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

11

I NP Completeness

11

1 NP Completeness I

1.1 Introduction

11

1.2 Complexity classes

13

1.2.1 Reductions

14

1.3 More NP-COMPLETE problems

15

1.3.1 3SAT

15

1.4 Bibliographical Notes

17

2 NP Completeness II

17

2.1 Max-Clique

17

2.2 Independent Set

19

2.3 Vertex Cover

19

2.4 Graph Coloring

20

3 NP Completeness III

22

3.1 Hamiltonian Cycle

22

3.2 Traveling Salesman Problem

23

3.3 Subset Sum

24

3.4 3 dimensional Matching (3DM)

26

3.5 Partition

26

3.6 Some other problems

26

4 Exercises - NP Completeness

27

4.1 Equivalence of optimization and decision problems

27

4.2 Showing problems are NP-COMPLETE

28

4.3 Solving special subcases of NP-COMPLETE problems in polynomial time

29

II Dynamic programming

38

5 Dynamic programming

38

5.1 Basic Idea - Partition Number

38

5.1.1 A Short sermon on memoization

40

5.2 Example – Fibonacci numbers

40

5.2.1 Why, where, and when?

40
5.2.2 Computing Fibonacci numbers .............................................. 41
5.3 Edit Distance ................................................................. 42
  5.3.1 Shortest path in a DAG and dynamic programming ................ 44
6 Dynamic programming II - The Recursion Strikes Back ............... 44
  6.1 Optimal search trees ....................................................... 45
  6.2 Optimal Triangulations .................................................... 46
  6.3 Matrix Multiplication ...................................................... 47
  6.4 Longest Ascending Subsequence ....................................... 48
  6.5 Pattern Matching .......................................................... 48

III Approximation algorithms ............................................... 49

7 Approximation algorithms .................................................. 49
  7.1 Greedy algorithms and approximation algorithms .................. 49
    7.1.1 Alternative algorithm – two for the price of one ............... 51
  7.2 Fixed parameter tractability, approximation, and fast exponential time algorithms (to say nothing of the dog) ............... 51
    7.2.1 A silly brute force algorithm for vertex cover ................ 51
    7.2.2 A fixed parameter tractable algorithm ......................... 51
      7.2.2.1 Remarks ....................................................... 53
  7.3 Traveling Salesman Person .............................................. 53
    7.3.1 TSP with the triangle inequality ............................... 53
      7.3.1.1 A 2-approximation ....................................... 53
      7.3.1.2 A 3/2-approximation to TSP_{△-Min} .................... 54
  7.4 Biographical Notes ..................................................... 56

8 Approximation algorithms II .............................................. 56
  8.1 Max Exact 3SAT .......................................................... 56
  8.2 Approximation Algorithms for Set Cover ........................... 58
    8.2.1 Guarding an Art Gallery ....................................... 58
    8.2.2 Set Cover ........................................................ 58
    8.2.3 Lower bound ...................................................... 59
    8.2.4 Just for fun – weighted set cover ............................. 60
      8.2.4.1 Analysis .................................................... 60
  8.3 Biographical Notes ..................................................... 61

9 Approximation algorithms III ............................................. 62
  9.1 Clustering ............................................................... 62
    9.1.1 The approximation algorithm for k-center clustering ........ 63
  9.2 Subset Sum .............................................................. 64
    9.2.1 On the complexity of ε-approximation algorithms ........... 65
    9.2.2 Approximating subset-sum ..................................... 65
      9.2.2.1 Bounding the running time of ApproxSubsetSum ........ 67
      9.2.2.2 The result .................................................. 67
  9.3 Approximate Bin Packing .............................................. 68
  9.4 Bibliographical notes .................................................. 68

10 Exercises - Approximation Algorithms ................................ 68
  10.1 Greedy algorithms as approximation algorithms .................. 68
IV Randomized algorithms

11 Randomized Algorithms

11.1 Some Probability

11.2 Sorting Nuts and Bolts

11.2.1 Running time analysis

11.2.1.1 Alternative incorrect solution

11.2.2 What are randomized algorithms?

11.3 Analyzing QuickSort

11.4 QuickSelect — median selection in linear time

12 Randomized Algorithms II

12.1 QuickSort and Treaps with High Probability

12.1.1 Proving that an element participates in small number of rounds

12.1.2 An alternative proof of the high probability of QuickSort

12.2 Chernoff inequality

12.2.1 Preliminaries

12.2.2 Chernoff inequality

12.2.2.1 The Chernoff Bound — General Case

12.3 Treaps

12.3.1 Construction

12.3.2 Operations

12.3.2.1 Insertion

12.3.2.2 Deletion

12.3.2.3 Split

12.3.2.4 Meld

12.3.3 Summery

12.4 Bibliographical Notes

13 Min Cut

13.1 Min Cut

13.1.1 Problem Definition

13.1.2 Some Definitions

13.2 The Algorithm

13.2.1 The resulting algorithm

13.2.1.1 On the art of randomly picking an edge

13.2.2 Analysis

13.2.2.1 The probability of success

13.2.2.2 Running time analysis

13.3 A faster algorithm

13.3.1 On coloring trees and min-cut

13.4 Bibliographical Notes

14 Randomized Algorithms

14.1 Randomized algorithms
VII Miscellaneous topics

26 Fast Fourier Transform
26.1 Introduction .................................................. 165
26.2 Computing a polynomial quickly on \( n \) values .................... 166
26.2.1 Generating Collapsible Sets ................................. 167
26.3 Recovering the polynomial .................................... 167
26.4 The Convolution Theorem .................................... 169
26.4.1 Complex numbers – a quick reminder ....................... 170

27 Sorting Networks
27.1 Model of Computation ......................................... 171
27.2 Sorting with a circuit – a naive solution ........................ 172
27.2.1 Definitions ..................................................... 172
27.2.2 Sorting network based on insertion sort .................... 173
27.3 The Zero-One Principle ....................................... 173
27.4 A bitonic sorting network ...................................... 174
27.4.1 Merging sequence .......................................... 175
27.5 Sorting Network .................................................. 176
27.6 Faster sorting networks ........................................ 176

28 Union Find
28.1 Union-Find ....................................................... 177
28.1.1 Requirements from the data-structure ...................... 177
28.1.2 Amortized analysis .......................................... 177
28.1.3 The data-structure .......................................... 177
28.2 Analyzing the Union-Find Data-Structure ...................... 179

29 Approximate Max Cut
29.1 Problem Statement ............................................ 182
29.1.1 Analysis ....................................................... 183
29.2 Semi-definite programming ................................... 184
29.3 Bibliographical Notes .......................................... 185

30 The Perceptron Algorithm
30.1 The perceptron algorithm ..................................... 185
30.2 Learning A Circle .............................................. 188
30.3 A Little Bit On VC Dimension ................................ 189
30.3.1 Examples ....................................................... 190

VIII Compression, entropy, and randomness

31 Huffman Coding
31.1 Huffman coding ................................................ 191
31.1.1 The algorithm to build Hoffman’s code ................... 193
31.1.2 Analysis ....................................................... 193
31.1.3 What do we get .................................................. 194
31.1.4 A formula for the average size of a code word .......... 195

32 Entropy, Randomness, and Information 195
32.1 Entropy ................................................................. 195
  32.1.1 Extracting randomness ...................................... 198

33 Even more on Entropy, Randomness, and Information 199
33.1 Extracting randomness .......................................... 199
  33.1.1 Enumerating binary strings with $j$ ones ............... 199
  33.1.2 Extracting randomness ...................................... 200
33.2 Bibliographical Notes ............................................ 201

34 Shannon's theorem 201
34.1 Coding: Shannon’s Theorem ................................... 201
  34.1.0.1 Intuition behind Shanon’s theorem ...................... 202
  34.1.0.2 What is wrong with the above? .......................... 202
34.2 Proof of Shannon’s theorem .................................... 203
  34.2.1 How to encode and decode efficiently ................... 203
    34.2.1.1 The scheme ............................................. 203
    34.2.1.2 The proof .............................................. 203
  34.2.2 Lower bound on the message size ....................... 206
34.3 Bibliographical Notes ............................................ 206

35 Exercises - Entropy 206

IX Miscellaneous topics II 209

36 Matchings 209
36.1 Definitions and basic properties ............................. 209
  36.1.1 Definitions .................................................. 209
  36.1.2 Matchings and alternating paths ......................... 209
36.2 Unweighted matching in bipartite graph .................... 211
  36.2.1 The slow algorithm: algSlowMatch ....................... 211
  36.2.2 The Hopcroft-Karp algorithm ............................. 211
    36.2.2.1 Some more structural observations ................. 211
    36.2.2.2 Improved algorithm .................................. 212
    36.2.2.3 Extracting many augmenting paths: algExtManyPaths 213
  36.2.4 The result .................................................. 215
36.3 Bibliographical notes ............................................ 215

37 Matchings II 216
37.1 Maximum Weight Matchings in A Bipartite Graph .......... 216
  37.1.1 On the structure of the problem ......................... 216
37.2 Maximum Weight Matchings in A Bipartite Graph .......... 217
  37.2.1 Faster Algorithm .......................................... 218
37.3 The Bellman-Ford Algorithm - A Quick Reminder .......... 218
37.4 Maximum Size Matching in a Non-Bipartite Graph .......... 218
  37.4.1 Finding an augmenting path .............................. 218
  37.4.2 The algorithm .............................................. 221
"Then you must begin a reading program immediately so that you may 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 currently [Jan 2006]^9, the

^9But 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.
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.
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): \(x_1 ← 0, x_2 ← 1, ..., x_m ← 1\) 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** for this negative answer, and this proof can be verified in polynomial time.

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 \(\Pi\) is **NP-HARD**, if being able to solve \(\Pi\) 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 NEW 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:

\[(a \lor b \lor c \lor \overline{d}) \iff ((b \land \overline{c}) \lor (\overline{d} \Rightarrow d) \lor (c \neq a \land b))\]

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

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.

\[
\begin{align*}
    y_1 &= x_1 \land x_4 \\
    y_2 &= \overline{x_4} \\
    y_3 &= y_2 \land x_3 \\
    y_4 &= x_2 \lor y_1 \\
    y_5 &= \overline{x_2} \\
    y_6 &= \overline{x_5} \\
    y_7 &= y_3 \lor y_5 \\
    y_8 &= y_4 \land y_7 \land y_6
\end{align*}
\]

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

\[(y_1 = x_1 \land x_4) \land (y_2 = \overline{x}_4) \land (y_3 = y_2 \land x_3) \land (y_4 = x_2 \lor y_1) \land (y_5 = \overline{x}_2) \land (y_6 = \overline{x}_5) \land (y_7 = y_3 \lor y_5) \land (y_8 = y_4 \land y_7 \land y_6) \land y_8.\]

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 \(T_{\text{SAT}}(n)\), we can solve the CSAT problem in time

\[T_{\text{CSAT}}(n) \leq O(n) + T_{\text{SAT}}(O(n)),\]

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, \(T_{\text{CSAT}}(n) \geq 2^n\). Which implies by the above reduction that

\[2^n \leq T_{\text{CSAT}}(n) \leq O(n) + T_{\text{SAT}}(O(n)).\]

Namely, \(T_{\text{SAT}}(n) \geq 2^{n/c} - O(n)\), 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 \(\text{SAT} \in \text{P}\) then \(\text{CSAT} \in \text{P}\).

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:

\[\overline{(a \lor b \lor \overline{c}) \land (a \lor \overline{c}) \land (c \lor e)}.\]

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

The problem 3SAT is formula satisfiability formula with exactly three literals in each clause.

**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) \( a = b \land c \) if it corresponds to an AND gate, (ii) \( a = b \lor c \) if it corresponds to an OR gate, and (iii) \( a = \overline{b} \) 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 \( a = b \land c \) will be translated into

\[
(a \lor \overline{b} \lor \overline{c}) \land (a \lor b) \land (\overline{a} \lor c) \tag{1.1}
\]

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

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

\[
(\overline{a} \lor b \lor c) \land (a \lor \overline{b}) \land (a \lor \overline{c})
\]

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

\[
(a \lor b) \land (\overline{a} \lor \overline{b})
\]

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

\[
(a \lor x \lor y) \land (a \lor \overline{x} \lor y) \land (a \lor x \lor \overline{y}) \land (a \lor \overline{x} \lor \overline{y})
\]

by introducing two new dummy variables \( x \) and \( y \). And a two variable clause \( a \lor b \) would be replaced by

\[
(a \lor b \lor y) \land (a \lor b \lor \overline{y})
\]

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 \( NP\text{-Complete} \).

\[
\begin{align*}
\text{Input: boolean circuit} & \quad \Downarrow O(n) \\
\text{3CNF formula} & \quad \Downarrow \\
\text{Decide if given formula is satisfiable using 3SAT solver} & \quad \Downarrow \\
\text{Return TRUE or FALSE} & 
\end{align*}
\]

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.

![Figure 2.1: A clique of size 4 inside a graph with 8 vertices.](image)

**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.

We will prove that MaxClique is NP-HARD. Before dwelling into that, the simple algorithm we devised for MaxClique shade 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.

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

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

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.

$\Leftarrow$ 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 $\overline{x_i}$.

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 $\overline{x_i}$; 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 $G_U$ 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 in 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 $k$?

Theorem 2.1.3. Clique is NP-COMPLETE.
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 $G$.

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.

<table>
<thead>
<tr>
<th>Independent Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance: A graph $G$, integer $k$</td>
</tr>
<tr>
<td>Question: Is there an independent set in $G$ of size $k$?</td>
</tr>
</tbody>
</table>

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 $S$.

<table>
<thead>
<tr>
<th>Vertex Cover</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance: A graph $G$, integer $k$</td>
</tr>
<tr>
<td>Question: Is there a vertex cover in $G$ of size $k$?</td>
</tr>
</tbody>
</table>

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

19
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 $G$. $\blacksquare$

**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 $k$ iff it has a vertex cover of size $n - k$. Thus, feeding $G$ and $n - k$ into (the supposedly given) black box that can solves 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**. $\blacksquare$

### 2.4. Graph Coloring

**Definition 2.4.1.** A **coloring**, by $c$ colors, of a graph $G = (V, E)$ is a mapping $C : V(G) \rightarrow \{1, 2, \ldots, c\}$ 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\(^3\).

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.

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 $x$ and $\overline{y}$, 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 $\overline{y}$ 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 $y$.

\(^3\)If you do not know the algorithm for this, please read about it to fill this monstrous gap in your knowledge.
Finally, for every clause we introduce a clause gadget. See the figure on the right – for how the gadget looks like for the clause $a \lor b \lor c$. Note, that the vertices marked by $a$, $b$ and $\overline{c}$ 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 $s$.

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 $3$Colorable is NP-Complete.

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 stabs 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 $k$ disjoints 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, if 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 $n + 1$. 

23
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 covert 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.

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 $x$. 

<table>
<thead>
<tr>
<th>Target</th>
<th>??</th>
<th>?</th>
<th>01</th>
<th>??</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>??</td>
<td>?</td>
<td>01</td>
<td>??</td>
</tr>
<tr>
<td>$a_2$</td>
<td>??</td>
<td>?</td>
<td>01</td>
<td>??</td>
</tr>
</tbody>
</table>
Next, consider a clause \( C \equiv a \lor b \lor \overline{c} \) appearing in \( F \). This clause requires that we choose at least one row from the rows corresponding to \( a \), \( b \) to \( c \). 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 clauses 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 \( \overline{c} \).

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 \( \text{subset sum} \). Clearly, this instance of \( \text{subset sum} \) has a solution if and only if the original instance of \( 3\text{SAT} \) had a solution. Since \( \text{subset sum} \) is in \( \text{NP} \) as an be easily verified, we conclude that that \( \text{subset sum} \) is \( \text{NP-Complete} \).

**Theorem 3.3.1.** \( \text{subset sum} \) is \( \text{NP-Complete} \).

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

---

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

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-Complete.**

*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 \( a_1, \ldots, a_n \) 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:

\[ a_1, \ldots, a_n, a_{n+1}, a_{n+2} \]

Clearly, Partition is in NP and thus it is NP-Complete.

3.6. Some other problems

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

**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 \( S_1, \ldots, S_k \in \mathcal{F} \) that cover \( S \). Formally, \( \bigcup_{i} S_i = S \)?
Chapter 4

Exercises - NP Completeness

4.1. Equivalence of optimization and decision problems

4.1.1. 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.

(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 $C'$).

(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.

4.1.2. 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

$$\sum_{s \in T} s = \sum_{s \in S \setminus T} s.$$
4.2. Showing problems are **NP-Complete**

4.2.1. Graph Isomorphisms

(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 $f : V_2 \to V$ satisfying $\{u, v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E$?

(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 exists 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?

4.2.2. NP Completeness collection

(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 $S_1, \ldots, S_k$ in $C$ such that $S \subseteq \cup_{i=1}^k S_i$?

(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 $U$ int disjoint sets $U_1, \ldots, U_K$ such that the sum of the sizes of the items inside each $U_i$ is $B$ or less?

(C) (5 pts.)

**TILING**

**Instance:** Finite set $\text{RECTS}$ of rectangles and a rectangle $R$ in the plane.

**Question:** Is there a way of placing the rectangles of $\text{RECTS}$ inside $R$, so that no pair of the rectangles intersect, and all the rectangles have their edges parallel of the edges of $R$?

(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 $C$.

4.2.3. LONGEST-PATH

Show that the problem of deciding whether an unweighted undirected graph has a path of length greater than $k$ is **NP-Complete**.
4.2.4. 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.

4.3. Solving special subcases of NP-COMPLETE problems in polynomial time

4.3.1. 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 $P = NP$?

4.3.2. 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.

(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 $F$.

(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?

(C) (6 pts.) Describe a polynomial time algorithm that solves the **2SAT** problem (using (b)). How fast is your algorithm?

4.3.3. 2-CNF-SAT

Prove that deciding satisfiability when all clauses have at most 2 literals is in P.

4.3.4. Hamiltonian Cycle Revisited

Let $C_n$ denote the cycle graph over $n$ vertices (i.e., $V(C_n) = \{1, \ldots, n\}$, and $E(C_n) = \{\{1, 2\}, \{2, 3\}, \ldots, \{n-1, n\}, \{n, 1\}\}$).

Let $C^k_n$ denote the graph where $\{i, j\} \in E(C^k_n)$ 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 $C^k_n$, 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 $k$?
4.3.5. Partition revisited
Let $S$ be an instance of partition, such that $n = \left| S \right|$, and $M = \max_{s \in S} s$. Show a polynomial time (in $n$ and $M$) algorithm that solves partition.

4.3.6. 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.)

(A) Prove that Not-3SAT is co-$NP$.
(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?

(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).

4.3.7. $NP$-Completeness Collection
(20 pts.) Prove that the following problems are $NP$-Complete.

**MINIMUM SET COVER**

(A) **Instance**: Collection $C$ of subsets of a finite set $S$ and an integer $k$.

**Question**: Are there $k$ sets $S_1, \ldots, S_k$ in $C$ such that $S \subseteq \bigcup_{i=1}^k S_i$?

**HITTING SET**

(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**

(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**

(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 $k$?

4.3.8. Independence
(10 pts.) Let $G = (V, E)$ be an undirected graph over $n$ vertices. Assume that you are given a numbering $\pi : V \rightarrow \{1, \ldots, n\}$ (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.

4.3.9. 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.

4.3.10. 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 $S_1, \ldots, S_k$ in $C$ such that $S \subseteq \bigcup_{i=1}^{k} S_i$?

(A) (5 pts.) Prove that MINIMUM SET COVER problem is NP-COMPLETE

(B) (5 pts.) Prove that the following problem is NP-COMPLETE.

HITTING SET

**Instance:** A collection $C$ of subsets of a finite 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 $C$.

(C) (5 pts.) Hitting set on the line

Given a set $I$ 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 $I \in I$ there is a point $p \in X$, such that $p \in I$.

4.3.11. 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 $U_1, \ldots, U_K$ such that the sum of the sizes of the items inside each $U_i$ is $B$ or less?

(A) (5 pts.) Show that the BIN PACKING problem is NP-COMPLETE

(B) (5 pts.) Show that the following problem is NP-COMPLETE.

TILING

**Instance:** Finite set $RECTS$ of rectangles and a rectangle $R$ in the plane.

**Question:** Is there a way of placing all the rectangles of $RECTS$ 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 $R$?
4.3.12. Knapsack

(A) (5 pts.) Show that the following problem is NP-COMPLETE.

**KNAPCSACK**

**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 $\sum_{u \in U'} v(u) \geq K$.

(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 $A = A_1 \cup A_2 \cup \cdots \cup A_m$ of $A$ into $m$ disjoint sets such that $\max \{ \sum_{a \in A_i} l(a) : 1 \leq i \leq m \} \leq D$?

(C) Scheduling with profits and deadlines

Suppose you have one machine and a set of $n$ tasks $a_1, a_2, \ldots, a_n$. 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.

(i) (3 pts.) State this problem as a decision problem.

(ii) (2 pts.) Show that the decision problem is NP-COMPLETE.

4.3.13. 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 $V'$?

(A) Show that VERTEX COVER is NP-COMPLETE. Hint: Do a reduction from INDEPENDENT SET to VERTEX COVER.

(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 $G$.

(C) Present a linear time algorithm that solves this problem for the case that $G$ is a tree.

(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.

---

It 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.
4.3.14. 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 $U$ int disjoint sets $U_1, \ldots, U_K$ such that the sum of the sizes of the items inside each $U_i$ is $B$ or less?

(A) Show that the BIN PACKING problem is **NP-Complete**

(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 $U$.

(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.

(D) Show that the following problem is **NP-Complete**.

**TILING**

**Instance:** Finite set $\text{RECTS}$ of rectangles and a rectangle $R$ in the plane.

**Question:** Is there a way of placing the rectangles of $\text{RECTS}$ inside $R$, so that no pair of the rectangles intersect, and all the rectangles have their edges parallel of the edges of $R$?

(E) Assume that $\text{RECTS}$ is a set of squares that can be arranged as to tile $R$ completely. Present a polynomial time algorithm that computes a subset $T \subseteq \text{RECTS}$, and a tiling of $T$, so that this tiling of $T$ covers, say, 10% of the area of $R$.

4.3.15. 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 $S_1, \ldots, S_k$ in $C$ such that $S \subseteq \bigcup_{i=1}^{k} S_i$?

(A) Prove that MINIMUM SET COVER problem is **NP-Complete**

(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 $n = |S|$.

(C) Prove the greedy algorithm is $O(\log n)$ factor optimal approximation.

(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 $C$.

(E) Given a set $I$ 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 $I \in I$ there is a point $p \in X$, such that $p \in I$. 
### 4.3.16. $k$-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) $r$?

(A) Describe an $n^{O(k)}$ time algorithm that solves this problem.

(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.

(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.

(D) Provide an $\epsilon$-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 $\leq (1 + \epsilon)r_{opt}$, where $r_{opt}$ is the minimum radius of such a cover of $P$.

(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 $P$.

(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 $r$.

### 4.3.17. 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.

(A) Prove that MAX SAT is NP-Hard.

(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 $M = |C|$.

(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.

(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 $C$.

### 4.3.18. Complexity

(A) Prove that $P \subseteq \text{co-NP}$.

(B) Show that if $\text{NP} \neq \text{co-NP}$, then every $\text{NP-Complete}$ problem is not a member of $\text{co-NP}$.

### 4.3.19. 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 $\text{TRUE}$ for the set $\{-5, -17, 7, -4, 3, -2, 4\}$, since $-5 + 7 + (-2) = 0$, and $\text{FALSE}$ for the set $\{-6, 7, -4, -13, -2, 5, 13\}$.
4.3.20. 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?

4.3.21. 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 4.1: Gadget for PLANAR-3-COLOR.

