 52

- weak circulation, 119
- weight, 46
 - cycle, 209
- weighted point, 226
- width, 221