4.3.22. 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 4.2: Gadget for DEGREE-4-PLANAR-3-COLOR.

4.3.23. Primality and Complexity

Prove that PRIMALITY (Given n, is n prime?) is in NP ∩ 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.
4.3.24. 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.

4.3.25. Polynomial time Hamiltonian path

(A) Prove that if \( G \) is an undirected bipartite graph with an odd number of vertices, then \( G \) is nonhamiltonian. Give a polynomial time algorithm algorithm for finding a hamiltonian cycle in an undirected bipartite graph or establishing that it does not exist.

(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.

(C) Explain why the results in previous questions do not contradict the facts that both HAM-CYCLE and HAM-PATH are NP-Complete problems.

4.3.26. (Really HARD)GRAPH-ISOMORPHISM

Consider the problem of deciding whether one graph is isomorphic to another.

(A) Give a brute force algorithm to decide this.

(B) Give a dynamic programming algorithm to decide this.

(C) Give an efficient probabilistic algorithm to decide this.

(D) Either prove that this problem is NP-Complete, give a poly time algorithm for it, or prove that neither case occurs.

4.3.27. \((t, k)\)-grids.

(20 pts.)

A graph \( G \) is a \((t, k)\)-grid if it vertices are

\[
V(G) = \{(i, j) \mid i = 1, \ldots, n/k, j = 1, \ldots, k\},
\]

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 \( G \).

(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).

(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 \( k \))? 

4.3.28. 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.

(A) (5 pts.) Provide a $n^{O(k)}$ time algorithm for this problem.

(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 5

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.

5.1. Basic Idea - Partition Number

Definition 5.1.1. 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 \( d \).

<table>
<thead>
<tr>
<th></th>
<th>6 = 6</th>
<th></th>
<th>6 = 5+1</th>
<th></th>
<th>6 = 4+2</th>
<th>6 = 4+1+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3+3</td>
<td>6</td>
<td>3 + 2 + 1</td>
<td>6</td>
<td>3+1+1+1</td>
<td>6+3+1+1+1</td>
</tr>
<tr>
<td>6</td>
<td>2+2</td>
<td>6</td>
<td>2+2+1+1</td>
<td>6</td>
<td>2+1+1+1</td>
<td>6+2+1+1+1</td>
</tr>
<tr>
<td>6</td>
<td>1+1+1+1+1+1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Suggestion 5.1.2. 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.
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 $O(p(n))$.

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

**Exercise 5.1.3.** Prove the above bounds on $p(n)$ (or better bounds).

Suggestion 5.1.4. Exercises in the class notes are a natural easy questions for inclusions in exams. You probably want to spend time doing them.

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 slowwwwwwwwwwwwwwwwww? 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. 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 PartitionS\_C takes (by itself) more than constant time, then this call performs a store in the cache. Indeed, for PartitionS\_C($num, max\_digit$), the parameters $num$ and $max\_digit$ are both integers in the range $1, \ldots, n$.

(B) Number of store operations in the cache is $O(n^2)$, since this is the number of different entries stored in the cache.

(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).

---

Throughout 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.
5.1.1. 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 5.1.5.** 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].

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**. We can sometime save space and improve running time in dynamic programming over memoization.

**Dynamic programing 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.

5.2. Example – Fibonacci numbers

Let us revisit the classical problem of computing Fibonacci numbers.

5.2.1. 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 (!)

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

---

As 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.
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.)

5.2.2. Computing Fibonacci numbers

The recursive function for computing Fibonacci numbers is depicted on the right. As before, the running time of \( \text{FibR}(n) \) 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 \( \text{FibDP} \) on the right. Clearly, the running time of \( \text{FibDP}(n) \) is linear (i.e., \( O(n) \)).

A careful inspection of \( \text{FibDP} \) exposes the fact that it fills the array \( F[...] \) 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 \( \text{FibI} \) is identical to the running time of \( \text{FibDP} \). Can we do better?

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

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

As such,

\[
\begin{pmatrix} \frac{F_{n-1}}{F_n} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}.
\]

Thus, computing the \( n \)th Fibonacci number can be done by computing \( \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3} \).

\[
\begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}.
\]
How to this quickly? Well, we know that \(a \cdot b \cdot c = (a \cdot b) \cdot c = a \cdot (b \cdot c)\), as such one can compute \(a^n\) by repeated squaring, see pseudo-code on the right. The running time of \(\text{FastExp}\) 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 \(\approx \log_{10} 1.68^{10^n}\) 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.

### 5.3. Edit Distance

We are given two strings \(A\) and \(B\), and we want to know how close the two strings are to 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,
- (iii) replace a character by a different character.

Price of each operation is one unit.

For example, consider the strings \(A = \text{"har-peled"\} and \(B = \text{"sharp eyed"\}. Their edit distance is 4, as can be easily seen.

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 5.1.

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 \(\text{ed}(...)\)? Clearly exponential, and roughly \(2^{n+m}\), where \(n+m\) is the size of the input.

So how many different recursive calls \(\text{ed}\) performs? Only: \(O(m \cdot n)\) different calls, since the only parameters that matter are \(n\) and \(m\).

\(^8\text{Associativity of multiplication...}\)
So the natural thing is to introduce memoization. The resulting algorithm \texttt{edM} is depicted on the right. The running time of \texttt{edM}(n, m) when executed on two strings of length \(n\) and \(m\) respective 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 at 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.

\begin{table}[h]
\begin{tabular}{|c|c|c|c|c|c|c|c|}
\hline
\textbf{A} & \textbf{L} & \textbf{G} & \textbf{O} & \textbf{R} & \textbf{I} & \textbf{T} & \textbf{H} \\
\hline
I & 0 & 1 & 2 & 3 & 4 & 5 & 6 7 & 8 & 9 \\
\hline
A & 1 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
L & 2 & 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
T & 3 & 2 & 1 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
R & 4 & 3 & 2 & 2 & 2 & 2 & 3 & 4 & 5 & 6 \\
\hline
U & 5 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 4 & 5 & 6 \\
\hline
I & 6 & 5 & 4 & 4 & 4 & 4 & 4 & 3 & 4 & 5 & 6 \\
\hline
S & 7 & 6 & 5 & 5 & 5 & 5 & 4 & 4 & 4 & 5 & 6 \\
\hline
T & 8 & 7 & 6 & 6 & 6 & 6 & 5 & 4 & 4 & 5 & 6 \\
\hline
I & 9 & 8 & 7 & 7 & 7 & 7 & 6 & 5 & 5 & 6 \\
\hline
C & 10 & 9 & 8 & 8 & 8 & 8 & 7 & 6 & 6 & 6 \\
\hline
\end{tabular}
\end{table}

\textbf{edDP}(A[1..m], B[1..n])
\begin{verbatim}
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
        p\text{insert} = T[i, j - 1] + 1
        p\text{delete} = T[i - 1, j] + 1
        p\text{replace/ignore} = T[i - 1, j - 1] + [A[i] ≠ B[j]]
        T[i, j] ← min(p\text{insert}, p\text{delete}, p\text{replace/ignore})
return T[m, n]
\end{verbatim}

Figure 5.2: Extracting the edit operations from the table.

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 \texttt{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 5.2. 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 \(T[m, n]\). 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 the remember
the current and previous row to compute the edit distance. We conclude:

**Theorem 5.3.1.** 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 5.3.2.** 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.)

### 5.3.1. 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*). 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.

---

### Chapter 6

**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.

---

*No cycles in the graph – it’s a miracle!*
6.1. 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 6.1.1. Given a set of $n$ (sorted) keys $A[1 \ldots n]$, 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 $i = 1, \ldots, n$. Namely, we know that $A[i]$ is going be accessed $f[i]$ times, for $i = 1, \ldots, n$.

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 left$_\mathcal{T}$, and all values larger than $r$ are in right$_\mathcal{T}$. Thus, the total search time, in this case, is

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

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

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

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

and the solution of this recurrence is $O(n^3)$.

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

A further improvement raises 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 $A[i \ldots j]$ 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.

6.2. 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 6.1.

Definition 6.2.1. 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 6.2.2. 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.
We are given two matrices: (i) a matrix $A$ 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.

Things becomes considerably more interesting when we have to multiply a chain of matrices. Consider for example the three matrices $A$, $B$, and $C$. The product $ABC = A(BC)$ 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 6.3.1. The input is $n$ matrices $M_1, \ldots, M_n$ such that $M_i$ of size $D[i-1] \times D[i]$ (i.e., $M_i$ has $D[i-1]$ rows and $D[i]$ columns), where $D[0 \ldots n]$ is array specifying the sizes. Find the ordering of multiplications to compute $M_1 \cdot M_2 \cdot \cdots \cdot M_{n-1} \cdot M_n$ 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 $M_i \cdot \cdots \cdot M_j$. We have

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

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

Given an array of numbers \( A[1 \ldots n] \) 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.

6.5. Pattern Matching

Assume you have a string \( S = "Magna Carta" \) and a pattern \( P = "?ag*a at*a*" \) 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.

<table>
<thead>
<tr>
<th>IsMatch(S[1 \ldots n], P[1 \ldots m])</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( m = 0 ) and ( n = 0 ) then return TRUE.</td>
</tr>
<tr>
<td>if ( m = 0 ) then return FALSE.</td>
</tr>
<tr>
<td>if ( n = 0 ) then</td>
</tr>
<tr>
<td>if ( P[1 \ldots m] ) is all stars then return TRUE</td>
</tr>
<tr>
<td>else return FALSE</td>
</tr>
<tr>
<td>if ( (P[m] = '?' ) then</td>
</tr>
<tr>
<td>return IsMatch(S[1 \ldots n - 1], P[1 \ldots m - 1])</td>
</tr>
<tr>
<td>if ( (P[m] \neq '*') ) then</td>
</tr>
<tr>
<td>if ( P[m] \neq S[n] ) then return FALSE</td>
</tr>
<tr>
<td>else return IsMatch(S[1 \ldots n - 1], P[1 \ldots m - 1])</td>
</tr>
<tr>
<td>for ( i = 0 ) to ( n ) do</td>
</tr>
<tr>
<td>if IsMatch(S[1 \ldots i], P[1 \ldots m - 1]) then</td>
</tr>
<tr>
<td>return TRUE</td>
</tr>
<tr>
<td>return FALSE</td>
</tr>
</tbody>
</table>

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 \( O(mn) \).

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.
Chapter 7

Approximation algorithms

7.1. 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:

**Definition 7.1.1.** A minimization problem is an optimization problem, where we look for a valid solution that minimizes a certain target function.

**Example 7.1.2.** 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 VertexCover(G) is just the set $S$ realizing this minimum.

**Definition 7.1.3.** Let Opt(G) denote the value of the target function for the optimal solution.
Intuitively, a vertex-cover of size “close” to the optimal solution would be considered to be good.

Definition 7.1.4. 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 $\alpha$-approximation algorithm for Min.

As a concrete example, an algorithm is a 2-approximation for VertexCoverMin, 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 7.1 shows that the approximation factor of GreedyVertexCover is at least $\frac{4}{3}$.

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 $i = 2, \ldots, n$, we add a set $R_i$ of $\lfloor \frac{n}{i} \rfloor$ 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 $i = 2, \ldots, n$. 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 $R_2, \ldots, R_n$ and put them into the vertex-cover. To see that, observe that if $R_2, \ldots, R_i$ 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 $R_2, \ldots, R_{i-1}$ 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 7.1.5.** The algorithm GreedyVertexCover is $\Omega(\log n)$ approximation to the optimal solution to VertexCoverMin.

**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

$$\frac{|R|}{|L|} = \frac{n(H_n - 2)}{n} = \Omega(\log n).$$

**Theorem 7.1.6.** 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 7.1.5. 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. 

50
7.1.1. Alternative algorithm – two for the price of one

One can still do much better than the greedy algorithm in this case. In particular, let \texttt{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 7.1.7.** \texttt{ApproxVertexCover} is a 2-approximation algorithm for \texttt{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. \hfill \blacksquare

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

7.2.1. A silly brute force algorithm for vertex cover

So given a graph \(G = (V, E)\) with \(n\) vertices, we can approximate \texttt{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 7.2.1.** Given a graph \(G = (V, E)\) with \(n\) vertices, one can solve \texttt{VertexCoverMin} in \(O(n^{\alpha+2})\) time, where \(\alpha\) is the size the minimum vertex cover.

7.2.2. 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 7.2.2.** 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 \texttt{VertexCoverMin} that tries all these possibilities. The resulting algorithm \texttt{algFPVertexCover} is depicted in Figure 7.2.

**Lemma 7.2.3.** The algorithm \texttt{algFPVertexCover} (depicted in Figure 7.2) returns the optimal solution to the given instance of \texttt{VertexCoverMin}.

**Proof:** It is easy to verify, that if the algorithm returns \(\beta\) then it found a vertex cover of size \(\beta\). Since the depth of the recursion is at most \(n\), it follows that this algorithm always terminates.
\textbf{fpVertexCoverInner}(X, \beta)

// Computes minimum vertex cover for the induced graph \(G_X\)
// \(\beta\): size of VC computed so far.

if \(X = \emptyset\) or \(G_X\) has no edges then return \(\beta\)
\(e \leftarrow\) any edge \(uv\) of \(G_X\).
\(\beta_1 = \text{fpVertexCoverInner}(X \setminus \{u, v\}, \beta + 2)\)
// 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 \(v\)
\(\beta_2 = \text{fpVertexCoverInner}(X \setminus (\{u\} \cup N_{G_X}(v)), \beta + |N_{G_X}(v)|)\)
// Only take \(v\) to the cover...
\(\beta_3 = \text{fpVertexCoverInner}(X \setminus (\{v\} \cup N_{G_X}(u)), \beta + |N_{G_X}(u)|)\)

return \(\min(\beta_1, \beta_2, \beta_3)\).

\textbf{algFPVertexCover}(G = (V, E))

return \text{fpVertexCoverInner}(V, 0)

Figure 7.2: Fixed parameter tractable algorithm for VertexCoverMin.

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.

\textbf{Lemma 7.2.4.} The depth of the recursion of \textbf{algFPVertexCover}(G) is at most \(\alpha\), where \(\alpha\) is the minimum size vertex cover in \(G\).

\textit{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 \(\alpha\) times.

The more interesting case, is when the algorithm picks \(N_{G_X}(v)\) (i.e., \(\beta_2\)) 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 \(\beta_3\).

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 \(\leq\alpha\).

\textbf{Theorem 7.2.5.} Let \(G\) be a graph with \(n\) vertices, and with the minimal vertex cover being of size \(\alpha\). Then, the algorithm \textbf{algFPVertexCover} (depicted in Figure 7.2) returns the optimal vertex cover for \(G\) and the running time of this algorithm is \(O(3^\alpha n^2)\).

\textit{Proof:} By \textbf{Lemma 7.2.4}, the recursion tree has depth \(\alpha\). 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 \(\alpha\) is some parameter that depends on the problem are \textbf{fixed parameter tractable} algorithms for the given problem.
7.2.2.1. 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 $2 - \Theta(1/\sqrt{\log n})$. If the Unique Games Conjecture is true, then no better constant approximation is possible in polynomial time.

7.3. 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 7.3.1.** 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 7.3.1 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.

7.3.1. TSP with the triangle inequality

7.3.1.1. A 2-approximation

Consider the following special case of TSP:

**TSP_{\triangle}**

- **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 \( \sigma \) 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 \( \sigma \)).

**Definition 7.3.2.** 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 7.3.3.** 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 \( \text{TSP}_{\Delta} \)-Min. To this end, let \( C_{\text{opt}} \) denote the optimal TSP tour in \( G \). Observe that \( C_{\text{opt}} \) 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 covert \( T \) 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 \( \mathcal{C} \) denote the Eulerian cycle in \( H \). Observe that

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

Next, we traverse \( \mathcal{C} \) starting from any vertex \( v \in V(\mathcal{C}) \). As we traverse \( \mathcal{C} \), we skip vertices that we already visited, and in particular, the new tour we extract from \( \mathcal{C} \) will visit the vertices of \( V(G) \) in the order they first appear in \( \mathcal{C} \). Let \( \pi \) denote the new tour of \( G \). Clearly, since we are performing shortcutting, and the triangle inequality holds, we have that \( \omega(\pi) \leq \omega(C_{\text{opt}}) \). The resulting algorithm is depicted in Figure 7.3.

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 7.3.4.** Given an instance of TSP with the triangle inequality (\( \text{TSP}_{\Delta} \)-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 \( C_{\text{opt}} \) is the minimum cost TSP tour of \( G \). The running time of the algorithm is \( O(n^2) \).

### 7.3.1.2. A 3/2-approximation to \( \text{TSP}_{\Delta} \)-Min

Let us revisit the concept of matchings.

**Definition 7.3.5.** Given a graph \( G = (V, E) \), 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 \( G \). Given a weight function \( w \) on the edges, a min-weight perfect matching, is the minimum weight matching among all perfect matching, where

\[
\omega(M) = \sum_{e \in M} \omega(e).
\]
Figure 7.3: The TSP approximation algorithm: (a) the input, (b) the duplicated graph, (c) the extracted Eulerian tour, and (d) the resulting shortcut path.

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

**Theorem 7.3.6.** Given a graph $G$ and weights on the edges, one can compute the min-weight perfect matching of $G$ in polynomial time.

**Lemma 7.3.7.** 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 $\leq \omega(\text{TSP}(G))/2$.

**Proof:** Let $\pi$ be the cycle realizing the TSP in $G$. Let $\sigma$ be the cycle resulting from shortcutting $\pi$ so that it uses only the vertices of $S$. Clearly, $\omega(\sigma) \leq \omega(\pi)$. Now, let $M_o$ and $M_e$ be the sets of even and odd edges of $\sigma$ 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 7.3.3).

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 7.3.8.** 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 $U = \sum_{v \in V(G')} d(v)$ 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

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

55
since $\mu$ 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 7.3.7. 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 7.3.9.** Given an instance of TSP with the triangle inequality, one can compute in polynomial time, a $(3/2)$-approximation to the optimal TSP.

### 7.4. Biographical Notes

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

## Chapter 8

### Approximation algorithms II

#### 8.1. 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: $C_1, \ldots, C_m$.

**Question:** Find the assignment to $x_1, \ldots, x_n$ 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 8.1.1. Algorithm $\text{Alg}$ for a maximization problem achieves an approximation factor $\alpha$ 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 8.1.2 (Linearity of expectations). Given two random variables $X, Y$ (not necessarily independent, we have that $E[X + Y] = E[X] + E[Y]$.

Theorem 8.1.3. 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 $x_1, \ldots, x_n$ be the $n$ variables used in the given instance. The algorithm works by randomly assigning values to $x_1, \ldots, x_n$, 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 $i = 1, \ldots, m$. 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 $E[Y] = (7/8)m$, where $m$ is the number of clauses in the input. Indeed, we have

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

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

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

(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,

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

and

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

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

■
Curiously, Theorem 8.1.3 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\(^3\), 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 \( P = NP \)) to within a factor of \( 7/8 + \varepsilon \). It is pretty amazing that a trivial algorithm like the above is essentially optimal.

### 8.2. Approximation Algorithms for Set Cover

#### 8.2.1. 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 \( \#\text{guards} \leq \sqrt{n k_{opt}} \), 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.

#### 8.2.2. 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 8.2.1.** 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 \( X_{\text{opt}} = \{1, 2, 3\}, \{4, 5\} \).

The greedy algorithm \textbf{GreedySetCover} for this problem is depicted on the right. 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 \( X_{\text{opt}} = \{V_1, \ldots, V_k\} \) be the optimal solution.

Let \( T_i \) denote the elements not covered in the beginning \( i \)th iteration of \textbf{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.

\textbf{Claim 8.2.3.} We have \( \alpha_1 \geq \alpha_2 \geq \ldots \geq \alpha_k \geq \ldots \geq \alpha_m \).

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 that the set used by \textbf{GreedySetCover}. Namely, in the \( i \)th iteration we would use \( U_{i+1} \) instead of \( U_i \). This contradicts the greediness of \textbf{GreedySetCover} of choosing the set covering the largest number of elements not covered yet. A contradiction. \( \blacksquare \)

\textbf{Claim 8.2.4.} 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 \( |T_{i+1}| = |T_i| - \alpha_i \leq (1 - 1/k)|T_i| \). \( \blacksquare \)

\textbf{Theorem 8.2.4.} The algorithm \textbf{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

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

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. \( \blacksquare \)

\subsection{8.2.3. Lower bound}

The lower bound example is depicted in the following figure.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure}
\caption{Lower bound example}
\end{figure}

We provide a more formal description of this lower bound next, and prove that it shows \( \Omega(\log n) \) approximation to \textbf{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 \( S_i = Y_i \cup Z_i \), \( Y_i = \{y_1, \ldots, y_{2^{i-1}}\} \) and \( Z_i = \{z_1, \ldots, z_{2^{i-1}}\} \). The family of sets \( \mathcal{F}_i \) contains the following sets

\[
X_j = \{y_{2^{j-1}}, \ldots, y_{2^{j-1}}, z_{2^{j-1}}, \ldots, z_{2^{j-1}}\},
\]

for \( j = 1, \ldots, i \). Furthermore, \( \mathcal{F}_i \) also contains the two special sets \( Y_i \) and \( Z_i \). Clearly, minimum set cover for \( \Lambda_i \) 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 \( \Lambda_i \). As such, the greedy algorithm \( \text{GreedySetCover} \) would pick \( X_i \) as first set to its cover. However, once you remove \( X_i \) from \( \Lambda_i \) (and from its ground set), you remain with the set system \( \Lambda_{i-1} \). We conclude that \( \text{GreedySetCover} \) would pick the sets \( X_i, X_{i-1}, \ldots, X_1 \) to the cover, while the optimal cover is by two sets. We conclude:

**Lemma 8.2.5.** 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 \( \text{GreedySetCover} \) would use \( i = \lceil \lg n \rceil \) sets for the cover. That is, \( \text{GreedySetCover} \) is a \( \Theta(\log n) \) approximation to SetCover.

### 8.2.4. Just for fun – weighted set cover

**Weighted Set Cover**

**Instance:** \( (S, \mathcal{F}, \rho) \):
- \( S \): 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 \( X \subseteq \mathcal{F} \), such that \( X \) covers \( S \). Formally, \( \bigcup_{X \in \mathcal{X}} X = S \), and \( \rho(X) = \sum_{X \in \mathcal{X}} \rho(X) \) is minimized.

The greedy algorithm in this case, \( \text{WGreedySetCover} \), 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 \( \alpha(X) = \rho(X)/t \). \( \text{WGreedySetCover} \) as such, picks the set with the lowest average price. Our purpose here to prove that this greedy algorithm provides \( O(\log n) \) approximation.

#### 8.2.4.1. 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 8.2.6.** We have that:

(A) \( \alpha_1 \leq \alpha_2 \leq \cdots \)

(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 \( \text{WGreedySetCover} \) 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 8.2.7. 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 $X_1, \ldots, X_m$ be the sets used in the optimal solution. Let $s_j = |U_{i-1} \cap X_j|$, for $j = 1, \ldots, m$, be the number of new elements covered by each one of these sets. Similarly, let $\rho_j = \rho(X_j)$, for $j = 1, \ldots, m$. The average cost of the $j$th set is $\rho_j/s_j$ (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 $a/b \leq a+c/b+d \leq c/d$. In particular, for any such numbers $\min_{j=1}^m \left( \frac{a}{b}, \frac{c}{d} \right) \leq \frac{a+c}{b+d}$, 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 8.2.8. 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 8.2.7, 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 the 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 8.2.9. $W\text{GreedySetCover}$ computes a $O(\log n)$ approximation to the optimal weighted set cover solution.

Proof: $W\text{GreedySetCover}$ paid at most twice the optimal solution to cover half the elements, by Lemma 8.2.8. 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, $W\text{GreedySetCover}$ paid at most twice the optimal cost.

8.3. 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 9

Approximation algorithms III

9.1. 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 $p = (p_1, \ldots, p_d)$ and $q = (q_1, \ldots, q_d)$.

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

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

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

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

$$\nu(X, S) = \max_{p \in X} d(p, S),$$

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 9.1. 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 \( k \) discs/balls of minimum radius. This is known as the \( k \)-center problem.

It is known that \( k \)-center clustering is \textsc{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 \( d(p,q) \), for \( p,q \in P \), with triangle inequality \( d(\cdot, \cdot) \), and a parameter \( k \).

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

#### 9.1.1. 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 9.1. That is, there is a single point which determines 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 \( u_1, \ldots, u_i \) such that they cover \( P \) (where all these balls have the same radius). (Namely, there is a point \( p \in P \) such that \( d(p, \{u_1, \ldots, u_i\}) = r_{i+1} \).

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

Observe, that the running time of the algorithm \texttt{AprxKCenter} is \( O(nk) \) as can be easily verified.

**Lemma 9.1.1.** We have that \( r_2 \geq \ldots \geq r_k \geq r_{k+1} \).

**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 9.1.2.** The radius of the clustering generated by \texttt{AprxKCenter} is \( r_{k+1} \).

**Lemma 9.1.3.** 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: \(D_1, \ldots, D_k\) and consider the \(k\) center points contained in the solution \(S\) computed by \texttt{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 \(2r_{\text{opt}}(P,k)\) 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

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

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 \(\alpha\)th iteration (that is, \(u_\alpha = x\)), and \(y\) was added in a later \(\beta\)th iteration (that is, \(u_\beta = y\), where \(\alpha < \beta\). Then,

\[
r_\beta = d(y, \{u_1, \ldots , u_{\beta-1}\}) \leq d(x, y),
\]

since \(x = u_\alpha\) and \(y = u_\beta\). But \(r_\beta \geq r_{k+1}\), by Lemma 9.1.1. Applying the triangle inequality again, we have that \(r_{k+1} \leq r_\beta \leq d(x, y) \leq d(x, q) + d(q, y) \leq 2r_{\text{opt}}\), implying the claim.

\[\blacksquare\]

**Theorem 9.1.4.** One can approximate the \(k\)-center clustering up to a factor of two, in time \(O(nk)\).

**Proof:** The approximation algorithm is \texttt{AprxKCenter}. The approximation quality guarantee follows from Lemma 9.1.3, since the furthest point of \(P\) from the \(k\)-centers computed is \(r_{k+1}\), which is guaranteed to be at most \(2r_{\text{opt}}\).

\[\blacksquare\]

### 9.2. Subset Sum

**Subset Sum**

**Instance:** \(X = \{x_1, \ldots , x_n\}\) – \(n\) integer positive numbers, \(t\) - target number

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

**Subsets Sum** is (of course) \(\text{NPC}\), 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 \(i = 1, \ldots , n\), 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 prohibitely large, and as such, this algorithm is not polynomial in \(n\). In particular, if \(M = 2^n\) then this algorithm is prohibitely slow. Since the relevant decision problem is \(\text{NPC}\), 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.

**Subset Sum Optimization**

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

**Question:** The largest number \(\gamma_{\text{opt}}\) 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 it sum is smaller than \(t\) but very close to \(t\). Next, we turn problem into an approximation problem.
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 9.2.1.** Let \((X, t)\) be an instance of **Subset Sum**. Let \(\gamma_{\text{opt}}\) 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 \(t/2\). \(\square\)

### 9.2.1. On the complexity of \(\varepsilon\)-approximation algorithms

**Definition 9.2.2 (PTAS).** For a maximization problem \(\text{PROB}\), an algorithm \(A(I, \varepsilon)\) (i.e., \(A\) receives as input an instance of \(\text{PROB}\), and an approximation parameter \(\varepsilon > 0\)) is a **polynomial time approximation scheme (PTAS)** if for any instance \(I\) we have

\[
(1 - \varepsilon) |\text{opt}(I)| \leq |A(I, \varepsilon)| \leq |\text{opt}(I)|,
\]

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

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

**Example 9.2.3.** 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 9.2.4 (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 9.2.5.** 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.

### 9.2.2. Approximating subset-sum

Let \(S = \{a_1, \ldots, a_n\}\) be a set of numbers. For a number \(x\), let \(x + S\) denote the translation of \(S\) by \(x\); namely, \(x + S = \{a_1 + x, a_2 + x, \ldots, a_n + x\}\). 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.

```markdown
**ExactSubsetSum** \((S, t)\)

\(n \leftarrow |S|\)
\(P_0 \leftarrow \{0\}\)

for \(i = 1 \ldots n\) do
    \(P_i \leftarrow P_{i-1} \cup (P_{i-1} + x_i)\)
    Remove from \(P_i\) all elements \(> t\)

return largest element in \(P_n\)
```
Thus, if we would like to turn \texttt{ExactSubsetSum} into a faster algorithm, we need to somehow make the lists $L_i$ smaller. This would be done by removing numbers which are very close together.

Definition 9.2.6. For two positive real numbers $z \leq y$, the number $y$ is a $\delta$-approximation to $z$ if $\frac{y}{1 + \delta} \leq z \leq y$.

The procedure \texttt{Trim} that trims a list $L'$ so that it removes close numbers is depicted on the left.

We can now modify \texttt{ExactSubsetSum} to use \texttt{Trim} to keep the candidate list shorter. The resulting algorithm \texttt{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 \texttt{ExactSubsetSum} algorithm) in the $i$th iteration.

\begin{algorithm}
\caption{\texttt{Trim($L', \delta$)}
\begin{algorithmic}
    \State $L = \langle y_1 \ldots y_m \rangle$
    \Comment $y_i \leq y_{i+1}$, for $i = 1, \ldots, n - 1$.
    \State $\text{curr} \leftarrow y_1$
    \State $L_{\text{out}} \leftarrow \{y_1\}$
    \For{$i = 2 \ldots m$}
        \If{$y_i > \text{curr} \cdot (1 + \delta)$}
            \State Append $y_i$ to $L_{\text{out}}$
            \State $\text{curr} \leftarrow y_i$
        \EndIf
    \EndFor
    \State return $L_{\text{out}}$
\end{algorithmic}
\end{algorithm}

\textbf{Observation 9.2.7.} If $x \in L'$ then there exists a number $y \in L_{\text{out}}$ such that $y \leq x \leq y(1 + \delta)$, where $L_{\text{out}} \leftarrow \text{Trim($L', \delta$)}$.

\begin{algorithm}
\caption{\texttt{ApproxSubsetSum($S$, $t$)}
\begin{algorithmic}
    \State $\text{Assume } S = \{x_1, \ldots, x_n\}$, where
    \Comment $x_1 \leq x_2 \leq \ldots \leq x_n$
    \State $n \leftarrow |S|$, $L_0 \leftarrow \{0\}$, $\delta = \varepsilon/2n$
    \For{$i = 1 \ldots n$}
        \State $E_i \leftarrow L_{i-1} \cup (L_{i-1} + x_i)$
        \State $L_i \leftarrow \text{Trim($E_i$, $\delta$)}$
        \Comment Remove from $L_i$ all elements $> t$.
    \EndFor
    \State return largest element in $L_n$
\end{algorithmic}
\end{algorithm}

\textbf{Claim 9.2.8.} For any $x \in P_i$ there exists $y \in L_i$ such that $y \leq x \leq (1 + \delta)^i y$.

\textit{Proof:} If $x \in P_1$ the claim follows by \textbf{Observation 9.2.7} above. Otherwise, if $x \in P_{i-1}$, then, by induction, there is $y' \in L_{i-1}$ such that $y' \leq x \leq (1 + \delta)^{i-1} y'$. \textbf{Observation 9.2.7} 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 $x \leq \alpha + x_i$, 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 \textbf{Observation 9.2.7}, 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 $x' \leq x \leq (1 + \delta)^i x'$.

\begin{flushright} \hfill $\blacksquare$ \end{flushright}
9.2.2.1. Bounding the running time of \textit{ApproxSubsetSum}

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

**Lemma 9.2.9.** 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. \( \blacksquare \)

**Lemma 9.2.10.** 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 9.2.9, that \( \log_{1+\delta} x = \frac{\ln x}{\ln(1+\delta)} \leq \frac{\ln x}{\ln \exp(\delta/2)} = 2 \frac{\ln x}{\delta} \).

**Observation 9.2.11.** In a list generated by \textit{Trim}, for any number \( x \), there are no two numbers in the trimmed list between \( x \) and \( (1 + \delta)x \).

**Lemma 9.2.12.** We have \( |L_i| = O\left(\frac{n^2 \log n}{e}\right) \), for \( i = 1, \ldots, n \).

**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 \( x_1 \ldots x_{i-1} \) 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 9.2.10. Thus, as \( \delta = \varepsilon/2n \), we have

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

**Lemma 9.2.13.** The running time of \textit{ApproxSubsetSum} is \( O\left(\frac{n^3 \log^2 n}{e}\right) \).

**Proof:** Clearly, the running time of \textit{ApproxSubsetSum} is dominated by the total length of the lists \( L_1, \ldots, L_n \) it creates. Lemma 9.2.12 implies that \( \sum_i |L_i| = O\left(\frac{n^3 \log n}{e}\right) \). The running time of \textit{Trim} is proportional to the size of the lists, implying the claimed running time. \( \blacksquare \)

9.2.2.2. The result

**Theorem 9.2.14.** \textit{ApproxSubsetSum} returns a number \( u \leq t \), such that

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

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

The running time of \textit{ApproxSubsetSum} is \( O\left(\frac{n^3}{e} \log n\right) \).

**Proof:** The running time bound is by Lemma 9.2.13.

As for the other claim, consider the optimal solution \( \text{opt} \in P_n \). By Claim 9.2.8, 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 \textit{ApproxSubsetSum} is within the required range. \( \blacksquare \)
9.3. Approximate Bin Packing

Consider the following problem.

**Min Bin Packing**

**Instance:** \(a_1 \ldots a_n \) – \( n \) 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. It’s 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 \(\lceil A \rceil\) bins where \( A = \sum_{i=1}^{n} a_i \).

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 \(\leq 2 \lceil A \rceil\). As such, the next fit algorithm for bin packing achieves a \(\leq 2 \lceil A \rceil / \lceil A \rceil = 2\) 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**. We state the approximation ratio for this algorithm without proof.

**Theorem 9.3.1.** Decreasing first fit is a \(1.5\)-approximation to Min Bin Packing.

9.4. 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].

Chapter 10

Exercises - Approximation Algorithms

This chapter include problems that are related to approximation algorithms.

10.1. Greedy algorithms as approximation algorithms

10.1.1. 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 proves 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 a factor of (at least) \( c \) longer than the length of the shortest TSP of \( G \).

2. (10 pts.) Show an example, that proves 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 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 \( G \).)

10.1.2. 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 \( \Omega(Opt(G_n) \log n) \), where \( Opt(G_n) \) is the cardinality of the smallest Vertex Cover of \( G_n \).

10.1.3. 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 \( G \).

3. (5 pts.) Let \( G \) be a graph with average degree \( \delta \) (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 \( G_k \) have? What is the average degree of \( G_k \)?


(20 pts.)

Let \( G \) be a graph defined over \( n \) vertices, and let the vertices be ordered: \( v_1, \ldots, v_n \). Let \( G_i \) be the induced subgraph of \( G \) on \( v_1, \ldots, v_i \). Formally, \( G_i = (V_i, E_i) \), where \( V_i = \{v_1, \ldots, v_i\} \) 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 \( 1, \ldots, k_{i-1} \) (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.)
10.1.5. 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 $v_1, \ldots, v_n$ be the resulting ordering, which is known as min last ordering.

1. (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. [3]

2. (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 $n$.

10.2. Approximation for hard problems

10.2.1. Even More on Vertex Cover

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.]

10.2.2. 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 $(v_1, v_2, \ldots, v_k)$ is adjacent to $(w_1, w_2, \ldots, w_k)$ 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.

10.2.3. 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.

---

3 There 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.
10.2.4. Smallest Interval

(20 pts.)

Given a set \( X \) of \( n \) real numbers \( x_1, \ldots, x_n \) (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.

10.2.5. 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 \( \alpha \). 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 \( \beta \) is a constant that depends only on \( \alpha \). Give an explicit formula for the dependency of \( \beta \) on \( \alpha \). What is the running time of your algorithm?

2. (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.

3. (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.)

10.2.6. Graph coloring revisited

1. (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.

2. (5 pts.) Prove that a graph \( G \) with \( m \) edges can be colored using \( 4 \sqrt{m} \) colors.

3. (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 11

Randomized Algorithms

11.1. Some Probability

Definition 11.1.1. (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 11.1.2. The conditional probability of $X$ given $Y$ is

$$\Pr[X = x \mid Y = y] = \frac{\Pr[(X = x) \cap (Y = y)]}{\Pr[Y = y]}.$$ 

An equivalent and useful restatement of this is that

$$\Pr[(X = x) \cap (Y = y)] = \Pr[X = x \mid Y = y] \cdot \Pr[Y = y].$$

Definition 11.1.3. Two events $X$ and $Y$ are independent, if $\Pr[X = x \cap Y = y] = \Pr[X = x] \cdot \Pr[Y = y]$. In particular, if $X$ and $Y$ are independent, then

$$\Pr[X = x \mid Y = y] = \Pr[X = x].$$

Definition 11.1.4. 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

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

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 11.1.5 (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

$$
E[X + Y] = \sum_{x} \sum_{y} (x + y) \Pr[(X = x) \cap (Y = y)]
$$

$$
= \sum_{x} \sum_{y} x \cdot \Pr[(X = x) \cap (Y = y)] + \sum_{x} \sum_{y} y \cdot \Pr[(X = x) \cap (Y = y)]
$$

$$
= \sum_{x} \sum_{y} x \cdot \Pr[(X = x) \cap (Y = y)] + \sum_{y} \sum_{x} y \cdot \Pr[(X = x) \cap (Y = y)]
$$

$$
= \sum_{x} x \cdot \Pr[X = x] + \sum_{y} y \cdot \Pr[Y = y]
$$

$$
= E[X] + E[Y].
$$

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

Definition 11.1.6. Given random variables $X$ and $Y$, the conditional expectation of $X$ given $Y$, is the quantity $E[X \mid Y]$.

Specifically, you are given the value $y$ of the random variable $Y$, and the $E[X \mid Y] = E[X \mid Y = y] = \sum_{x} x \cdot \Pr[X = x \mid Y = y]$.

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

Lemma 11.1.7. For any two random variables $X$ and $Y$ (not necessarily independent), we have that $E[X] = E[E[X \mid Y]]$.

Proof: We use the definitions carefully:

$$
E[E[X \mid Y]] = \sum_{y} E[X \mid Y = y] = \sum_{y} \sum_{x} x \cdot \Pr[X = x \mid Y = y]
$$

$$
= \sum_{y} \Pr[Y = y] \cdot \left( \sum_{x} x \cdot \Pr[X = x \mid Y = y] \right)
$$

$$
= \sum_{y} \Pr[Y = y] \cdot \left( \sum_{x} \frac{\Pr[(X = x) \cap (Y = y)]}{\Pr[Y = y]} \right)
$$

$$
= \sum_{x} \sum_{y} x \cdot \Pr[(X = x) \cap (Y = y)] = \sum_{x} \sum_{y} x \cdot \Pr[(X = x) \cap (Y = y)]
$$

$$
= \sum_{x} x \cdot \left( \sum_{y} \Pr[(X = x) \cap (Y = y)] \right) = \sum_{x} x \cdot \Pr[X = x] = E[X].
$$

11.2. Sorting Nuts and Bolts

Problem 11.2.1 (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 known 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.

11.2.1. Running time analysis

Definition 11.2.2. Let \( \mathcal{R}T \) 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 11.2.3. For a randomized algorithm, we can speak about the expected running time. Namely, we are interested in bounding the quantity \( \mathbb{E}[\mathcal{R}T] \) for the worst input.

Definition 11.2.4. The expected running-time of a randomized algorithm for input of size \( n \) is
\[
T(n) = \max_{U \text{ is an input of size } n} \mathbb{E}[\mathcal{R}T(U)],
\]
where \( \mathcal{R}T(U) \) is the running time of the algorithm for the input \( U \).

Definition 11.2.5. The rank of an element \( x \) in a set \( S \), denoted by 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 11.2.6. 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 \( \Pr[\text{rank}(n_{pivot}) = k] = \frac{1}{n} \). Furthermore, if the rank of the pivot is \( k \) then
\[
T(n) = \mathbb{E}_{k=\text{rank}(n_{pivot})} \left[ O(n) + T(k-1) + T(n-k) \right] = O(n) + \mathbb{E}_k [T(k-1) + T(n-k)]
\]
\[
= T(n) = O(n) + \sum_{k=1}^{n} \Pr[\text{Rank}(Pivot) = k] \cdot (T(k-1) + T(n-k))
\]
\[
= O(n) + \sum_{k=1}^{n} \frac{1}{n} \cdot (T(k-1) + T(n-k)),
\]
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 \( O(n \log n) \).

11.2.1.1. Alternative incorrect solution

The algorithm MatchNutsAndBolts is lucky if \( \frac{2}{3} \leq \text{rank}(n_{pivot}) \leq \frac{3}{4} n \). Thus, \( \Pr[\text{“lucky”}] = 1/2 \). 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
\[
T(n) \leq O(n) + \Pr[\text{“lucky”}] \cdot (T(n/4) + T(3n/4)) + \Pr[\text{“unlucky”}] \cdot T(n).
\]
Namely, \( T(n) = O(n) + \frac{1}{2} \cdot \left( T(\frac{n}{4}) + T(\frac{3n}{4}) \right) + \frac{1}{2} T(n) \). 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.
11.2.2. 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 $\mathcal{R}(n)$, where $n$ denotes the size of the input. In particular, the expected running time $E[\mathcal{R}(n)]$ is a quantity that we would be interested in.

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

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

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

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

Caveat Emptor: Note, 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 $\text{FlipCoins}$ depicted on the right. The expected running time of $\text{FlipCoins}$ is a geometric random variable with probability $1/2$, as such we have that $E[\mathcal{R}(\text{FlipCoins})] = O(2)$. However, $\text{FlipCoins}$ can run forever if it always gets 1 from the $\text{RandBit}$ function.

This is of course a ludicrous argument. Indeed, the probability that $\text{FlipCoins}$ 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 11.2.7. The running time of a randomized algorithm $\text{Alg}$ is $O(f(n))$ with high probability if

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

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

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

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

11.3. Analyzing QuickSort

The previous analysis works also for QuickSort. However, there is an alternative analysis which is also very interesting and elegant. Let $a_1, \ldots, a_n$ 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 11.3.1. 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.
QuickSelect($X,k$)
// Input: $X = \{x_1,\ldots,x_n\}$ numbers, $k$.
// Assume $x_1,\ldots,x_n$ 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 $y$
$X_\leq$ = all elements in $X$ $<$ than $y$
$X_>$ = all elements in $X$ $>$ than $y$
// By assumption $|X_\leq| + |X_>| + 1 = |X|$.
if $r < k$ then
  return QuickSelect($X_>$, $k - r$)
else
  return QuickSelect($X_\leq$, $k$)

Figure 11.1: QuickSelect pseudo-code.

Also, we have that $\mu = \mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1]$. 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 $a_i,\ldots,a_j$. Thus, we have that

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

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 $a_{i+1},\ldots,a_{j-1}$ 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 $2/((j-i+1)$, as claimed.

Thus, the running time of QuickSort is

$$\mathbb{E}[\mathcal{RT}(n)] = \mathbb{E} \left[ \sum_{i<j} X_{ij} \right] = \sum_{i<j} \mathbb{E}[X_{ij}] = \sum_{i<j} \frac{2}{j-i+1} = 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{j-i+1}$$

$$= 2 \sum_{i=1}^{n-1} \sum_{\Delta=2}^{n-i+1} \frac{1}{\Delta} \leq 2 \sum_{i=1}^{n-1} \sum_{\Delta=2}^{n} \frac{1}{\Delta} \leq 2 \sum_{i=1}^{n-1} \frac{1}{\Delta} = 2 \sum_{i=1}^{n-1} H_n = 2nH_n.$$  

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.

11.4. 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 11.1 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 11.4.1. Given a set $X$ of $n$ numbers, and any integer $k$, the expected running time of $\text{QuickSelect}(X, n)$ is $O(n)$.

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

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

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

$$E[n_{i+1} | n_i] \leq \Pr[y_i \text{ in the middle}] \frac{3}{4}n_i + \Pr[y_i \text{ not in the middle}] n_i$$

$$\leq \alpha \frac{3}{4}n_i + (1 - \alpha)n_i = n_i(1 - \alpha/4) \leq n_i(1 - (1/2)/4) = (7/8)n_i.$$ 

Now, we have that

$$m_{i+1} = E[n_{i+1}] = E[E[n_{i+1} | n_i]] \leq E[(7/8)n_i] = (7/8)E[n_i] = (7/8)m_i$$

$$= (7/8)^i m_0 = (7/8)^i n,$$

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

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

as desired. 

Chapter 12

Randomized Algorithms II

12.1. QuickSort and Treaps with High Probability

You must be asking yourself what are treaps. For the answer, see Section 12.3 p83.

One can think about $\text{QuickSort}$ as playing a game in rounds. Every round, $\text{QuickSort}$ 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 \textbf{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 \textbf{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 \textbf{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

$$\alpha = \Pr[C_{QS} \geq 32n \ln n] \leq \Pr[\bigcup_{i=1}^n X_i] \leq \sum_{i=1}^n \Pr[X_i].$$

Here, we used the union bound\textsuperscript{5}, that states that for any two events $A$ and $B$, we have that $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$. Assume, for the time being, that $\Pr[X_i] \leq 1/n^3$. This implies that

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

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

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

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

Consider a run of \textbf{QuickSort} for an input made out of $n$ numbers. Consider a specific element $x$ in this input, and let $S_1, S_2, \ldots$ 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 \textit{lucky}, in the $j$th iteration, if the call to the \textbf{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

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

Furthermore, $Y_1, Y_2, \ldots, Y_m$ are all independent variables.

Note, that $x$ can participate in at most

$$\rho = \log_{4/3} n \leq 3.5 \ln n \quad (12.1)$$

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 $\rho$ 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 need to perform till one gets $\rho$ heads. Of course, in expectation, we need to do this $2\rho$ 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 12.2.

\textbf{Lemma 12.1.1.} In a sequence of $M$ coin flips, the probability that the number of ones is smaller than $L \leq M/4$ is at most $\exp(-M/8)$.

\textsuperscript{5}Also known as Boole’s inequality.
To use Lemma 12.1.1, we set

\[ M = 32 \ln n \geq 8\rho, \]

see Eq. (12.1). 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 \( \Pr[Y_j = 0] = \Pr[Y_j = 1] = 1/2 \) and that \( Y_1, Y_2, \ldots, Y_M \) are independent. By Lemma 12.1.1, we have that the probability that there are only \( \rho \leq M/4 \) ones in \( Y_1, \ldots, Y_M \), 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 \texttt{QuickSort} to be at most \( 1/n^3 \).

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

**Theorem 12.1.2.** With high probability (i.e., \( 1 - 1/n^2 \)) the depth of the recursion of \texttt{QuickSort} is \( \leq 32 \ln n \). Thus, with high probability, the running time of \texttt{QuickSort} is \( O(n \log n) \).

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

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 \( M = 32 t \ln n \), 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 12.2.3 below, the probability for that is

\[
\Pr[Y \geq \Delta] \leq \exp(-\Delta^2/2M) = \exp\left( -\frac{(M - 2h)^2}{2M} \right) = \exp\left( -\frac{(32t - 8)^2 \ln^2 n}{128t \ln n} \right) \\
= \exp\left( -\frac{(4t - 1)^2 \ln n}{2t} \right) \leq \exp\left( -\frac{3t^2 \ln n}{t} \right) \leq \frac{1}{n^{3t}}.
\]

Of course, the same result holds for the algorithm \texttt{MatchNutsAndBolts} for matching nuts and bolts.

12.1.2. **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

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

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 has not shrank significantly (i.e., we pretend it did not shrink at all).

Now, observe that for any two random variables we have that \( \mathbb{E}[X] = \mathbb{E}_y \left[ \mathbb{E}[X \mid Y = y] \right] \), see Lemma 11.1.7p73.. As such, we have that

\[ \mathbb{E}[X_i] = \mathbb{E}_y \left[ \mathbb{E}[X_i \mid X_{i-1} = y] \right] \leq \mathbb{E}_{X_{i-1} = y} \left[ \frac{7}{8} \right] = \frac{7}{8} \mathbb{E}[X_{i-1}] \leq \left( \frac{7}{8} \right)^i \mathbb{E}[X_0] = \left( \frac{7}{8} \right)^i n. \]
In particular, consider $M = 8 \log_{8/7} n$. We have that
\[
\mu = E[X_M] \leq \left(\frac{7}{8}\right)^M n \leq \frac{1}{n^7} n = \frac{1}{n^7}.
\]

Of course, $t$ participates in more than $M$ recursive calls, if and only if $X_M \geq 1$. However, by Markov’s inequality (Theorem 12.2.1), we have that

\[
\Pr\left[\text{element } t \text{ participates in more than } M \text{ recursive calls}\right] \leq \Pr[X_M \geq 1] \leq \frac{E[X_M]}{1} \leq \frac{1}{n^7},
\]
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 $n(1/n^7) \leq 1/n^6$.

### 12.2. Chernoff inequality

#### 12.2.1. Preliminaries

**Theorem 12.2.1 (Markov’s Inequality).** For a non-negative variable $X$, and $t > 0$, we have:

\[
\Pr[X \geq t] \leq \frac{E[X]}{t}.
\]

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

\[
E[X] = \sum_x x \cdot \Pr[X = x] = \sum_{x < t_0} x \cdot \Pr[X = x] + \sum_{x \geq t_0} x \cdot \Pr[X = x]
\]
\[
\geq 0 + t_0 \cdot \Pr[X \geq t_0] > 0 + t_0 \cdot \frac{E[X]}{t_0} = E[X],
\]
a contradiction.

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

\[
\Pr[(X = x) \cap (Y = y)] = \Pr[X = x] \cdot \Pr[Y = y].
\]

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

**Claim 12.2.2.** If $X$ and $Y$ are independent, then $E[XY] = E[X]E[Y]$.

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

#### 12.2.2. Chernoff inequality

**Theorem 12.2.3 (Chernoff inequality).** Let $X_1, \ldots, X_n$ be $n$ independent random variables, such that $\Pr[X_i = 1] = \Pr[X_i = -1] = \frac{1}{2}$, for $i = 1, \ldots, n$. Let $Y = \sum_{i=1}^n X_i$. Then, for any $\Delta > 0$, we have

\[
\Pr[Y \geq \Delta] \leq \exp(-\Delta^2/2n).
\]

**Proof:** Clearly, for an arbitrary $t$, to be specified shortly, we have

\[
\Pr[Y \geq \Delta] = \Pr[ty \geq t\Delta] = \Pr[\exp(tY) \geq \exp(t\Delta)] \leq \frac{E[\exp(tY)]}{\exp(t\Delta)}.
\]

\[\text{(12.2)}\]
where the first part follows since \( \exp(\cdot) \) preserve ordering, and the second part follows by Markov’s inequality (Theorem 12.2.1).

Observe that, by the definition of \( E[\cdot] \) and by the Taylor expansion of \( \exp(\cdot) \), we have

\[
E[\exp(tX_i)] = \frac{1}{2} e^{t} + \frac{1}{2} e^{-t} = \frac{e^{t} + e^{-t}}{2} = \frac{1}{2} \left( 1 + \frac{t}{1!} + \frac{t^2}{2!} + \frac{t^3}{3!} + \cdots \right) + \frac{1}{2} \left( 1 - \frac{t}{1!} + \frac{t^2}{2!} - \frac{t^3}{3!} + \cdots \right) = \left( 1 + \frac{t^2}{2!} + \cdots + \frac{t^{2k}}{(2k)!} + \cdots \right).
\]

Now, \((2k)! = k!(k + 1)(k + 2) \cdots 2k \geq k!2^k\), and thus

\[
E[\exp(tX_i)] = \sum_{i=0}^{\infty} \frac{t^{2i}}{(2i)!} \leq \sum_{i=0}^{\infty} \frac{t^{2i}}{2^{2i}i!} = \sum_{i=0}^{\infty} \frac{1}{i!} \left( \frac{t}{2} \right)^i = \exp \left( \frac{t^2}{2} \right),
\]

again, by the Taylor expansion of \( \exp(\cdot) \). Next, by the independence of the \( X_i \)'s, we have

\[
E[\exp(tY)] = E \left[ \exp \left( \sum_i tX_i \right) \right] = E \left[ \prod_i \exp(tX_i) \right] = \prod_i E[\exp(tX_i)] \leq \prod_i \exp \left( \frac{t^2}{2} \right) = \exp \left( \frac{n t^2}{2} \right).
\]

We have, by Eq. (12.2), that

\[
\Pr[Y \geq \Delta] \leq \frac{E[\exp(tY)]}{\exp(t\Delta)} \leq \exp \left( \frac{n t^2}{2} \right) \exp \left( \frac{n t^2}{2} - t \Delta \right).
\]

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

\[
\Pr[Y \geq \Delta] \leq \exp \left( \frac{n}{2} \left( \frac{\Delta}{n} \right)^2 - \frac{\Delta}{n} \right) = \exp \left( -\frac{\Delta^2}{2n} \right).
\]

Note, the above theorem states that

\[
\Pr[Y \geq \Delta] = \sum_{i=\Delta}^{n} \Pr[Y = i] = \sum_{i=n/2+\Delta/2}^{n} \binom{n}{i} \frac{1}{2^n} \leq \exp \left( -\frac{\Delta^2}{2n} \right),
\]

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 12.2.4.** Let \( X_1, \ldots, X_n \) be \( n \) independent random variables, such that \( \Pr[X_i = 1] = \Pr[X_i = -1] = \frac{1}{2} \), for \( i = 1, \ldots, n \). Let \( Y = \sum_{i=1}^{n} X_i \). Then, for any \( \Delta > 0 \), we have

\[
\Pr[|Y| \geq \Delta] \leq 2 \exp \left( -\frac{\Delta^2}{2n} \right).
\]

By easy manipulation, we get the following result.
Corollary 12.2.5. Let $X_1, \ldots, X_n$ be $n$ independent coin flips, such that $\Pr[X_i = 1] = \Pr[X_i = 0] = \frac{1}{2}$, for $i = 1, \ldots, n$. Let $Y = \sum_{i=1}^{n} X_i$. Then, for any $\Delta > 0$, we have

$$\Pr\left[\frac{Y}{2} - \frac{n}{2} \geq \Delta\right] \leq \exp\left(-\frac{2\Delta^2}{n}\right) \quad \text{and} \quad \Pr\left[Y - \frac{n}{2} \geq \Delta\right] \leq \exp\left(-\frac{2\Delta^2}{n}\right).$$

In particular, we have $\Pr\left[Y - \frac{n}{2} \geq \Delta\right] \leq 2 \exp\left(-\frac{2\Delta^2}{n}\right)$.

Proof: Transform $X_i$ into the random variable $Z_i = 2X_i - 1$, and now use Theorem 12.2.3 on the new random variables $Z_1, \ldots, Z_n$. \hfill \Box

Lemma 12.1.1 (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 $\exp\left(-\frac{M}{8}\right)$.

Proof: Let $Y = \sum_{i=1}^{m} X_i$ the sum of the $M$ coin flips. By the above corollary, we have:

$$\Pr[Y \leq L] = \Pr\left[\frac{M}{2} - Y \geq \frac{M}{2} - L\right] = \Pr\left[\frac{M}{2} - Y \geq \Delta\right],$$

where $\Delta = M/2 - L \geq M/4$. Using the above Chernoff inequality, we get

$$\Pr[Y \leq L] \leq \exp\left(-\frac{2\Delta^2}{M}\right) \leq \exp(-M/8).$$

12.2.2.1. The Chernoff Bound — General Case

Here we present the Chernoff bound in a more general settings.

Problem 12.2.6. Let $X_1, \ldots, X_n$ be $n$ independent Bernoulli trials, where

$$\Pr[X_i = 1] = p_i \quad \text{and} \quad \Pr[X_i = 0] = 1 - p_i,$$

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 12.2.7 (Chernoff inequality). For any $\delta > 0$,

$$\Pr[Y > (1 + \delta)\mu] < \left(\frac{e^\delta}{(1 + \delta)^{1+\delta}}\right)^\mu.$$

Or in a more simplified form, for any $\delta \leq 2e - 1$,

$$\Pr[Y > (1 + \delta)\mu] < \exp\left(-\mu\delta^2/4\right),$$

and

$$\Pr[Y > (1 + \delta)\mu] < 2^{-\mu(1+\delta)},$$

for $\delta \geq 2e - 1$.

Theorem 12.2.8. Under the same assumptions as the theorem above, we have

$$\Pr[Y < (1 - \delta)\mu] \leq \exp\left(-\frac{\delta^2}{2}\right).$$

The proofs of those more general form, follows the proofs shown above, and are omitted. The interested reader can get the proofs from:

12.3. 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.

12.3.1. 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 \( X = \{x_1, \ldots, x_n\} \), with (random) priorities \( p(x_1), \ldots, p(x_n) \), 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 \( T_L \) and \( T_R \). We build the natural tree, by creating a node for \( x_k \), having \( T_L \) its left child, and \( T_R \) 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 12.3.1. Given \( n \) elements, the expected depth of a treap \( T \) 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 \( d \).

Furthermore, the probability that \( T \) 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 12.1.2 of [79].

12.3.2. Operations

The following innocent observation is going to be the key insight in implementing operations on treaps:

Observation 12.3.2. Given \( n \) distinct elements, and their (distinct) priorities, the treap storing them is uniquely defined.

12.3.2.1. Insertion

Given an element \( x \) to be inserted into an existing treap \( T \), insert it in the usual way into \( T \) (i.e., treat it a regular search binary tree). This takes \( O(\text{height}(T)) \). 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.

\(^*\)That is, if we want to decrease the probability of failure, that is \( \delta \), we need to increase \( c \).
We call \textbf{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 \textbf{RotateRight}(D):

\[ \begin{array}{c}
D \quad 0.3 \\
E \quad 0.2 \\
x \quad 0.1 \\
A \quad 0.0 \\
C \quad 0.5 \\
E \quad 0.4 \\
D \quad 0.3 \\
A \quad 0.0 \\
C \quad 0.5 \\
D \quad 0.3 \\
A \quad 0.0 \\
C \quad 0.5 \\
E \quad 0.4 \\
\end{array} \]

\textbf{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 12.3.2, 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 \( O(\text{height}(T)) \).

12.3.2.2. Deletion

Deletion is just an insertion done in reverse. Specifically, to delete an element \( x \) from a treap \( T \), 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.

12.3.2.3. Split

Given an element \( x \) stored in a treap \( T \), we would like to split \( T \) into two treaps – one treap \( T_x \) for all the elements smaller or equal to \( x \), and the other treap \( T_x \) 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 \( T_x \), and we disconnect it from \( T \) 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 \( T_x \). This again takes time that is proportional to the depth of the treap.

12.3.2.4. Meld

Given two treaps \( T_L \) and \( T_R \) such that all the elements in \( T_L \) are smaller than all the elements in \( T_R \), we would like to merge them into a single treap. Find the largest element \( x \) stored in \( T_L \) (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 \( T_L \) has no right child. Attach \( T_R \) as the right child of \( x \). Now, restore \( x \) priority to its original priority, and rotate it back so the priorities properties hold.

12.3.3. Summery

\textbf{Theorem 12.3.3}. Let \( T \) 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 \( \leq \frac{1}{n^f} \), where \( d \) is an arbitrary constant, and \( f \) is a constant that depends only on \( c \) and \( d \).

In particular, a treap can handle insertion/deletion in \( O(\log n) \) time with high probability.

84
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 \( \mathcal{T}_1, \ldots, \mathcal{T}_m \) 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 12.3.1, the probability that \( \mathcal{T}_i \) has large depth is tiny. Specifically, we have that

\[
\alpha_i = \Pr[\text{depth}(\mathcal{T}_i) > tc' \log n^c] = \Pr[\text{depth}(\mathcal{T}_i) > c' \left( \frac{\log n^c}{\log |\mathcal{T}_i|} \right) \cdot \log |\mathcal{T}_i|] \leq \frac{1}{n^{t_c}},
\]

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 \( 1/n^t \). 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 \( 1/n^t \), as desired.

12.4. 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 13

Min Cut

I built on the sand
And it tumbled down,
I built on a rock
And it tumbled down.
Now when I build, I shall begin
With the smoke from the chimney.

– Leopold Staff, Foundations.

13.1. Min Cut

13.1.1. Problem Definition

Let \( G = (V, E) \) be undirected graph with \( n \) vertices and \( m \) edges. We are interested in cuts in \( G \).

Definition 13.1.1. 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

\[
(S, V \setminus S) = \left\{ uv \mid u \in S, v \in V \setminus S, \text{ and } uv \in E \right\},
\]

where \( S \neq \emptyset \) and \( V \setminus S \neq \emptyset \). We will refer to the number of edges in the cut \( (S, V \setminus S) \) as 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.

### 13.1.2. Some Definitions

We remind the reader of the following concepts. The conditional probability of \( X \) given \( Y \) is \( \Pr[X = x | Y = y] = \Pr[(X = x) \cap (Y = y)] / \Pr[Y = y] \). An equivalent, useful restatement of this is that

\[
\Pr[(X = x) \cap (Y = y)] = \Pr[X = x | Y = y] \cdot \Pr[Y = y].
\]  

(13.1)

Two events \( X \) and \( Y \) are independent, if \( \Pr[X = x \cap Y = y] = \Pr[X = x] \cdot \Pr[Y = y] \). In particular, if \( X \) and \( Y \) are independent, then \( \Pr[X = x | Y = y] = \Pr[X = x] \).

The following is easy to prove by induction using Eq. (13.1).

**Lemma 13.1.2.** Let \( \mathcal{E}_1, \ldots, \mathcal{E}_n \) be \( n \) events which are not necessarily independent. Then,

\[
\Pr[\cap_{i=1}^n \mathcal{E}_i] = \Pr[\mathcal{E}_1] \ast \Pr[\mathcal{E}_2 | \mathcal{E}_1] \ast \Pr[\mathcal{E}_3 | \mathcal{E}_1 \cap \mathcal{E}_2] \ast \cdots \ast \Pr[\mathcal{E}_n | \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_{n-1}].
\]  

### 13.2. The Algorithm

The basic operation used by the algorithm is edge contraction, depicted in Figure 13.1. 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 13.2.

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 \( w \)).

**Observation 13.2.1.** 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 13.3, 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 13.3: (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.

Figure 13.3 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? The failure occurs because of the contraction at Figure 13.3 (e), as we had contracted an edge in the minimum cut. In the new graph, depicted in Figure 13.3 (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.

13.2.1. The resulting algorithm

Observation 13.2.2. Let $e_1, \ldots, e_{n-2}$ be a sequence of edges in $G$, such that none of them is in the minimum cut, and such that $G' = G/\{e_1, \ldots, e_{n-2}\}$ is a single multi-edge. Then, this multi-edge corresponds to a minimum cut in $G$.

---

Naturally, if the algorithm had succeeded in finding the minimum cut, this would have been our success.
Algorithm MinCut(G)

\[ G_0 \leftarrow G \]
\[ i = 0 \]
\[ \text{while } G_i \text{ has more than two vertices do} \]
\[ e_i \leftarrow \text{random edge from } E(G_i) \]
\[ G_{i+1} \leftarrow G_i/e_i \]
\[ i \leftarrow i + 1 \]
\[ \text{Let } (S, V \setminus S) \text{ be the cut in the original graph corresponding to the single edge in } G_i \]
\[ \text{return } (S, V \setminus S). \]

Figure 13.4: The minimum cut algorithm.

13.2.1.1. On the art of randomly picking an edge

Every edge has a weight associated with it (which is the number of edges in the original graph it represents). A vertex weight is the total weight associated with it. We maintain during the contraction for every vertex the total weight of the edges adjacent to it. We need the following easy technical lemma.

Lemma 13.2.3. Let \( X = \{x_1, \ldots, x_n\} \) be a set of \( n \) elements, and let \( \omega(x_i) \) be an integer positive weight. One can pick randomly, in \( O(n) \) time, an element from the set \( X \), with the probability of picking \( x_i \) being \( \omega(x_i)/W \), where \( W = \sum_{i=1}^{n} \omega(x_i) \).

Proof: Pick randomly a real number \( r \) in the range 0 to \( W \). We also precompute the prefix sums \( \beta_i = \sum_{k=1}^{i} \omega(x_k) = \beta_{i-1} + \omega(x_i) \), for \( i = 1, \ldots, n \), which can be done in linear time. Now, find the first index \( i \), such that \( \beta_{i-1} < r \leq \beta_i \). Clearly, the probability of \( x_i \) to be picked is exactly \( \omega(x_i)/W \).

Now, we pick a vertex randomly according to the vertices weight in \( O(n) \) time, and we pick randomly an edge adjacent to a vertex in \( O(n) \) time (again, according to the weights of the edges). Thus, we uniformly sample an edge of the graph, with probability proportional to its weight, as desired.

13.2.2. Analysis

13.2.2.1. 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 13.2.4. 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 13.2.5. 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 13.2.6.** MinCut outputs the mincut with probability \( \geq \frac{2}{n(n-1)} \).

**Proof:** Let \( E_i \) be the event that \( e_i \) is not in the minimum cut of \( G_i \). By Observation 13.2.2, MinCut outputs the minimum cut if the events \( E_0, \ldots, E_{n-3} \) all happen (namely, all edges picked are outside the minimum cut).

By Lemma 13.2.5, it holds

\[
\Pr[E_i \mid E_0 \cap E_1 \cap \ldots \cap E_{i-1}] \geq 1 - \frac{2}{|V(G_i)|} = 1 - \frac{2}{n-i}.
\]

Implying that

\[
\Delta = \Pr[E_0 \cap \ldots \cap E_{n-3}]
\]

\[
= \Pr[E_0] \cdot \Pr[E_1 \mid E_0] \cdot \Pr[E_2 \mid E_0 \cap E_1] \cdot \ldots \cdot \Pr[E_{n-3} \mid E_0 \cap \ldots \cap E_{n-4}]
\]

As such, we have

\[
\Delta \geq \prod_{i=0}^{n-3} \left(1 - \frac{2}{n-i}\right) = \prod_{i=0}^{n-3} \frac{n-i-2}{n-i} = \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n \cdot (n-1)}.
\]

**13.2.2.2. Running time analysis.**

**Observation 13.2.7.** MinCut runs in \( O(n^2) \) time.

**Observation 13.2.8.** The algorithm always outputs a cut, and the cut is not smaller than the minimum cut.

**Definition 13.2.9.** (informal) 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 13.2.10.** 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 13.2.6. 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 \( 0 \leq x \leq 1 \).

**Theorem 13.2.11.** 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.
Figure 13.5: \texttt{Contract}(G, t) shrinks G till it has only \(t\) vertices. \texttt{FastCut} computes the minimum cut using \texttt{Contract}.

### 13.3. 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 \texttt{MinCutRep} needs so many executions? Well, the probability of success in the first \(\nu\) iterations is

\[
\Pr\left[ E_0 \cap \ldots \cap E_{\nu-1} \right] \geq \prod_{i=0}^{\nu-1} \left( 1 - \frac{2}{n-i} \right) = \prod_{i=0}^{\nu-1} \frac{n-i-2}{n-i} = \frac{n-2}{n} \ast \frac{n-3}{n-1} \ast \frac{n-4}{n-2} \ldots = \frac{(n-\nu)(n-\nu-1)}{n \cdot (n-1)}. \tag{13.2}
\]

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 \texttt{Contract}, depicted in Figure 13.5, which also depict the new algorithm \texttt{FastCut}.

**Lemma 13.3.1.** The running time of \texttt{FastCut}(\(G\)) is \(O(n^2 \log n)\), where \(n = |V(G)|\).

**Proof:** Well, we perform two calls to \texttt{Contract}(\(G, t\)) which takes \(O(n^2)\) time. And then we perform two recursive calls on the resulting graphs. We have:

\[
T(n) = O(n^2) + 2T\left( \frac{n}{\sqrt{2}} \right)
\]

The solution to this recurrence is \(O(n^2 \log n)\) as one can easily (and should) verify.

**Exercise 13.3.2.** Show that one can modify \texttt{FastCut} so that it uses only \(O(n^2)\) space.

---

\(^2\)This would require a more involved algorithm, that is life.
Lemma 13.3.3. The probability that \( \text{Contract}(G,n/\sqrt{2}) \) 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 \( \nu = n-t = n - \left[ 1 + n/\sqrt{2} \right] \) into Eq. (13.2). We have

\[
\Pr[\mathcal{E}_0 \cap \ldots \cap \mathcal{E}_{n-1}] \geq \frac{n(t-1)}{n \cdot (n-1)} = \frac{\left[ 1 + n/\sqrt{2} \right] \left( 1 + n/\sqrt{2} \right) - 1}{n(n-1)} \geq \frac{1}{2}.
\]

The following lemma bounds the probability of success. A more elegant argument is given in Section 13.3.1 below.

Lemma 13.3.4. FastCut finds the minimum cut with probability larger than \( \Omega(1/\log n) \).

Proof: Do not read this proof – a considerably more elegant argument is given in Section 13.3.1.

Let \( P(n) \) be the probability that the algorithm succeeds on a graph with \( n \) vertices.

The probability to succeed in the first call on \( H_1 \) is the probability that \( \text{Contract} \) did not hit the minimum cut (this probability is larger than \( 1/2 \) by Lemma 13.3.3), times the probability that the algorithm succeeded on \( H_1 \) in the recursive call (those two events are independent). Thus, the probability to succeed on the call on \( H_1 \) is at least \( (1/2) \cdot P(n/\sqrt{2}) \).

Thus, the probability to fail on \( H_1 \) is \( \leq 1 - \frac{1}{2} P\left( \frac{n}{\sqrt{2}} \right) \).

The probability to fail on both \( H_1 \) and \( H_2 \) is smaller than

\[
\left( 1 - \frac{1}{2} P\left( \frac{n}{\sqrt{2}} \right) \right)^2,
\]

since \( H_1 \) and \( H_2 \) are being computed independently. Note that if the algorithm, say, fails on \( H_1 \) but succeeds on \( H_2 \) then it succeeds to return the mincut. Thus the above expression bounds the probability of failure. And thus, the probability for the algorithm to succeed is

\[
P(n) \geq 1 - \left( 1 - \frac{1}{2} P\left( \frac{n}{\sqrt{2}} \right) \right)^2 = P\left( \frac{n}{\sqrt{2}} \right) - \frac{1}{4} \left( P\left( \frac{n}{\sqrt{2}} \right) \right)^2.
\]

We need to solve this recurrence. (This is very tedious, but since the details are non-trivial we provide the details of how to do so.) Divide both sides of the equation by \( P(n/\sqrt{2}) \) we have:

\[
\frac{P(n)}{P(n/\sqrt{2})} \geq 1 - \frac{1}{4} P(n/\sqrt{2}).
\]

It is now easy to verify that this inequality holds for \( P(n) \geq c/\log n \) (since the worst case is \( P(n) = c/\log n \) we verify this inequality for this value). Indeed,

\[
\frac{c/\log n}{c/\log(n/\sqrt{2})} \geq 1 - \frac{c}{4 \log(n/\sqrt{2})}.
\]

As such, letting \( \Delta = \log n \), we have

\[
\frac{\log n - \log \sqrt{2}}{\log n} = \frac{\Delta - \log \sqrt{2}}{\Delta} \geq \frac{4(\log n - \log \sqrt{2}) - c}{4(\log n - \log \sqrt{2})} = \frac{4(\Delta - \log \sqrt{2}) - c}{4(\Delta - \log \sqrt{2})}.
\]

Equivalently, \( 4(\Delta - \log \sqrt{2})^2 \geq 4\Delta(\Delta - \log \sqrt{2}) - c\Delta \). Which implies \( -8\Delta \log \sqrt{2} + 4 \log^2 \sqrt{2} \geq -4\Delta \log \sqrt{2} - c\Delta \). Namely,

\[
c\Delta - 4\Delta \log \sqrt{2} + 4 \log^2 \sqrt{2} \geq 0,
\]

which clearly holds for \( c \geq 4 \log \sqrt{2} \).

We conclude, that the algorithm succeeds in finding the minimum cut in probability

\[
\geq 2 \log 2/\log n.
\]

(Note that the base of the induction holds because we use brute force, and then \( P(i) = 1 \) for small \( i \).)
Exercise 13.3.5. 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 13.3.6. 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 13.3.1. The bound on the probability follows from Lemma 13.3.4, and using the amplification analysis as done in Lemma 13.2.10 for MinCutRep.

13.3.1. On coloring trees and min-cut

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. 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 $E_h$ denote this event, and let $\rho_h = \Pr[E_h]$. 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 $\rho_{h-1}$, 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 $\Pr$[the edge $uu_l$ is colored black] $\cdot \rho_{h-1} / 2$. 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

$$\rho_h = 1 - \left(1 - \frac{\rho_{h-1}}{2}\right)^2 = \frac{\rho_{h-1}}{2} \left(2 - \frac{\rho_{h-1}}{2}\right) = \rho_{h-1} - \frac{\rho_{h-1}^2}{4}.$$

Lemma 13.3.7. 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

$$\frac{1}{h} - \frac{1}{4h^2} \geq \frac{1}{h + 1} \iff 4h(h + 1) - (h + 1) \geq 4h^2 \iff 4h^2 + 4h - 1 \geq 4h^2 \iff 3h \geq 1,$$

which trivially holds.

The recursion tree for FastCut corresponds to such a coloring. Indeed, it is a binary tree as every call performs two recursive calls. Inside such a call, we independently perform two (independent) contractions reducing the given graph with $n$ vertices to have $n/\sqrt{2}$ vertices. If this contraction succeeded (i.e., it did not hit the min-cut), then consider this edge to be colored by black (and white otherwise). Clearly, the algorithm succeeds, if and only if, there is black colored path from the root of the tree to the leaf. Since the tree has depth $H \leq 2 + \log \sqrt{n}$, and by Lemma 13.3.7, we have that the probability of FastCut to succeed is at least $1/(h + 1) \geq 1/(3 + \log \sqrt{n})$.

Galton-Watson processes. Imagine that you start with a single node. The node is going to have two children, and each child survives with probability half (independently). If a child survives it is going to have two children, and so on. Clearly, a single node give a rise to a random tree. The natural question is what is the probability that the original node has descendants $h$ generations in the future. In the above we proved that this probability is at least $1/(h + 1)$. See below for more details on this interpretation.
13.4. 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 13.3.1). It is inspired by Galton-Watson processes (which is a special case of a branching process). The problem that initiated the study of these processes goes back to the 19th century [WG75]. 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 $X \in \{0, 1, 2, \ldots\}$. 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 $\mathbb{E}[X] > 1$. In our case, $X$ was the number of the two children of a node that their edges were colored black.

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. Interestingly, countries with family names that were introduced long time ago have very few surnames (i.e., Koreans 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).

Chapter 14

Randomized Algorithms

This chapter include problems on randomized algorithms

14.1. Randomized algorithms

14.1.1. 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 $x_1, x_2, \ldots, x_n$. After any prefix $x_1, \ldots, x_{i-1}$ has been considered, the algorithm has computed the $k$th smallest element in $x_1, \ldots, x_{i-1}$ (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.
1. (5 pts.) Describe an incremental algorithm for computing the \(k\)th smallest element.

2. (5 pts.) How many comparisons does your algorithm perform in the worst case?

3. (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.

**14.1.2. Minimum Cut Festival**

(20 pts.)

1. 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.

2. For any \(\alpha \geq 1\), define an \(\alpha\) approximate cut in a multigraph \(G\) as any cut whose cardinality is within a multiplicative factor \(\alpha\) 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 \(\alpha\)-approximate cut in \(G\).

3. 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 \(G\).

4. Formulate and prove a similar result of the number of \(\alpha\)-approximate cuts in a multigraph \(G\).

**14.1.3. 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.

1. (10 pts.) Show that there are graphs in which the probability that this algorithm finds an \(s-t\) min-cut is exponentially small.

2. (10 pts.) How large can the number of \(s-t\) min-cuts in an instance be?

**14.1.4. 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.

(a) 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.).

(b) 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 \(n^{0.9}\).
14.1.5. 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 $U = \{1, \ldots, m\}$, and $p = m + 1$ is a prime.
   Let $W \subseteq U$, such that $n = |W|$, and $s$ an integer number larger than $n$. Let $g_k(x, s) = (kx \mod p) \mod s$.
   Let $\beta(k, j, s) = |\{x \mid x \in W, g_k(x, s) = j\}|$. Prove that
   \[
   \sum_{k=1}^{p-1} \sum_{j=1}^{s} \left( \frac{\beta(k, j, s)}{2} \right) < \frac{(p-1)n^2}{s}.
   \]

2. (2 pts.) Prove that there exists $k \in U$, such that
   \[
   \sum_{j=1}^{s} \left( \frac{\beta(k, j, s)}{2} \right) < \frac{n^2}{s}.
   \]

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 $h(x) = (k'x \mod p) \mod n^2$ 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).

14.1.6. Sorting Random Numbers

(20 pts.)
Suppose we pick a real number $x_i$ at random (uniformly) from the unit interval, for $i = 1, \ldots, n$.

1. (5 pts.) Describe an algorithm with an expected linear running time that sorts $x_1, \ldots, x_n$.

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, the 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 15

Network Flow

15.1. 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 splitable, 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.

Definition 15.1.1. 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 15.1.

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 15.1.2 (flow). A flow in a network is a function $f(\cdot, \cdot)$ on the edges of $G$ such that:
15.2. 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 15.2.1.** \( |f| = f(s, V) \).

**Lemma 15.2.2.** 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 \( (u, u) \) is not an edge in the graph, and as such its flow is zero. As for property (ii), we have

\[
f(X, X) = \sum_{(u,v) \in X \cup \{u\}} (f(u, v) + f(v, u)) + \sum_{u \in X} f(u, u) = \sum_{(u,v) \in X \cup \{u\}} f(u, v) + f(v, u) + 0 = 0,
\]

by the anti-symmetry property of flow (Definition 15.1.2 (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 15.2.3.** \( |f| = f(V, t) \).

---

(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 \( \sum_{v} f(u, v) = 0 \).

(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 edge can be negative (i.e., there is a positive flow flowing on this edge in the other direction).

**Problem 15.1.3 (Maximum flow).** Given a network \( G \) find the **maximum flow** in \( G \). Namely, compute a legal flow \( f \) such that \( |f| \) is maximized.
We have:

\[
|f| = f(s, V) = f(V \setminus (V \setminus \{s\}), V)
\]
\[
= f(V, V) - f(V \setminus \{s\}, V)
\]
\[
= -f(V \setminus \{s\}, V) = f(V, V \setminus \{s\})
\]
\[
= f(V, t) + f(V, V \setminus \{s, t\})
\]
\[
= f(V, t) + \sum_{u \in V \setminus \{s, t\}} f(V, u)
\]
\[
= f(V, t) + \sum_{u \in V \setminus \{s, t\}} 0
\]
\[
= f(V, t),
\]

since \(f(V, V) = 0\) by Lemma 15.2.2 (i) and \(f(V, u) = 0\) by Lemma 15.2.2 (iv).

**Proof:** We have:

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 15.2.4.** Given capacity \(c\) and flow \(f\), the residual capacity of an edge \((u, v)\) is

\[c_f(u, v) = c(u, v) - f(u, v).\]

**Definition 15.2.5.** 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

\[E_f = \left\{(u, v) \in V \times V \mid c_f(u, v) > 0\right\}.
\]

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 15.2 for an example of a residual network.

Since every edge of \(G\) induces at most two edges in \(E_f\), it follows that \(G_f\) has at most twice the number of edges of \(G\); formally, \(|E_f| \leq 2|E|\).

**Lemma 15.2.6.** 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 15.2.7.** Given a flow network $G = (V, E)$, a flow $f$ in $G$, and $h$ be a flow 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 15.1.2.

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

$$|f + h| = (f + h)(s, V) = f(s, V) + h(s, V) = |f| + |h|.$$ □

**Definition 15.2.8.** For $G$ and a flow $f$, a path $\pi$ in $G_f$ between $s$ and $t$ is an **augmenting path**.

Note, that all the edges of $\pi$ 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 $\pi$, we can improve $f$ by pushing a positive amount of flow along the augmenting path $\pi$. An augmenting path is depicted on the right, for the network flow of Figure 15.2.

**Definition 15.2.9.** For an augmenting path $\pi$ let $c_f(\pi)$ be the maximum amount of flow we can push through $\pi$. We call $c_f(\pi)$ the **residual capacity** of $\pi$. Formally,

$$c_f(\pi) = \min_{(u, v) \in \pi} c_f(u, v).$$

We can now define a flow that realizes the flow along $\pi$.

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 15.2.10.** For an augmenting path $\pi$, the flow $f_\pi$ 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 15.2.11.** Let $f$ be a flow, and let $\pi$ be an augmenting path for $f$. Then $f + f_\pi$ is a “better” flow. Namely,

$$|f + f_\pi| = |f| + |f_\pi| > |f|.$$ Namely, $f + f_\pi$ is flow with larger value than $f$. Consider the flow in Figure 15.4.

![Figure 15.3: An augmenting path for the flow of Figure 15.2.](image)

![Figure 15.4: The flow resulting from applying the residual flow $f_\pi$ of the path $p$ of Figure 15.3 to the flow of Figure 15.2.](image)
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?

### 15.3. The Ford-Fulkerson method

**Definition 15.4.1.** 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 15.4.2.** 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 \( f(S - s, V) = \sum_{u \in S} f(u, V) = 0 \) by Lemma 15.2.2 (v) (note that \( u \) can not be \( t \) as \( t \in T \)).

**Claim 15.4.3.** 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 15.4.4 (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:

1. \( f \) is a maximum flow in \( G \).
2. The residual network \( G_f \) contains no augmenting paths.
3. \(|f| = c(S, T)\) 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

\[ S = \{ v \mid \text{Exists a path between } s \text{ and } v \text{ in } G_f \} \]

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 15.4.3, 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 \(\text{mtdFordFulkerson}\) terminates, then it had computed the maximum flow. What is still allusive is showing that the \(\text{mtdFordFulkerson}\) method always terminates. This turns out to be correct only if we are careful about the way we pick the augmenting path.

Chapter 16

Network Flow II - The Vengeance

16.1. Accountability

The comic in Figure 16.1 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).

Thus, by just learning maximum flows (and not knowing Latin) you can double your future salary!

![Comic Image](http://www.cs.berkeley.edu/~jrs/)

Figure 16.1: [Link to Comic Image]
16.2. The Ford-Fulkerson Method

The mtdFordFulkerson method is depicted on the right.

Lemma 16.2.1. 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 \( m = |E(G)| \).

Proof: Observe that the mtdFordFulkerson method performs only subtraction, addition and min operations. Thus, if it finds an augmenting path \( \pi \), 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 16.2.2 (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.

16.3. 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 \( \pi \) using BFS in \( G_f \).

Definition 16.3.1. 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 16.3.5 below), and first show its implications.

Lemma 16.3.2. 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 \( \pi \). Furthermore, this edge was fully utilized; namely, \( c_f(\pi) = c_f(\mu(v)) \), 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 \( \sigma \) that contained the edge \((v, u)\). Let \( g \) be the flow used to compute \( \sigma \). We have, by the monotonicity of \( \delta(\cdot) \) [i.e., Lemma 16.3.5 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 (magical?) reappearance. Since \( \delta_0(u) \geq 0 \) and the maximum value of \( \delta_0(u) \) is \( n \), it follows that \((u, v)\) can disappear and reappear at most \( n/2 \) times during the execution of the Edmonds-Karp algorithm.

The careful reader would observe that \( \delta(\cdot) \) 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.
There are now two possibilities:  

(i) The 

(ii) The

Thus, the

Every edge might disappear at most 

By the choice of

(i) (ii) (iii)

Performing a single iteration of the algorithm boils down to computing an Augmenting path. Computing such a path takes

Lemma 16.3.4. The 

Proof: Every edge might disappear at most 

Performing a single iteration of the algorithm boils down to computing an Augmenting path. Computing such a path takes time as we have to perform BFS to find the augmenting path. It follows, that the overall running time of the algorithm is

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 16.3.5. If the 

Proof: Assume, for the sake of contradiction, that this is false. Consider the flow just after the first iteration when this claim failed. Let denote the flow before this (fatal) iteration was performed, and let be the flow after.

Let be the vertex such that is minimal, among all vertices for which the monotonicity fails. Formally, this is the vertex where is minimal and .

Let be the shortest path in from to . Clearly, .

By the choice of it must be that , since otherwise the monotonicity property fails for , and is closer to than in , and this, in turn, contradicts our choice of as being the closest vertex to that fails the monotonicity property. There are now two possibilities:

(i) If then

This contradicts our assumptions that .

(ii) If then the augmenting path used in computing contains the edge . Indeed, the edge reappeared in the residual graph (while not being present in ). The only way this can happens is if the augmenting path pushed a flow in the other direction on the edge . Namely, . However, the algorithm always augment along the shortest path. Thus, since by assumption , we have

Figure 16.2: (i) A bipartite graph. (ii) A maximum matching in this graph. (iii) A perfect matching (in a different graph).
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 $v$.

16.4. Applications and extensions for Network Flow

16.4.1. Maximum Bipartite Matching

Definition 16.4.1. 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 16.2 for examples of these definitions.

Theorem 16.4.2. 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 16.3. 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 $O(nm^2)$.

16.4.2. 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 16.4. Clearly, computing flow in both networks in equivalent.

Figure 16.4: (i) A flow network with several sources and sinks, and (ii) an equivalent flow network with a single source and sink.
Chapter 17

Network Flow III - Applications

17.1. Edge disjoint paths

17.1.1. Edge-disjoint paths in a directed graphs

**Question 17.1.1.** 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 17.1.2.** 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 17.1.3 (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 17.1.4.** Let $f$ be a 0/1 flow in a network $J$ with flow value $\mu$. Then there are $\mu$ 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 $\pi$ 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 $\pi$ this forms $\mu$ 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 $\mu$ 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 17.1.4. As such, we get:

**Theorem 17.1.5.** 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:
Lemma 17.1.6. 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 17.1.5, there are $|F|$ edge disjoint paths in $G$ (since $|F|$ is the amount of the maximum flow).

17.1.2. Edge-disjoint paths in undirected graphs

We would like to solve the $s$-$t$ disjoint path problem for an undirected graph.

Problem 17.1.7. 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 17.1.1 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 17.1.8. 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.

17.2. Circulations with demands

17.2.1. 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.
Definition 17.2.1. A circulation with demands \(\{d_v\}\) is a function \(f\) that assigns non-negative 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 17.2.2. Is there a circulation that comply with the demand requirements?

See Figure 17.1 and Figure 17.2 for an example.

Lemma 17.2.3. 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.

17.2.1.1. The algorithm for computing a circulation

The algorithm performs the following steps:

(A) \(G = (V, E)\) - input flow network with demands on vertices.
(B) Check that \(D = \sum_{v,d_v>0} d_v = \sum_{v,d_v<0} -d_v\).
(C) 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 \(-d_v\).
(D) 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 \(d_u\).
(E) 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 17.2.4. 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.

17.3. 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 transfers 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:

\[
L_v = f_0^{\text{in}}(v) - f_0^{\text{out}}(v) = \sum_{e \text{ into } v} \ell(e) - \sum_{e \text{ out of } v} \ell(e).
\]

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 \( c'_e = c(e) - \ell(e) \).

Lemma 17.3.1. 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,

\[
f^{\text{in}}(v) - f^{\text{out}}(v) = \sum_{e \text{ into } v} (\ell(e) + f'(e)) - \sum_{e \text{ out of } v} (\ell(e) + f'(e)) = L_v + (d_v - L_v) = d_v.
\]

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 \( G' \).

17.4. Applications

17.4.1. 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 17.4.1.** Given \(n\) consumers and \(u\) products with their constraints \(c_1, c'_1, c_2, c'_2, \ldots, c_n, c'_n, p_1, p'_1, \ldots, p_u, p'_u\) 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 18**

**Network Flow IV - Applications II**

**18.1. Airline Scheduling**

**Problem 18.1.1.** 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 18.1 (i).

1: Boston (depart 6 A.M.) - Washington DC (arrive 7 A.M.),
2: Urbana (depart 7 A.M.) - Champaign (arrive 8 A.M.)
3: Washington (depart 8 A.M.) - Los Angeles (arrive 11 A.M.)
4: Urbana (depart 11 A.M.) - San Francisco (arrive 2 P.M.)
5: San Francisco (depart 2:15 P.M.) - Seattle (arrive 3:15 P.M.)
6: Las Vegas (depart 5 P.M.) - Seattle (arrive 6 P.M.).

Figure 18.1: (i) a set \(\mathcal{F}\) of flights that have to be served, and (ii) the corresponding graph \(G\) representing these flights.

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 dest(i) to origin(j) (assuming that the time constraints are satisfied).

Example 18.1.2. As a concrete example, consider the flights:
2. Washington (depart 8 A.M.) - Los Angeles (arrive 11 A.M.)
3. 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.

18.1.1. 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?

18.1.2. 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 ∈ VJ. 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., ℓ(e_i) = 1 and c(e_i) = 1).
- If the same plane can perform flight i and j (i.e., (i, j) ∈ 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.

Let J denote the resulting graph. See Figure 18.2 for an example.

Lemma 18.1.3. There is a way to perform all flights of F using at most k planes if and only if there is a feasible circulation in the network J.

Proof: Assume there is a way to perform the flights using k’ ≤ 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 is 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.

### 18.2. 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 18.3.
The input is a bitmap on a grid where every grid node represents a pixel. We covert 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 18.2.1.** Given input as above, partition \( V \) (the set of pixels) into two disjoint subsets \( F \) and \( B \), such that

\[
q(F, B) = \sum_{i \in F} f_i + \sum_{j \in B} b_j - \sum_{(i,j) \in E \cap \{|i,j| = 1\}} p_{ij}.
\]

is maximized.

We can rewrite \( q(F, B) \) as:

\[
q(F, B) = \sum_{i \in F} f_i + \sum_{j \in B} b_j - \sum_{(i,j) \in E \cap \{|i,j| = 1\}} p_{ij} = \sum_{i \in V} (f_i + b_i) - \left( \sum_{i \in F} f_i + \sum_{j \in E} b_j + \sum_{(i,j) \in E \cap \{|i,j| = 1\}} p_{ij} \right).
\]

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

\[
u(F, B) = \sum_{i \in B} f_i + \sum_{j \in F} b_j + \sum_{(i,j) \in E \cap \{|i,j| = 1\}} p_{ij}. \tag{18.1}
\]

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. (18.1) 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 corresponds 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 $e'_j = (j, t)$ with capacity $c(e'_j) = b_j$. 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 18.2.2.** A minimum cut $(F, B)$ in $J$ minimizes $u(F, B)$.

Using the minimum-cut max-flow theorem, we have:

**Theorem 18.2.3.** One can solve the segmentation problem, in polynomial time, by computing the max flow in the graph $J$.

### 18.3. 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 18.3.1.** A set $X \subseteq P$ 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 18.3.2 (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.

### 18.3.1. 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 edge $e_i = (s, i)$ 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 18.3.3.** 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 $t$. 

113
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

$$\gamma = \sum_{i \in X \text{ and } p_i > 0} p_i = \sum_{i \in F \text{ and } p_i > 0} p_i - \sum_{i \in X \text{ and } p_i > 0} p_i = C - \sum_{i \in X \text{ and } p_i > 0} p_i,$$

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 18.3.4.** 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 18.3.5.** 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.

### 18.4. 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 18.4.1.** 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.\(^5\))

**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.

\(^5\) nada = nothing.
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.

### 18.4.1. Problem definition

**Problem 18.4.2.** 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?

### 18.4.2. 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 \( \alpha = \sum_{x \neq z, \; x < y} g_{xy} \) (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 \( \alpha \) 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

\[
\alpha = \sum_{(x, y) \in S'} g_{xy}.
\]

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 \( \alpha \) 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 \( \alpha \). This implies the following result.

**Theorem 18.4.3.** Team \( z \) has been eliminated if and only if the maximum flow in \( G \) has value strictly smaller than \( \alpha \). Thus, we can test in polynomial time if \( z \) has been eliminated.
18.4.3. A compact proof of a team being eliminated

Interestingly, once \( z \) is eliminated, we can generate a compact proof of this fact.

**Theorem 18.4.4.** Suppose that team \( z \) has been eliminated. Then there exists a “proof” of this fact of the following form:

1. The team \( z \) can finish with at most \( m \) wins.
2. There is a set of teams \( \tilde{S} \subset S \) so that \( \sum_{x \in \tilde{S}} w_x + \sum_{(x,y) \in \tilde{S}} g_{xy} > m |\tilde{S}| \).

(And hence one of the teams in \( \tilde{S} \) must end with strictly more than \( m \) wins.)

**Proof:** If \( z \) is eliminated then the max flow in \( G \) has value \( \gamma \), which is smaller than \( \alpha \), see Eq. (18.2). By the minimum-cut max-flow theorem, there exists a minimum cut \((S,T)\) of capacity \( \gamma \) in \( G \), and let \( \tilde{S} = \{ x \mid v_x \in S \} \).

**Claim 18.4.5.** 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 \( \tilde{S} \).

**Proof:** \( x \notin \tilde{S} \) or \( y \notin \tilde{S} \) \( \implies u_{xy} \notin S \) : If \( x \) is not in \( \tilde{S} \) then \( v_x \) is in \( T \). But then, if \( u_{xy} \) is in \( S \) the edge \((u_{xy},v_y)\) 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 \( \alpha \)). As such, in such a case \( u_{xy} \) must be in \( T \). This implies that if either \( x \) or \( y \) are not in \( \tilde{S} \) then it must be that \( u_{xy} \in \tilde{S} \). (And as such \( u_{xy} \notin \tilde{S} \).)

\( x \in \tilde{S} \) and \( y \in \tilde{S} \) \( \implies u_{xy} \in S \) : Assume that both \( x \) and \( y \) are in \( \tilde{S} \), 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 \( x \in \tilde{S} \), and (ii) \((s,u_{xy})\) where at least one of \( x \) or \( y \) is not in \( \tilde{S} \). As such, the capacity of the cut \((S,T)\) is

\[ c(S,T) = \sum_{x \in \tilde{S}} (m - w_x) + \sum_{(x,y) \notin \tilde{S}} g_{xy} = m |\tilde{S}| - \sum_{x \in \tilde{S}} w_x + \left( \alpha - \sum_{(x,y) \in \tilde{S}} g_{xy} \right). \]

However, \( c(S,T) = \gamma < \alpha \), and it follows that

\[ m |\tilde{S}| - \sum_{x \in \tilde{S}} w_x - \sum_{(x,y) \notin \tilde{S}} g_{xy} < \alpha - \alpha = 0. \]

Namely, \( \sum_{x \in \tilde{S}} w_x + \sum_{(x,y) \notin \tilde{S}} g_{xy} > m |\tilde{S}| \), as claimed.

\[ \blacksquare \]
Chapter 19

Network Flow V - Min-cost flow

19.1. Minimum Average Cost Cycle

Let \( G = (V, E) \) be a digraph (i.e., a directed graph) with \( n \) vertices and \( m \) edges, and \( \omega : E \to \mathbb{R} \) be a weight function on the edges. A directed cycle is a closed walk \( C = (v_0, v_1, \ldots, v_t) \), where \( v_t = v_0 \) and \( (v_i, v_{i+1}) \in E \), for \( i = 0, \ldots, t-1 \). The average cost of a directed cycle is \( \text{AvgCost}(C) = \omega(C)/t = (\sum_{e \in C} \omega(e))/t \).

For each \( k = 0, 1, \ldots, \) 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 \( i = 0, \ldots, n \) 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 19.1.1.** 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, \( \alpha \) 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.

**MinAvgCostCycle(G) = 0 \implies \alpha \geq 0:** We can rewrite \( \alpha \) 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 \( \alpha \) 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 \( \pi \) of length \( n - k \) and a path \( \sigma \) 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 19.1: Decomposing \( P_n \) into a path \( \sigma \) and a cycle \( \pi \).
since \( \omega(\pi) \geq 0 \) as \( \pi \) 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 \iff \alpha \leq 0:** Let \( C = (v_0, v_1, \ldots, v_l) \) 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 \( \xi \) 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 \( \tau \) denote this walk (i.e., \( |\tau| = n - r \)). We have that

\[
d_n(w) \leq \omega(\xi \parallel \tau) = d_i(v_0) + \omega(\tau),
\]

where \( \xi \parallel \tau \) denotes the path formed by concatenating the path \( \tau \) to \( \xi \).

Similarly, let \( \rho \) be the walk formed by walking on \( C \) from \( w \) all the way back to \( v_0 \). Note that \( \xi \parallel \rho \) goes around \( C \) several times, and as such, \( \omega(\xi \parallel \rho) = 0 \), 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 \( \rho \) to it, we have that

\[
d_k(w) + \omega(\rho) \geq d_k(v_0) \geq d_i(v_0) \geq d_n(w) - \omega(\tau),
\]

by Eq. (19.1). Rearranging, we have that \( \omega(\rho) \geq d_n(w) - \omega(\tau) - d_k(w) \). Namely, we have

\[
\forall k \quad 0 = \omega(\xi \parallel \rho) = \omega(\rho) + \omega(\tau) \geq (d_n(w) - \omega(\tau) - d_k(w)) + \omega(\tau) = d_n(w) - d_k(w).
\]

\[
\implies \forall k \quad \frac{d_n(w) - d_k(w)}{n-k} \leq 0
\]

\[
\implies \beta(w) = \max_{k=0}^{n-1} {\frac{d_n(w) - d_k(w)}{n-k}} \leq 0.
\]

As such, \( \alpha = \min_{v \in V(G)} \beta(v) \leq \beta(w) \leq 0 \), and we conclude that \( \alpha = 0 \). \( \blacksquare \)

Finding the minimum average cost cycle is now not too hard. We compute the vertex \( v \) that realizes \( \alpha \) in Theorem 19.1.1. 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 \( 0 = \alpha = \min_{v \in V} \max_{k=0}^{n-1} \frac{d_n(w) - d_k(w)}{n-k} \). Namely, we have that for the vertex \( v \) it holds

\[
\max_{k=0}^{n-1} \frac{d_n(v) - d_k(v)}{n-k} = 0 \quad \implies \forall k \in \{0, \ldots, n-1\} \quad \frac{d_n(v) - d_k(v)}{n-k} \leq 0.
\]

This implies that \( d_n(v) \leq d_i(v) \), for all \( i \). Now, we repeat the proof of Theorem 19.1.1. Let \( P_n \) be the path with \( n \) edges realizing \( d_n(v) \). We decompose it into a path \( \pi \) of length \( k \) and a cycle \( \tau \). We know that \( \omega(\tau) \geq 0 \) (since all cycles have non-negative weights now). Now, \( \omega(\pi) \geq d_k(v) \). As such, \( \omega(\pi) = d_n(v) - \omega(\tau) \leq d_n(v) - d_k(v) \leq 0 \), since \( \tau \) is a path of length \( k \) ending at \( v \), and its cost is \( \geq d_k(v) \). Namely, the cycle \( \tau \) 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 \( \alpha \), 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 19.1.2.** Given a directed 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.
19.2. 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\(^3\) 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 \(d_w(s,t)\) 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 19.2.1. (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 \(\pi\) realizing \(d_\ell(s,t)\) also realizes \(d_w(s,t)\).

(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 \(p(v)\) 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 \(\pi\) 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 \(d_\ell(s,t) = d_w(s,t) + p(s) - p(t)\). 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.  

19.3. 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 \(\phi\), 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 \(\phi\).

It would be easier to look on the problem of minimum-cost circulation problem. Here, we are given instead of \(\phi\) 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

\[^3\text{http://en.wikipedia.org/wiki/Bellman-Ford_algorithm}\]
Assumption 19.3.1. To simplify the exposition, we will assume that if capacity of the edges.

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 19.3.1.** 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 19.3.2.** 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 $h(u \to v) > 0$ 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 $h(u \to v)$ is negative, then $h(v \to u) = -h(u \to v)$ 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 $h(u \to v) > 0$.

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 $h(u \to v) > 0$.

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 19.3.3. Let \( f \) be a circulation in a graph \( G \). Then, \( f \) can be decomposed into at most \( m \) cycles, \( C_1, \ldots, C_m \), such that, for any \( e \in E(G) \), we have

\[
  f(e) = \sum_{i=1}^{t} \lambda_i \cdot \chi_{C_i}(e),
\]

where \( \lambda_1, \ldots, \lambda_t > 0 \) 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 \( \lambda_1 \) 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 \( \lambda_1 \) and \( C_1 \) implies the claim.

Theorem 19.3.4. A flow \( f \) is a minimum cost feasible circulation if and only if each directed cycle of \( G_t \) has nonnegative cost.

Proof: Let \( C \) be a negative cost cycle in \( G_t \). Then, we can circulate more flow on \( C \) and get a flow with smaller price. In particular, let \( \epsilon > 0 \) be a sufficiently small constant, such that \( g = f + \epsilon \cdot \chi_C \) is still a feasible circulation (observe, that since the edges of \( C \) are \( G_t \), all of them have residual capacity that can be used to this end). Now, we have that

\[
  \text{cost}(g) = \text{cost}(f) + \sum_{e \in C} \kappa(e) \cdot \epsilon = \text{cost}(f) + \epsilon \cdot \sum_{e \in C} \kappa(e) = \text{cost}(f) + \epsilon \cdot \kappa(C) < \text{cost}(f),
\]

since \( \kappa(C) < 0 \), which is a contradiction to the minimality of \( f \).

As for the other direction, assume that all the cycles in \( G_t \) have non-negative cost. Then, let \( g \) be any feasible circulation. Consider the circulation \( h = g - f \). By Lemma 19.3.2, all the edges used by \( h \) are in \( G_t \), and by Lemma 19.3.3 we can find \( t \leq |E(G_t)| \) cycles \( C_1, \ldots, C_t \) in \( G_t \), and coefficients \( \lambda_1, \ldots, \lambda_t \), 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 \( G_t \). This implies that \( \text{cost}(g) \geq \text{cost}(f) \). Namely, \( f \) is a minimum-cost circulation.

19.4. 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 \( G_t \) (using the algorithm of Section 19.1). If the cost of \( C \) is non-negative, we are done since we had arrived to the minimum cost circulation, by Theorem 19.3.4.

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 19.1.2, 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.
19.5. 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.

**Lemma 19.5.1.** Let $f$ be a flow, and let $g$ the flow resulting from applying the cycle $C = C_t$ to it. Then, $\mu(g) \geq \mu(f)$.

**Proof:** Assume for the sake of contradiction, that $\mu(g) < \mu(f)$. Namely, we have

$$\frac{\kappa(C_g)}{|C_g|} < \frac{\kappa(C_f)}{|C_f|}. \quad (19.2)$$

Now, the only difference between $G_f$ and $G_g$ are the edges of $C_f$. In particular, some edges of $C_t$ 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 $C_t$ will be present in $G_g$.

Now, $C_g$ must use at least one of the new edges of $G_g$, 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 $\pi = C_t \cup C_g$. We have that the average of $\pi$ is

$$\alpha = \frac{\kappa(C_t) + \kappa(C_g)}{|C_t| + |C_g|} < \max \left( \frac{\kappa(C_g)}{|C_g|}, \frac{\kappa(C_t)}{|C_t|} \right) = \mu(f),$$

by Eq. (19.2). We can write $\pi$ is a union of $k$ edge-disjoint cycles $\sigma_1, \ldots, \sigma_k$ 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 $\sigma_1, \ldots, \sigma_k$ have no edges in $U$, it follows that they are all contained in $G_t$. We have

$$\kappa(C_t) + \kappa(C_g) = \sum_i \kappa(\sigma_i) + 0.$$

Thus, there is some non-negative integer constant $c$, such that

$$\alpha = \frac{\kappa(C_t) + \kappa(C_g)}{|C_t| + |C_g|} = \frac{\sum_i \kappa(\sigma_i)}{c + \sum_i |\sigma_i|} \geq \frac{\sum_i \kappa(\sigma_i)}{\sum_i |\sigma_i|},$$

since $\alpha$ is negative (since $\alpha < \mu(f) < 0$ as otherwise the algorithm would had already terminated). Namely, $\mu(f) > (\sum_i \kappa(\sigma_i))/(|\sum_i |\sigma_i|)$. Which implies that there is a cycle $\sigma_r$, such that $\mu(f) > \kappa(\sigma_r)/|\sigma_r|$ and this cycle is contained in $G_t$. But this is a contradiction to the minimality of $\mu(f)$.

19.5.1. 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 \( \alpha \), 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 \( \beta \). 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 19.2.1, and as such
\[
d(v) - d(u) \leq \beta(u \rightarrow v) = \kappa(u \rightarrow v) - \mu(f).
\] (19.3)

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. (19.3) implies that
\[
\forall (u,v) \in E(G_i) \quad \psi(u \rightarrow v) = \kappa(u \rightarrow v) + d(u) - d(v) \geq \mu(f).
\] (19.4)

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.

### 19.5.2. Bounding the number of iterations

**Lemma 19.5.2.** 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, \( \mu(g) \geq (1-1/m)\mu(f) \).

**Proof:** Let \( C_0, \ldots, C_{m-1} \) 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 \( G_k \) has no negative reduced cost edge, and as such \( \mu(g) \geq 0 \). 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
\[
\kappa(C_h) = \psi(C_h) = \sum_{e \in C_h} \psi(e) = \psi(e') + \sum_{e \in C_h, e \neq e'} \psi(e) \geq 0 + (|C_h| - 1)\mu(f),
\]
by Eq. (19.4). Namely, the average cost of \( C_h \) is
\[
0 > \mu(f_h) = \frac{\kappa(C_h)}{|C_h|} \geq \frac{|C_h| - 1}{|C_h|} \mu(f) \geq \left(1 - \frac{1}{n}\right)\mu(f).
\]

The claim now easily follows from Lemma 19.5.1.

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 19.5.3.** 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 19.5.4. Let \( f \) be the circulation maintained by the algorithm at iteration \( \rho \). Then there exists an edge \( e \) in the residual network \( G_\ell \) such that it never appears in the residual networks of circulations maintained by the algorithm, for iterations larger than \( \rho + t \), where \( t = 2nm/\ln n \).

Proof: Let \( g \) 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 \( g \). Namely,
\[
\psi(u \to v) = \kappa(u \to v) + d(u) - d(v),
\]
where \( d(u) \) is the length of the shortest walk ending at \( u \) where the weight of edge \( (u, w) \) is \( \kappa(u \to w) - \mu(g) \).

Now, conceptually, we are running the algorithm using this reduced cost function over the edges, and consider the minimum average cost cycle at iteration \( \rho \) with cost \( \alpha = \mu(f) \). There must be an edge \( e \in E(G_\ell) \), such that \( \psi(e) \leq \alpha \). (Note, that \( \alpha \) is a negative quantity, as otherwise the algorithm would have terminated at iteration \( \rho \).

We have that, at iteration \( \rho + t \), it holds
\[
\mu(g) \geq \alpha \cdot \left(1 - \frac{1}{n}\right)^t \geq \alpha \cdot \exp(-2m/\ln n) \geq \frac{\alpha}{2n},
\]
by Lemma 19.5.2 and Observation 19.5.3 and since \( \alpha < 0 \). On the other hand, by Eq. (19.4), we know that for all the edges \( f \) in \( E(G_\ell) \), it holds \( \psi(f) \geq \mu(g) \geq \alpha/2n \). As such, \( e \) can not be an edge of \( G_g \) since \( \psi(e) \leq \alpha \). Namely, it must be that \( g(e) = c(e) \).

So, assume that at a later iteration, say \( \rho + t + \tau \), the edge \( e \) reappeared in the residual graph. Let \( h \) be the flow at the \( (\rho + t + \tau) \)th iteration, and let \( G_h \) be the residual graph. It must be that \( h(e) < c(e) = g(e) \). Now, consider the circulation \( i \equiv g - h \). It has a positive flow on the edge \( e \), since \( i(e) = g(e) - h(e) > 0 \). In particular, there is a directed cycle \( C \) of positive flow of \( i \) in \( G_i \) that includes \( e \), as implied by Lemma 19.3.3. Note, that Lemma 19.3.2 implies that \( C \) is also a cycle of \( G_h \).

Now, the edges of \( C^{-1} \) are present in \( G_g \). To see that, observe that for every edge \( g \in C \), we have that \( 0 < i(g) = g(g) - h(g) \leq g(g) - \ell(g) \). Namely, \( g(g) > \ell(g) \) and as such \( g^{-1} \in E(G_g) \). As such, by Eq. (19.4), we have \( \psi(g^{-1}) \geq \mu(g) \).

This implies
\[
\forall g \in C \quad \psi(g) = -\psi(g^{-1}) \leq -\mu(g) \leq -\frac{\alpha}{2n},
\]
by Eq. (19.5). 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 19.5.1. 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 19.1.2. Putting everything together, we get the following.

Theorem 19.5.5. 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 \( O(m^2 n^3 \log n) \) time.

19.6. Bibliographical Notes

The minimum average cost cycle algorithm, of Section 19.1, 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 20

Network Flow VI - Min-Cost Flow Applications

20.1. 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 $\alpha$ be the value of this flow. Next, we compute the min-cost flow in this network from $s$ to $t$ with flow $\alpha$, 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 20.1.1.** 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.)

20.2. 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\(^\dagger\). 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.

\(^\dagger\)Verify that you know to do this — its a natural question for the exam.
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.

**Theorem 20.2.1.** 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.

### 20.3. 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.

### 20.4. Covering by Cycles

Given a directed 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 20.4.1.** 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.

### 20.5. 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 $\phi$ of the maximum matching in $G$. Next, we compute the min-cost flow in $G$ with this value $\phi$, where the edges connected to the source or the sing 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 20.5.1.** Given a bipartite graph $G$ and costs on the edges, one can compute the maximum cardinality minimum cost matching in polynomial time.
20.6. The transportation problem

In the transportation problem, we are given \( m \) facilities \( f_1, \ldots, f_m \). The facility \( f_i \) contains \( x_i \) units of some commodity, for \( i = 1, \ldots, m \). Similarly, there are \( u_1, \ldots, u_n \) customers that would like to buy this commodity. In particular, \( u_i \) would like to buy \( d_i \) units, for \( i = 1, \ldots, n \). 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 \( f_1, \ldots, f_m \) on one side, and \( u_1, \ldots, u_n \) on the other side. There is an edge from \((f_i, u_j)\) with costs \( c_{ij} \), for \( i = 1, \ldots, m \) and \( j = 1, \ldots, n \). Next, we create a source vertex that is connected to \( f_i \) with capacity \( x_i \), for \( i = 1, \ldots, m \). Similarly, we create an edge from \( u_j \) to the sink \( t \), with capacity \( d_i \), for \( j = 1, \ldots, n \). We compute the min-cost flow in this network that pushes \( \phi = \sum_j d_k \) units from the source to the sink. Clearly, the solution encodes the required optimal solution to the transportation problem.

Theorem 20.6.1. The transportation problem can be solved in polynomial time.

Chapter 21

Exercises - Network Flow

This chapter include problems that are related to network flow.

21.1. Network Flow

21.1.1. 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上游 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下游 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中枢 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.
21.1.2. 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 \( \delta \), 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 \( \delta \).)

   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 \( p_1 \geq p_2 \geq \cdots \geq p_k \), so that if the back-up set for \( v \) consists of devices at distances \( d_1 \leq d_2 \leq \cdots \leq d_k \), 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.

21.1.3. 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?).

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 $f^{\text{in}}(v) - f^{\text{out}}(v) = d_v$ 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.

21.1.5. 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.

21.1.6. 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 $x_1, y_1, x_2, y_2, \ldots$ 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 $x_1, y_1, x_2, y_2, \ldots$ 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.

### 21.1.7. 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.

### 21.1.8. 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.

### 21.1.9. 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 ot prescribe how this algorithms 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.

### 21.1.10. 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.

21.1.11. 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 $k = 1, \ldots, n$. 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.

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.

2. (8 pts.) Give a polynomial-time algorithm that determines whether a matrix $M$ with 0-1 entries is rearrangeable.

21.1.12. 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).

21.1.13. Transitivity

(10 pts.)

Given a graph $G = (V, E)$, and a natural number $k$, we can define a relation $\rightarrow_{G,k} \rightarrow$ on pairs of vertices of $G$ as follows. If $x, y \in V$, we say that $x \rightarrow_{G,k} \rightarrow y$ 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.


(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.

<table>
<thead>
<tr>
<th>Country</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>grown-up men</td>
<td>11.998</td>
<td>9.083</td>
<td>2.919</td>
<td>24.000</td>
</tr>
<tr>
<td>grown-up women</td>
<td>12.983</td>
<td>10.872</td>
<td>3.145</td>
<td>27.000</td>
</tr>
<tr>
<td>children</td>
<td>1.019</td>
<td>2.045</td>
<td>0.936</td>
<td>4.000</td>
</tr>
<tr>
<td>total</td>
<td>26.000</td>
<td>22.000</td>
<td>7.000</td>
<td>55.000</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Country</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>grown-up men</td>
<td>11.000</td>
<td>10.000</td>
<td>3.000</td>
<td>24.000</td>
</tr>
<tr>
<td>grown-up women</td>
<td>13.000</td>
<td>10.000</td>
<td>4.000</td>
<td>27.000</td>
</tr>
<tr>
<td>children</td>
<td>1.000</td>
<td>2.000</td>
<td>0.000</td>
<td>4.000</td>
</tr>
<tr>
<td>total</td>
<td>26.000</td>
<td>22.000</td>
<td>7.000</td>
<td>55.000</td>
</tr>
</tbody>
</table>

1. (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.

2. (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 $\alpha$ to either $\lfloor \alpha \rfloor$ or $\lceil \alpha \rceil$. Use a flow computation to check if the desired rounding is possible.

3. (10 pts.) Prove that the rounding we are looking for in (a) and (b) always exists.

21.1.15. 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.

21.1.16. 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 $G$.

```plaintext
MAX-FLOW-BY-SCALING(G, s, t)
1 C ← max_{(u,v) \in E} c(u, v)
2 initialize flow f to 0
3 K ← 2[lg C]
4 while K ≥ 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 ← 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.

21.1.17. Perfect Matching
(20 pts.)

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

$$N(X) = \{ y \in V \mid (x, y) \in E \text{ for some } x \in X \},$$

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 $|L|$.

21.1.18. Number of augmenting paths

1. (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.]

2. (10 pts.) Suppose that a flow network $G = (V, E)$ has symmetric edges, that is, $(u, v) \in E$ if and only $(v, u) \in E$. 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.]

21.1.19. Minimum Cut Festival

(20 pts.)

1. 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.

2. For any $\alpha \geq 1$, define an $\alpha$ approximate cut in a multigraph $G$ as any cut whose cardinality is within a multiplicative factor $\alpha$ 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 $\alpha$-approximate cut in $G$.

3. 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 $G$.

4. Formulate and prove a similar result of the number of $\alpha$-approximate cuts in a multigraph $G$. 

133
21.1.20. 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.


(10 pts.)

Let $f$ be a flow in a network, and let $\alpha$ be a real number. The scalar flow product, denoted by $\alpha f$, 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 $\alpha$ in the range $0 \leq \alpha \leq 1$.

21.1.22. 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.

21.1.23. 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 $O(\sqrt{V}E)$ 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.

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 $P_1, P_2, ..., P_k$ are vertex-disjoint augmenting paths with respect to $M$, then the symmetric difference $M \oplus (P_1 \cup P_2 \cup ... \cup P_k)$ 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 \( \mathcal{P} \leftarrow \{P_1, P_2, ..., P_k\} \) be a maximum set of vertex-disjoint shortest augmenting paths with respect to \( M \)
4. \( M \leftarrow M \oplus (P_1 \cup P_2 \cup \ldots \cup P_k) \)
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 \( P_1, P_2, ..., P_k \) be a maximum set of vertex-disjoint augmenting paths of length \( l \) with respect to \( M \). Let \( M' = M \oplus (P_1 \cup P_2 \cup \ldots \cup P_k) \), and suppose that \( P \) is a shortest augmenting path with respect to \( M' \).

3. (2 pts.) Show that if \( P \) is vertex-disjoint from \( P_1, P_2, ..., P_k \), then \( P \) has more than \( l \) edges.

4. (2 pts.) Now suppose \( P \) is not vertex-disjoint from \( P_1, P_2, ..., P_k \). Let \( \mathcal{A} \) be the set of edges \((M \oplus M^*) \oplus P\). Show that \( |\mathcal{A}| \geq (k+1)l \). 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 \( P_1, P_2, ..., P_k \) for a given matching \( M \). Conclude that the total running time of Hopcroft-Karp is \( O(\sqrt{VE}) \).

### 21.2. Min Cost Flow

#### 21.2.1. 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.

(A) (10 pts.) Provide a polynomial time algorithm that computes the largest number of paths that can be streamed from the server.

(B) (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.
21.2.2. 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 $\alpha$ units of material on an edge $e$ of length $\ell$ 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?

21.2.3. 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.

21.2.4. Perfect Matching

(20 pts.)

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 $|L|$.

21.2.5. Max flow by augmenting

1. (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.]

2. (10 pts.) Suppose that a flow network $G = (V, E)$ has symmetric edges, that is, $(u, v) \in E$ if and only $(v, u) \in E$. 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.]

21.2.6. And now for something completely different.

(10 pts.)

Prove that the following problems are NPC or provide a polynomial time algorithm to solve them:

1. Given a directly 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 directly 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 $v$).

21.2.7. 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 compares with the randomized algorithm shown in class?
Chapter 22

Linear Programming

22.1. Introduction and Motivation

In the VCR/guns/nuclear-bombs/napkins/star-wars/professors/butter/mice problem, the benevolent dictator, Biga Pigui-nus, 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:

\[ x_{\text{gun}} + 1000 \times x_{\text{nuclear-bomb}} \geq 1000, \]

where \(x_{\text{gun}}\) is the number of guns constructed, and \(x_{\text{nuclear-bomb}}\) is the number of nuclear-bombs constructed. On the other hand,

\[ 100 \times x_{\text{gun}} + 1000000 \times x_{\text{nuclear-bomb}} \leq x_{\text{security}} \]

where \(x_{\text{security}}\) 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: \(x_1, \ldots, x_n\) 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 \(x_1, \ldots, x_n\) 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.

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.
22.1. 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 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\(^{\text{v}}\).

22.1.2. 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 on the right.

It is not too hard to write down min-cost network flow using linear programming.

22.2. The Simplex Algorithm

22.2.1. Linear program where all the variables are positive

\[
\begin{align*}
\text{max} & \quad \sum_{j=1}^{n} c_j x_j \\
\text{subject to} & \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \\
& \quad \text{for } i = 1, 2, \ldots, m.
\end{align*}
\]

We are given a LP, depicted on the left, 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:

\[
2x' - 2x'' + y' - y'' \geq 5, \quad x' \geq 0, \quad y' \geq 0, \quad x'' \geq 0 \quad \text{and} \quad y'' \geq 0.
\]

**Lemma 22.2.1.** 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 $I$.

22.2.2. Standard form

Using Lemma 22.2.1, we can now require a LP to be specified using only positive variables. This is known as standard form.

\(^{\text{v}}\)There were other economists that were born in Russia, but lived in the west that got the Nobel prize – Leonid Hurwicz for example.
A linear program in standard form.
\[
\begin{align*}
\text{max} & \quad \sum_{j=1}^{n} c_j x_j \\
\text{subject to} & \quad \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m \\
x_j & \geq 0 \quad \text{for } j = 1, \ldots, n.
\end{align*}
\]

Here the matrix notation rises, by setting
\[
c = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}, \quad A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2(n-1)} & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{(m-1)1} & a_{(m-1)2} & \ldots & a_{(m-1)(n-1)} & a_{(m-1)n} \\ a_{m1} & a_{m2} & \ldots & a_{m(n-1)} & a_{mn} \end{pmatrix},
\]
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.

22.2.3. Slack Form

We next rewrite the LP into \textit{slack form}. It is a more convenient form for describing the \textit{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 (\textit{slack variables}) rewriting the inequality
\[
\sum_{i=1}^{n} a_i x_i \leq b
\]
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 \textit{basic variables}. All the other variables on the right side are \textit{nonbasic variables} (original isn’t it?). A LP in this form is in \textit{slack form}.

The slack form is defined by a tuple \((N, B, A, b, c, v)\).

---

\(^{2}\)The word \textit{convenience} is used here in the most liberal interpretation possible.
Consider the following

Show that any linear program can be transformed into equivalent slack form.

Exercise 22.2.2. Show that any linear program can be transformed into equivalent slack form.

Example 22.2.3. Consider the following LP which is in slack form, and its translation into the tuple \((N, B, A, b, c, v)\).

\[
\begin{align*}
\text{max} & \quad z = v + \sum_{j \in N} c_j x_j, \\
\text{s.t.} & \quad x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for} \quad i \in B, \\
& \quad x_i \geq 0, \quad \forall i = 1, \ldots, n + m.
\end{align*}
\]

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.

22.2.4. 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.

\[
\begin{align*}
\text{max} & \quad 5x_1 + 4x_2 + 3x_3 \\
\text{s.t.} & \quad 2x_1 + 3x_2 + x_3 \leq 5, \\
& \quad 4x_1 + x_2 + 2x_3 \leq 11, \\
& \quad 3x_1 + 4x_2 + 2x_3 \leq 8, \\
& \quad x_1, x_2, x_3 \geq 0.
\end{align*}
\]

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

\[
\begin{align*}
\text{max} & \quad z = 5x_1 + 4x_2 + 3x_3 \\
\text{s.t.} & \quad w_1 = 5 - 2x_1 - 3x_2 - x_3, \\
& \quad w_2 = 11 - 4x_1 - x_2 - 2x_3, \\
& \quad w_3 = 8 - 3x_1 - 4x_2 - 2x_3, \\
& \quad x_1, x_2, x_3, w_1, w_2, w_3 \geq 0.
\end{align*}
\]

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 \(x_1\) the value of the objective function would go up, since the coefficient of \(x_1\) for \(z\) is strictly larger than zero. Clearly, if we change the value of one non-basic variable a bit, all the basic variables would still comply with all these conditions. Namely, \(x_1 = 2.5\). Putting it into the system, we now have a solution which is

\[
\begin{align*}
    w_1 &= 5 - 2x_1 - 3x_2 - x_3 = 5 - 2x_1 \\
    w_2 &= 11 - 4x_1 - x_2 - 2x_3 = 11 - 4x_1 \\
    w_3 &= 8 - 3x_1 - 4x_2 - 2x_3 = 8 - 3x_1.
\end{align*}
\]

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

\[
\begin{align*}
    w_1 &= 5 - 2x_1 \geq 0, \\
    w_2 &= 11 - 4x_1 \geq 0, \\
    \text{and } w_3 &= 8 - 3x_1 \geq 0.
\end{align*}
\]

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

\[
x_1 = 2.5, \quad x_2 = 0, \quad x_3 = 0, \quad w_1 = 0, \quad w_2 = 1, \quad w_3 = 0.5 \quad \Rightarrow \quad z = 5x_1 + 4x_2 + 3x_3 = 12.5.
\]

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:

\[
x_1 = 2.5 - 0.5w_1 - 1.5x_2 - 0.5x_3. \tag{22.1}
\]

The problem is that \(x_1\) still appears in the right size 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. (22.1). Collecting similar terms, we get the following equivalent LP:

\[
\begin{align*}
\max \quad & z = 12.5 - 2.5w_1 - 3.5x_2 + 0.5x_3 \\
    x_1 &= 2.5 - 0.5w_1 - 1.5x_2 - 0.5x_3 \\
    w_2 &= 1 + 2w_1 + 5x_2 \\
    w_3 &= 0.5 + 1.5w_1 + 0.5x_2 - 0.5x_3.
\end{align*}
\]

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 transfered 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.

\[
\begin{align*}
\text{max} \quad z &= 13 - w_1 - 3x_2 - w_3 \\
\text{s.t.} \quad x_1 &= 2 - 2w_1 - 2x_2 + w_3 \\
& \quad w_2 = 1 + 2w_1 + 5x_2 \\
& \quad x_3 = 1 + 3w_1 + x_2 - 2w_3
\end{align*}
\]

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 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, an 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.

### 22.2.4.1. Starting somewhere

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.
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 22.2.4.** 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 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 23**

**Linear Programming II**

**23.1. The Simplex Algorithm in Detail**

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 $L$ 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 23.1. Namely, we first try to decrease $x_0$ as much as possible.
\[ \max \ z = v + \sum_{j \in N} c_j x_j, \]
\[ \text{s.t.} \ x_i = b_i - \sum_{j \in N} a_{ij} x_j \text{ for } i \in B, \]
\[ x_i \geq 0, \quad \forall i = 1, \ldots, n + m. \]

**23.2. 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 23.2. 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
\[ \left\{ j \mid c_j > 0, j \in N \right\}. \]

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 \( \tau \)) 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 variable) till \( x_l \) (the leaving variable) becomes zero.

Setting all nonbasic variables to zero, and letting \( x_e \) grow, implies that \( x_l = b_l - a_{le} x_e \), 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
\[ x_l = b_l - \sum_{j \in N} a_{ij} x_j \Rightarrow x_e = \frac{b_l}{a_{le}} - \sum_{j \in N \cup \{l\}} \frac{a_{jj}}{a_{le}} x_j, \text{ where } a_{ll} = 1. \]

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 the 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.

### 23.2.1. 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 23.6.

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.

### 23.2.2. Correctness of linear programming

**Definition 23.2.1.** 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 23.2.2 (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. ■

### 23.2.3. 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.
23.3. 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.

23.3.1. Duality by Example

Consider the linear program $L$ depicted on the right (Figure 23.3). Note, that any feasible solution, gives us a lower bound on the maximal value of the target function, denoted by $\eta$. 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:

$$2( x_1 + 4x_2 ) \leq 2(1)$$
$$+3(3x_1 - x_2 + x_3) \leq 3(3).$$

The resulting inequality is

$$11x_1 + 5x_2 + 3x_3 \leq 11. \quad (23.1)$$

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. (23.1) has larger coefficients that 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:

$$y_1(x_1 + 4x_2) \leq y_1(1)$$
$$+ y_2(3x_1 - x_2 + x_3) \leq y_2(3).$$

$$\frac{y_1(x_1 + 4x_2) + y_2(3x_1 - x_2 + x_3)}{y_1 + 3y_2} \leq \frac{y_1(1) + y_2(3)}{y_1 + 3y_2} \leq \frac{y_1 + y_2}{y_1 + 3y_2} \leq \frac{y_1 + 3y_2}{y_1 + 3y_2}. \quad (23.2)$$

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

$$4 \leq y_1 + 3y_2$$
$$1 \leq 4y_1 - y_2$$
$$3 \leq y_2,$$

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. (23.2).

Thus, if we want the best upper bound on $\eta$ (the maximal value of $z$) then we want to solve the LP $L$ depicted in Figure 23.4. This is the dual program to $L$ and its optimal solution is an upper bound to the optimal solution for $L$.

Figure 23.3: The linear program $L$.

Figure 23.4: The dual LP $L$. The primal LP is depicted in Figure 23.3.
23.3.2. The Dual Problem

Given a linear programming problem (i.e., primal problem, seen in Figure 23.5 (a), its associated dual linear program is in Figure 23.5 (b). The standard form of the dual LP is depicted in Figure 23.5 (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 23.6.

We just proved the following result.

**Lemma 23.3.1.** 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.

23.3.3. The Weak Duality Theorem

**Theorem 23.3.2.** If \((x_1, x_2, \ldots, x_m)\) is feasible for the primal LP and \((y_1, y_2, \ldots, y_m)\) 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

\[
\sum_j c_j x_j \leq \sum_j \left( \sum_{i=1}^m a_{ij} y_i \right) x_j \leq \sum_i \left( \sum_j a_{ij} x_j \right) y_i \leq \sum_i b_i y_i .
\]

We just proved the following result.

**Lemma 23.3.1.** 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.
Interestingly, if we apply the weak duality theorem on the dual program (namely, Figure 23.6 (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.

23.4. The strong duality theorem

The strong duality theorem states the following.

**Theorem 23.4.1.** If the primal LP problem has an optimal solution \( x^* = (x_1^*, \ldots, x_n^*) \) then the dual also has an optimal solution, \( y^* = (y_1^*, \ldots, y_m^*) \), 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 of they are feasible. We omit the tedious proof.

23.5. Some duality examples

23.5.1. 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.

\[
\begin{align*}
\text{max} & \quad d_t \\
\text{s.t.} & \quad d_s \leq 0 \\
& \quad d_u + \omega(u, v) \geq d_v \quad \forall (u, v) \in E, \\
& \quad d_x \geq 0 \quad \forall x \in V.
\end{align*}
\]

Equivalently, we get

\[
\begin{align*}
\text{max} & \quad d_t \\
\text{s.t.} & \quad d_s \leq 0 \\
& \quad d_v - d_u \leq \omega(u, v) \quad \forall (u, v) \in E, \\
& \quad d_x \geq 0 \quad \forall x \in V.
\end{align*}
\]
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.

$$\min \sum_{(u,v) \in E} y_{uv} \omega(u,v)$$

s.t.

$$y_s - \sum_{(s,u) \in E} y_{su} \geq 0 \quad (\ast)$$

$$\sum_{(u,v) \in E} y_{ux} - \sum_{(s,v) \in E} y_{sv} \geq 0 \quad \forall x \in V \setminus \{s, t\} \quad (\ast \ast)$$

$$\sum_{(u,t) \in E} y_{ut} \geq 1 \quad (\ast \ast \ast)$$

$$y_{uv} \geq 0 \quad \forall (u, v) \in E,$$

$$y_s \geq 0.$$

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.

23.5.2. Set Cover and Packing

Consider an instance of Set Cover with $(S, \mathcal{F})$, where $S = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{F_1, \ldots, F_m\}$, where $F_i \subseteq S$. The natural LP to solve this problem is

$$\min \sum_{F_j \in \mathcal{F}} x_j$$

s.t.

$$\sum_{F_j \in \mathcal{F}, \ u_i \in F_j} x_j \geq 1 \quad \forall u_i \in S,$$

$$x_j \geq 0 \quad \forall F_j \in \mathcal{F}.$$

The dual LP is

$$\max \sum_{u_i \in S} y_i$$

s.t.

$$\sum_{u_i \in F_j} y_i \leq 1 \quad \forall F_j \in \mathcal{F},$$

$$y_i \geq 0 \quad \forall u_i \in S.$$

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.
23.5.3. 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.

$$\text{max} \sum_{(s,v) \in E} x_{s\to v}$$

$$x_{u\to v} \leq c(u \to v) \quad \forall (u, v) \in E$$

$$\sum_{(u,v) \in E} x_{u\to v} - \sum_{(v,w) \in E} x_{v\to w} \leq 0 \quad \forall v \in V \setminus \{s, t\}$$

$$- \sum_{(u,v) \in E} x_{u\to v} + \sum_{(v,w) \in E} x_{v\to w} \leq 0 \quad \forall v \in V \setminus \{s, t\}$$

$$0 \leq x_{u\to v} \quad \forall (u,v) \in E.$$

To perform the duality transform, we define a dual variable for each inequality. We get the following dual LP:

$$\text{max} \sum_{(s,v) \in E} x_{s\to v}$$

$$x_{u\to v} \leq c(u \to v) \quad \forall (u, v) \in E$$

$$\sum_{(u,v) \in E} x_{u\to v} - \sum_{(v,w) \in E} x_{v\to w} \leq 0 \quad \forall v \in V \setminus \{s, t\}$$

$$- \sum_{(u,v) \in E} x_{u\to v} + \sum_{(v,w) \in E} x_{v\to w} \leq 0 \quad \forall v \in V \setminus \{s, t\}$$

$$0 \leq x_{u\to v} \quad \forall (u,v) \in E.$$

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:

$$\text{min} \sum_{(u,v) \in E} c(u \to v) y_{u\to v}$$

$$1 \leq y_{s\to v} + y_v - y'_v \quad \forall (s, v) \in E$$

$$0 \leq y_{u\to v} + y_v - y'_v - y_u + y'_u \quad \forall (u, v) \in E(G \setminus \{s, t\})$$

$$0 \leq y_{v\to t} - y_v + y'_v \quad \forall (u,v) \in E$$

$$y_{u\to v} \geq 0 \quad \forall (u,v) \in E$$

$$y_v \geq 0 \quad \forall v \in V$$

$$y'_v \geq 0 \quad \forall v \in V$$

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

\footnote{We 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.}
the following modified LP:

\[
\begin{align*}
\min & \sum_{(u,v) \in E} c(u \rightarrow v) y_{u \rightarrow v} \\
1 & \leq y_{s \rightarrow v} + d_v \quad \forall (s, v) \in E \\
0 & \leq y_{u \rightarrow v} + d_v - d_u \quad \forall (u, v) \in E \setminus \{s, t\} \\
0 & \leq y_{v \rightarrow t} - d_v \quad \forall (v, t) \in E \\
y_{u \rightarrow v} & \geq 0 \quad \forall (u, v) \in E \\
d_s = 1, & \quad d_t = 0 
\end{align*}
\]

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:

\[
\begin{align*}
\min & \sum_{(u,v) \in E} c(u \rightarrow v) y_{u \rightarrow v} \\
0 & \leq y_{s \rightarrow v} + d_v - d_s \quad \forall (s, v) \in E \\
0 & \leq y_{u \rightarrow v} + d_v - d_u \quad \forall (u, v) \in E \setminus \{s, t\} \\
0 & \leq y_{v \rightarrow t} - d_v \quad \forall (v, t) \in E \\
y_{u \rightarrow v} & \geq 0 \quad \forall (u, v) \in E \\
d_s = 1, & \quad d_t = 0 
\end{align*}
\]

Which simplifies to the following LP:

\[
\begin{align*}
\min & \sum_{(u,v) \in E} c(u \rightarrow v) y_{u \rightarrow v} \\
d_u - d_v & \leq y_{u \rightarrow v} \quad \forall (u, v) \in E \\
y_{u \rightarrow v} & \geq 0 \quad \forall (u, v) \in E \\
d_s = 1, & \quad d_t = 0 
\end{align*}
\]

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. Let us first start with the other direction, given a cut \((S, T)\) with \(s \in S\) and \(t \in T\), observe that setting

\[
\begin{align*}
d_u = 1 & \quad \forall u \in S \\
d_u = 0 & \quad \forall u \in T \\
y_{u \rightarrow v} = 1 & \quad \forall (u, v) \in (S, T) \\
y_{u \rightarrow v} = 0 & \quad \forall (u, v) \in E \setminus (S, T)
\end{align*}
\]

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 \([d^*_u, d^*_v]\). 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 \( d_u - d_v \leq y_{u \to v} \) in the LP).

So, let \( X_{u \to v} \) be an indicator variable which is one if the edge is in the generated cut. We just argued that \( E[X_{u \to v}] = \Pr[X_{u \to v} = 1] \leq y^*_{(u,v)} \).
We thus have that the expected cost of this random cut is

\[
E \left[ \sum_{(u,v) \in E} X_{u \to v} c(u \to v) \right] = \sum_{(u,v) \in E} c(u \to v) E[X_{u \to v}] \leq \sum_{(u,v) \in E} c(u \to v) y^*_{u \to v} = \alpha^*.
\]

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 \( \alpha^* \), see Remark 23.5.2 below. However, we argued that \( \alpha^* \) is no larger than the cost of any cut. We conclude that \( \alpha^* \) 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., \( \alpha^* \)).
We just argued that \( \alpha^* \) is the cost of the min cut in the given network. As such, we proved the following.

**Lemma 23.5.1.** The Min-Cut Max-Flow Theorem follows from the strong duality Theorem for Linear Programming.

**Remark 23.5.2.** In the above, we used the following “trivial” but powerful argument. Assume you have a random variable \( Z \), and consider its expectation \( \mu = E[Z] \).
The expectation \( \mu \) 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.

### 23.6. Solving LPs without ever getting into a loop - symbolic perturbations

#### 23.6.1. The problem and the basic idea

Consider the following LP:

\[
\begin{align*}
\text{max} \quad & \quad z = v + \sum_{j \in N} c_j x_j, \\
\text{s.t.} \quad & \quad x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for} \ i = 1, \ldots, n, \\
& \quad x_i \geq 0, \quad \forall i = 1, \ldots, n + m.
\end{align*}
\]

(Here \( B = \{1, \ldots, n\} \) and \( N = \{n + 1, \ldots, n + m\} \).) 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 \( \varepsilon_i = \varepsilon^j \). The quantities \( \varepsilon_1, \ldots, \varepsilon_n \) are infinitesimals of different scales. We slightly perturb the above LP by adding them to each equation. We get the following modified LP:

\[
\begin{align*}
\text{max} \quad & \quad z = v + \sum_{j \in N} c_j x_j, \\
\text{s.t.} \quad & \quad x_i = \varepsilon_i + b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for} \ i = 1, \ldots, n, \\
& \quad x_i \geq 0, \quad \forall i = 1, \ldots, n + m.
\end{align*}
\]

153
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 $\varepsilon_1, \ldots, \varepsilon_n$ 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

$$\alpha = \alpha_0 + \alpha_1 \varepsilon_1 + \cdots + \alpha_n \varepsilon_n \quad \text{and} \quad \beta = \beta_0 + \beta_1 \varepsilon_1 + \cdots + \beta_n \varepsilon_n,$$

(23.3)

then $\alpha > \beta$ if and only if there is an index $i$ such that $\alpha_0 = \beta_0, \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}$ and $\alpha_i > \beta_i$. That is, $\alpha > \beta$ if the vector $(\alpha_0, \alpha_1, \ldots, \alpha_n)$ is lexicographically larger than $(\beta_0, \beta_1, \ldots, \beta_n)$.

Significantly, but not obviously at this stage, the simplex algorithm would never divide an $\varepsilon_i$ by an $\varepsilon_j$, 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. (23.3)) is now never zero. This implies immediately that the Simplex algorithm always makes progress, and it does terminates. We still need to address the two issues:

(A) How are the symbolic perturbations are updated at each iteration?

(B) Why the constants can never be zero?

### 23.6.2. Pivoting as a Gauss elimination step

Consider the LP equations

$$x_i + \sum_{j \in \mathbb{N}} a_{ij} x_j = b_i, \quad \text{for } i \in B,$$

where $B = \{1, \ldots, n\}$ and $N = \{n+1, \ldots, n+m\}$. We can write these equations down in matrix form

$$
\begin{array}{cccccc|c}
1 & 0 & \ldots & 0 & x_{n+1} & x_{n+2} & \ldots & x_j & \ldots & x_{n+m} & \text{const} \\
0 & 1 & \ldots & 0 & a_{1,n+1} & a_{1,n+2} & \ldots & a_{1,j} & \ldots & a_{1,n+m} & b_1 \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \ldots & 0 & a_{k,n+1} & a_{k,n+2} & \ldots & a_{k,j} & \vdots & a_{k,n+m} & b_k \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\
0 & \ldots & 0 & 1 & a_{n,n+1} & a_{n,n+2} & \ldots & a_{n,j} & \ldots & a_{n,n+m} & b_n \\
\end{array}
$$

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 $r$. Now, add $a_{i,j} r$ 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.
23.6.2.1. Back to the perturbation scheme

We now add a new matrix to the above representations on the right side, that keeps track of the $\varepsilon$s. This looks initially as follows.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$\ldots$</th>
<th>$x_n$</th>
<th>$x_{n+1}$</th>
<th>$\ldots$</th>
<th>$x_j$</th>
<th>$\ldots$</th>
<th>$x_{n+m}$</th>
<th>const</th>
<th>$\varepsilon_1$</th>
<th>$\varepsilon_2$</th>
<th>$\ldots$</th>
<th>$\varepsilon_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>$\ldots$</td>
<td>0</td>
<td>$a_{1,n+1}$</td>
<td>$\ldots$</td>
<td>$a_{1,j}$</td>
<td>$\ldots$</td>
<td>$a_{1,n+m}$</td>
<td>$b_1$</td>
<td>1</td>
<td>0</td>
<td>$\ldots$</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$\ldots$</td>
<td>0</td>
<td>$a_{2,n+1}$</td>
<td>$\ldots$</td>
<td>$a_{2,j}$</td>
<td>$\ldots$</td>
<td>$a_{2,n+m}$</td>
<td>$b_2$</td>
<td>0</td>
<td>1</td>
<td>$\ldots$</td>
<td>0</td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\ldots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\ldots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>$\ldots$</td>
<td>0</td>
<td>1</td>
<td>$\ldots$</td>
<td>0</td>
<td>$\ldots$</td>
<td>0</td>
<td>$a_{k,n+1}$</td>
<td>$\ldots$</td>
<td>$a_{k,j}$</td>
<td>$\ldots$</td>
<td>$a_{k,n+m}$</td>
</tr>
</tbody>
</table>

Now, we run the algorithm as described above, using the $\varepsilon$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 $\varepsilon$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.

23.6.2.2. 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 $x_i = a_0 + \sum \alpha_i \cdot \varepsilon_i$. We interpret this as $x_i = a_0$, by setting all the $\varepsilon$s to be zero.

Chapter 24

Approximation Algorithms using Linear Programming

24.1. 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, it required that every edge \( vu \in E \) is covered. Namely, we require that \( x_v \lor x_u \) to be \textbf{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, we would like to minimize the total cost of the vertices we pick for the cover; namely, we would like to minimize \( \sum_{v \in V} x_v c_v \). Putting it together, we get the following integer programming instance:

\[
\begin{align*}
\text{min} & \quad \sum_{v \in V} c_v x_v, \\
\text{such that} & \quad x_v \in \{0, 1\} \quad \forall v \in V \\
& \quad x_v + x_u \geq 1 \quad \forall vu \in E.
\end{align*}
\] (24.1)

Naturally, solving this integer programming efficiently is \textbf{NP-HARD}, so instead let us try to relax this optimization problem to be a \textbf{LP} (which we can solve efficiently, at least in practice\(^\ddagger\)). 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 \textbf{LP} is

\[
\begin{align*}
\text{min} & \quad \sum_{v \in V} c_v x_v, \\
\text{such that} & \quad 0 \leq x_v \quad \forall v \in V, \\
& \quad x_v \leq 1 \quad \forall v \in V, \\
& \quad x_v + x_u \geq 1 \quad \forall vu \in E.
\end{align*}
\] (24.2)

So, consider the optimal solution to this \textbf{LP}, assigning value \( \widehat{x}_v \) to the variable \( X_v \), for all \( v \in V \). As such, the optimal value of the \textbf{LP} solution is

\[
\widehat{\alpha} = \sum_{v \in V} c_v \widehat{x}_v.
\]

Similarly, let the optimal integer solution to integer program (\textbf{IP}) Eq. (24.1) denoted by \( x'_v \), for all \( v \in V \) and \( \alpha' \), respectively. Note, that any feasible solution for the \textbf{IP} of Eq. (24.1), is a feasible solution for the \textbf{LP} of Eq. (24.2). As such, we must have that

\[
\widehat{\alpha} \leq \alpha',
\]

where \( \alpha' \) 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 \( \widehat{x}_v \) 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 \textbf{rounding}.

To this end, it is beneficial to consider a vertex \( v \) and its fractional value \( \widehat{x}_v \). If \( \widehat{x}_v = 1 \) then we definitely want to put it into our solution. If \( \widehat{x}_v = 0 \) then the \textbf{LP} consider this vertex to be useless, and we really do not want to use it. Similarly, if \( \widehat{x}_v = 0.9 \), then the \textbf{LP} considers this vertex to be very useful (0.9 useful to be precise, whatever this “means”). Intuitively, since the \textbf{LP} puts its money where its belief is (i.e., \( \widehat{\alpha} \) value is a function of this “belief” generated by the \textbf{LP}), we should trust the \textbf{LP} values as a guidance to which vertices are useful and which are not. Which brings to forefront the following idea: Let’s pick all the vertices that are above a certain threshold of usefulness according to the \textbf{LP} solution. Formally, let

\[
S = \left\{ v \left| \widehat{x}_v \geq 1/2 \right. \right\}.
\]

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

\[
\widehat{x}_v + \widehat{x}_u \geq 1.
\]

\(^\ddagger\)And also in theory if the costs are integers, using more advanced algorithms than the \textbf{Simplex} algorithm.
Since 0 ≤ \( \hat{x}_v \) ≤ 1 and 0 ≤ \( \hat{x}_u \) ≤ 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

\[
c_S = \sum_{v \in S} c_v = \sum_{v \in S} 1 \cdot c_v \leq 2 \sum_{v \in V} \hat{x}_v \cdot c_v \leq 2 \sum_{v \in V} \hat{x}_v c_v = 2 \alpha \leq 2 \alpha^I,
\]

since \( \hat{x}_v \geq 1/2 \) as \( v \in S \).

Since \( \alpha^I \) is the cost of the optimal solution, we got the following result.

**Theorem 24.1.1.** 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.

### 24.2. 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 \( \mathcal{X} \subseteq F \) 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.

\[
\min \alpha = \sum_{U \in \mathcal{F}} x_U, \\
\text{s.t. } x_U \in \{0, 1\} \quad \forall U \in \mathcal{F}, \\
\sum_{U \in \mathcal{F}, s \in U} x_U \geq 1 \quad \forall s \in S.
\]

Next, we relax this IP into the following LP.

\[
\min \alpha = \sum_{U \in \mathcal{F}} x_U, \\
0 \leq x_U \leq 1 \quad \forall U \in \mathcal{F}, \\
\sum_{U \in \mathcal{F}, s \in U} x_U \geq 1 \quad \forall s \in S.
\]

As before, consider the optimal solution to the LP: \( \forall U \in \mathcal{F}, \hat{x}_U \), and \( \alpha \). Similarly, let the optimal solution to the IP (and thus for the problem) be: \( \forall U \in \mathcal{F}, x^I_{U} \), and \( \alpha^I \). As before, we would try to use the LP solution to guide us in the
As before, if \( \hat{x_U} \) is close to 1 then we should pick \( U \) to the cover and if \( \hat{x_U} \) is close to 0 we should not. As such, it's natural to pick \( U \in \mathcal{F} \) into the cover by randomly choosing it into the cover with probability \( \hat{x_U} \). 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
\[
\mathbb{E}\left[ \text{cost of } \mathcal{G} \right] = \mathbb{E}\left[ \sum_{S \in \mathcal{F}} Z_S \right] = \sum_{S \in \mathcal{F}} \mathbb{E}[Z_S] = \sum_{S \in \mathcal{F}} \Pr[S \in \mathcal{G}] = \sum_{S \in \mathcal{F}} \hat{x_S} = \hat{\alpha} \leq \alpha'.
\] (24.3)

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 \( m = \lceil \log n \rceil = O(\log n) \) 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
\[
\sum_{U \in \mathcal{F}, s \in U} \hat{x_U} \geq 1,
\] (24.4)
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
\[
\Pr[s \text{ not covered by } \mathcal{G}_i] = \prod_{U \in \mathcal{F}, s \in U} \Pr[U \text{ was not picked into } \mathcal{G}_i] = \prod_{U \in \mathcal{F}, s \in U} (1 - \hat{x_U}) \leq \exp(-\hat{x_U}) = \exp\left(- \sum_{U \in \mathcal{F}, s \in U} \hat{x_U} \right) \leq \exp(-1) \leq \frac{1}{2},
\]
by Eq. (24.4). As such, the probability that \( s \) is not covered in all \( m \) iterations is at most
\[
\left( \frac{1}{2} \right)^m \leq \frac{1}{n^{10}},
\]
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. (24.3), in each iteration the expected cost of the cover computed is at most the cost of the optimal solution (i.e., \( \alpha' \)). As such the expected cost of the solution computed is
\[
cost_{\mathcal{H}} \leq \sum_i \cost_{\mathcal{B}_i} \leq m\alpha' = O(\alpha' \log n).
\]

Putting everything together, we get the following result.

Theorem 24.2.1. 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.
24.3. Minimizing congestion

Let \( G \) be a graph with \( n \) vertices, and let \( \pi_i \) and \( \sigma_i \) be two paths with the same endpoints \( v_i, u_i \in V(G) \), for \( i = 1, \ldots, t \).

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 \( \pi_i \) or \( \sigma_i \). We would like to do it in such a way that no edge in the graph is being used too much.

Definition 24.3.1. 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:

\[
\begin{align*}
\text{min} & \quad w \\
\text{s.t.} & \quad x_i \geq 0, & i = 1, \ldots, t, \\
& \quad x_i \leq 1, & i = 1, \ldots, t, \\
& \quad \sum_{e \in \pi_i} x_i + \sum_{e \in \sigma_i} (1 - x_i) \leq w & & \forall e \in E.
\end{align*}
\]

Let \( \bar{x}_i \) be the value of \( x_i \) in the optimal solution of this LP, and let \( \bar{w} \) be the value of \( w \) in this solution. Clearly, the optimal congestion must be bigger than \( \bar{w} \).

Let \( X_i \) be a random variable which is one with probability \( \bar{x}_i \), and zero otherwise. If \( X_i = 1 \) then we use \( \pi \) to route from \( v_i \) to \( u_i \), otherwise we use \( \sigma_i \). Clearly, the congestion of \( \theta \) is

\[
Y_\theta = \sum_{e \in \pi_i} X_i + \sum_{e \in \sigma_i} (1 - X_i).
\]

And in expectation

\[
\alpha_\theta = E[Y_\theta] = E \left[ \sum_{e \in \pi_i} X_i + \sum_{e \in \sigma_i} (1 - X_i) \right] = \sum_{e \in \pi_i} E[X_i] + \sum_{e \in \sigma_i} E[1 - X_i]
= \sum_{e \in \pi_i} \bar{x}_i + \sum_{e \in \sigma_i} (1 - \bar{x}_i) \leq \bar{w}.
\]

Using the Chernoff inequality, we have that

\[
\Pr[Y_\theta \geq (1 + \delta)\alpha_\theta] \leq \exp \left( -\frac{\alpha_\theta \delta^2}{4} \right) \leq \exp \left( -\frac{\bar{w} \delta^2}{4} \right).
\]

(Note, that this works only if \( \delta < 2e - 1 \), see Theorem 12.2.7). Let \( \delta = \sqrt{\frac{400}{\bar{w}} \ln t} \). We have that

\[
\Pr[Y_\theta \geq (1 + \delta)\alpha_\theta] \leq \exp \left( -\frac{\delta^2 \bar{w}}{4} \right) \leq \frac{1}{t^{100}},
\]

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 \( (1 + \delta)\bar{w} \).

To see what this result means, let us play with the numbers. Let assume that \( t = n \), and \( \bar{w} \geq \sqrt{n} \). Then, the solution has congestion larger than the optimal solution by a factor of

\[
1 + \delta = 1 + \sqrt{\frac{20}{\bar{w}} \ln t} \leq 1 + \frac{\sqrt{20 \ln n}}{n^{1/4}},
\]

which is of course extremely close to 1, if \( n \) is sufficiently large.

**Theorem 24.3.2.** 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 \( \text{opt} \) is the congestion of the optimal solution, and \( \delta = \sqrt{\frac{20\ln t}{\bar{w}}} \).
When the congestion is low. Assume that \( \bar{w} \) is a constant. In this case, we can get a better bound by using the Chernoff inequality in its more general form, see Theorem 12.2.7. Indeed, set \( \delta = c \ln t / \ln \ln t \), where \( c \) is a constant. For \( \mu = \alpha_e \), we have that

\[
\Pr \left[ Y_{\delta} \geq (1 + \delta)\mu \right] \leq \left( \frac{e^{\delta}}{(1 + \delta)^{1+\delta}} \right)^{\mu} = \exp \left( \mu (\delta - (1 + \delta) \ln(1 + \delta)) \right) = \exp \left( -\mu c' \ln t \right)
\]

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.

Chapter 25

Exercises - Linear Programming

This chapter include problems that are related to linear programming.

25.1. Miscellaneous

25.1.1. 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?

25.2. Tedious

25.2.1. Tedious Computations

(20 pts.)

Provide detailed solutions for the following problems, showing each pivoting stage separately.

1. (5 pts.)

maximize \( 6x_1 + 8x_2 + 5x_3 + 9x_4 \)

subject to

\[
2x_1 + x_2 + x_3 + 3x_4 \leq 5 \\
x_1 + 3x_2 + x_3 + 2x_4 \leq 3 \\
x_1, x_2, x_3, x_4 \geq 0.
\]

2. (5 pts.)

maximize \( 2x_1 + x_2 \)

subject to
\[2x_1 + x_2 \leq 4\]
\[2x_1 + 3x_2 \leq 3\]
\[4x_1 + x_2 \leq 5\]
\[x_1 + 5x_2 \leq 1\]
\[x_1, x_2 \geq 0.\]

3. (5 pts.)
maximize \[6x_1 + 8x_2 + 5x_3 + 9x_4\]
subject to
\[x_1 + x_2 + x_3 + x_4 = 1\]
\[x_1, x_2, x_3, x_4 \geq 0.\]

4. (5 pts.)
minimize \[x_{12} + 8x_{13} + 9x_{14} + 2x_{23} + 7x_{24} + 3x_{34}\]
subject to
\[x_{12} + x_{13} + x_{14} \geq 1\]
\[-x_{12} + x_{23} + x_{24} = 0\]
\[-x_{13} - x_{23} + x_{34} = 0\]
\[x_{14} + x_{24} + x_{34} \leq 1\]
\[x_{12}, x_{13}, \ldots, x_{34} \geq 0.\]

### 25.2.2. Linear Programming for a Graph

1. (3 pts.) Given a weighted, directed graph \(G = (V, E)\), with weight function \(w : E \to \mathbb{R}\) 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, \(K_1, K_2, \ldots, K_k\), 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.

### 25.2.3. 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:

<table>
<thead>
<tr>
<th></th>
<th>Ithaca-Newark</th>
<th>Newark-Boston</th>
<th>Ithaca-Boston</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>300</td>
<td>160</td>
<td>360</td>
</tr>
<tr>
<td>B</td>
<td>220</td>
<td>130</td>
<td>280</td>
</tr>
<tr>
<td>M</td>
<td>100</td>
<td>80</td>
<td>140</td>
</tr>
</tbody>
</table>

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:
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 place 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.

25.2.4. Distinguishing between probabilities

(5 pts.) Suppose that $Y$ is a random variable taking on one of the $n$ know values:

$$a_1, a_2, \ldots, a_n.$$ 

Suppose we know that $Y$ either has distribution $p$ given by

$$P(Y = a_j) = p_j$$

or it has distribution $q$ given by

$$P(Y = a_j) = q_j.$$ 

Of course, the numbers $p_j, j = 1, 2, \ldots, n$ 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 $x_j, j = 1, 2, \ldots, n$, such that the probability of saying the distribution is $p$ when in fact it is $q$ has probability no larger than $\beta$, where $\beta$ 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.

25.2.5. 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:

$$\text{minimize } \sum_{e \in E(G)} \text{cost}(e)x_e$$

subject to

$$x_e \in \{0, 1\} \quad \forall e \in E(G)$$

$$\sum_{e \in \pi} x_e \geq 1 \quad \forall \pi \in \mathcal{P}.$$ 

(A) (5 pts.) What does this IP computes?

(B) (5 pts.) Write down the relaxation of this IP into a linear program.

(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)?
(D) (5 pts.) The strong duality theorem states the following.

**Theorem 25.2.1.** If the primal LP problem has an optimal solution \( x^* = (x_1^*, \ldots, x_n^*) \) then the dual also has an optimal solution, \( y^* = (y_1^*, \ldots, y_m^*) \), such that

\[
\sum_j c_jx_j^* = \sum_i b_iy_i^*.
\]

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 26

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.

26.1. Introduction

In this chapter, we will address the problem of multiplying two polynomials quickly.

Definition 26.1.1. A polynomial \( p(x) \) of degree \( n \) is a function of the form \( p(x) = \sum_{j=0}^{n} a_j x^j = a_0 + x(a_1 + x(a_2 + \ldots + xa_n)) \).

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 26.1.2. For any set \( \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} \) 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 \( k = 0, \ldots, n - 1 \).

An explicit formula for \( p(x) \) as a function of those point-value pairs is

\[
p(x) = \sum_{i=0}^{n-1} y_i \prod_{j \neq i}(x - x_j).
\]

Note, that the \( i \)th term in this summation is zero for \( X = x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1} \), 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

\[
\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\} \quad \text{for} \quad p(x),
\]

and \( \{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\} \quad \text{for} \quad q(x) \).
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

\[
\{(x_0, r(x_0)), \ldots, (x_{2n-1}, r(x_{2n-1}))\} = \{(x_0, p(x_0)q(x_0)), \ldots, (x_{2n-1}, p(x_{2n-1})q(x_{2n-1}))\} = \{(x_0, y_0 y'_0), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\}.
\]

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.

### 26.2. 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 \( \Psi = \{x_1, \ldots, x_n\} \), 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 \( \Psi \), 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 \( \text{SQ}(\Psi) \) 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 \( |\text{SQ}(\Psi)| = n/2^i \) numbers, for \( i = 0, \ldots, k \). Let us call such a set of numbers collapsible.

Given a set of numbers \( \mathcal{X} = \{x_0, \ldots, x_n\} \) and a polynomial \( p(x) \), let

\[
p(\mathcal{X}) = \{(x_0, p(x_0)), \ldots, (x_n, p(x_n))\}.
\]

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 \( \Psi \), 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 \( u(\text{SQ}(A)), v(\text{SQ}(A)) \) we can, in linear time, compute \( p(\Psi) \). But, \( \text{SQ}(\Psi) \) have \( n/2 \) value pairs because we assumed that \( \Psi \) 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 26.1.

What is the running time of FFTAlg? 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 (\( \Psi \) is collapsible!). Thus, the running time is \( T(n) = 2T(n/2) + O(n) \), which is \( O(n \log n) \) — exactly what we wanted.
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

\( u(y) = \sum_{i=0}^{n/2-1} a_{2i} y^i \)
\( v(y) = \sum_{i=0}^{n/2-1} a_{1+2i} y^i \).
\( Y = SQ(X) = \{ x^2 \mid x \in X \} \).
\( U = \text{FFTAlg}(u, Y) \) /* U = u(Y) */
\( V = \text{FFTAlg}(v, Y) \) /* 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 \( \cup \) \( \{(x, p(x))\} \)

end

return Out

Figure 26.1: The FFT algorithm.

26.2.1. 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 26.4.1. 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 \( \gamma \) denote a \( n \)th root of unity. There are \( n \) such roots, and let \( \gamma_j(n) \) denote the \( j \)th root, see Figure 26.2 on page 171. In particular, let

\[
\gamma_j(n) = \cos((2\pi j)/n) + i \sin((2\pi j)/n) = \gamma^j.
\]

Let \( \mathcal{A}(n) = \{ \gamma_0(n), \ldots, \gamma_{n-1}(n) \} \). It is easy to verify that \( |SQ(\mathcal{A}(n))| \) has exactly \( n/2 \) elements. In fact, \( SQ(\mathcal{A}(n)) = \mathcal{A}(n/2) \), 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 26.2.1.** 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.

26.3. Recovering the polynomial

This part of the writeup is somewhat more technical. Putting it shortly, we are going to apply the FFTAlg algorithm once again to recover the original polynomial. The details follow.
We need to use the fact here that evaluating \( p(V) \) evaluating \( p(V) \).

Proof: Consider the \( \beta \)

\[
\begin{pmatrix}
\gamma_0 & \gamma_1 & \gamma_2 & \cdots & \gamma_{n-1} \\
1 & 1 & 1 & \cdots & 1 \\
\gamma_1 & \gamma_2 & \gamma_3 & \cdots & \gamma_{n-1} \\
1 & \gamma_1 & \gamma_2 & \cdots & \gamma_{n-1} \\
\vdots & \vdots & \vdots & \cdots & \vdots \\
\gamma_{n-1} & \gamma_{n-1} & \gamma_{n-1} & \cdots & \gamma_{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:

\[
\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
\vdots \\
a_{n-1}
\end{pmatrix}
= V^{-1}
\begin{pmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
\vdots \\
y_{n-1}
\end{pmatrix}
\]

Namely, we can recover the polynomial \( p(x) \) from the point-value pairs

\[
\{(y_0, p(y_0)), (y_1, p(y_1)), \ldots, (y_{n-1}, p(y_{n-1}))\}
\]

by doing a single matrix multiplication of \( V^{-1} \) by the vector \([y_0, y_1, \ldots, y_{n-1}]\). However, multiplying a vector with \( n \) entries with a matrix of size \( n \times n \) takes \( O(n^2) \) time. Thus, we had not benefitted anything so far.

However, since the Vandermonde matrix is so well behaved\(^8\), it is not too hard to figure out the inverse matrix.

Claim 26.3.1.

\[
V^{-1} = \frac{1}{n}
\begin{pmatrix}
1 & \beta_0 & \beta_0^2 & \beta_0^3 & \cdots & \beta_0^{n-1} \\
1 & \beta_1 & \beta_1^2 & \beta_1^3 & \cdots & \beta_1^{n-1} \\
1 & \beta_2 & \beta_2^2 & \beta_2^3 & \cdots & \beta_2^{n-1} \\
1 & \beta_3 & \beta_3^2 & \beta_3^3 & \cdots & \beta_3^{n-1} \\
\vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\
1 & \beta_{n-1} & \beta_{n-1}^2 & \beta_{n-1}^3 & \cdots & \beta_{n-1}^{n-1}
\end{pmatrix}
\]

where \( \beta_j = (\gamma_j(n))^{-1} \).

Proof: Consider the \((u, v)\) entry in the matrix \( C = V^{-1}V \). We have

\[
C_{u,v} = \sum_{j=0}^{n-1} \frac{\beta_u v_j}{n}.
\]

We need to use the fact here that \( \gamma_j = (\gamma_1)^j \) as can be easily verified. Thus,

\[
C_{u,v} = \sum_{j=0}^{n-1} \frac{\beta_u \gamma_j^v}{n} = \sum_{j=0}^{n-1} \frac{(\beta_u \gamma_j)^v}{n} = \sum_{j=0}^{n-1} \frac{(\beta_u \gamma_j)^v}{n}.
\]

\(^8\)Not to mention famous, beautiful and well known – in short a celebrity matrix.
Clearly, if \( u = v \) then

\[
C_{u,u} = \frac{1}{n} \sum_{j=0}^{n-1} (\beta_j u^j) = \frac{1}{n} \sum_{j=0}^{n-1} (1)^j = \frac{n}{n} = 1.
\]

If \( u \neq v \) then,

\[
\beta_u v = (\gamma_u)^{-1} \gamma_v = (\gamma_1)^{-u} \gamma_1^v = (\gamma_1)^{-u} = \gamma_{v-u}.
\]

And

\[
C_{u,v} = \frac{1}{n} \sum_{j=0}^{n-1} (\gamma_{v-u})^j = \frac{1}{n} \cdot \frac{\gamma_n^{n} - 1}{\gamma_{v-u} - 1} = \frac{1 - 1}{\gamma_{v-u} - 1} = 0,
\]

this follows by the formula for the sum of a geometric series, and as \( \gamma_{v-u} \) 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. ■

Let us recap, given \( n \) point-value pairs \([\{\gamma_0, y_0\}, \ldots, \{\gamma_n-1, y_n-1\}]\) 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 \([y_0, y_1, \ldots, y_n]\) by the matrix \( V^{-1} \). Namely,

\[
\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
\vdots \\
a_{n-1}
\end{pmatrix} = \frac{1}{n} \begin{pmatrix} 1 & \beta_0 & \beta_0^2 & \beta_0^3 & \cdots & \beta_0^{n-1} \\
1 & \beta_1 & \beta_1^2 & \beta_1^3 & \cdots & \beta_1^{n-1} \\
1 & \beta_2 & \beta_2^2 & \beta_2^3 & \cdots & \beta_2^{n-1} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \beta_{n-1} & \beta_{n-1}^2 & \beta_{n-1}^3 & \cdots & \beta_{n-1}^{n-1}
\end{pmatrix} \begin{pmatrix} y_0 \\
y_1 \\
y_2 \\
\vdots \\
y_{n-1}
\end{pmatrix}.
\]

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: \( \beta_0, \ldots, \beta_{n-1} \).

The final stroke, is to observe that \( \{\beta_0, \ldots, \beta_{n-1}\} = \{\gamma_0, \ldots, \gamma_{n-1}\} \); indeed \( \beta_i^n = (\gamma_i^{-1})^n = (\gamma_i^n)^{-1} = 1^{-1} = 1 \). Namely, we can apply the FFTAlg algorithm on \( W(x) \) to compute \( a_0, \ldots, a_{n-1} \).

We conclude:

**Theorem 26.3.2.** 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 26.3.3.** Given two polynomials of degree \( n \), they can be multiplied in \( O(n \log n) \) time.

### 26.4. The Convolution Theorem

Given two vectors \( A = [a_0, a_1, \ldots, a_n] \) and \( B = [b_0, \ldots, b_n] \), their dot product is the quantity

\[
A \cdot B = (A, B) = \sum_{i=0}^{n} a_i b_i.
\]

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 \( j \notin \{0, \ldots, n\} \)).

\[
A_r = [a_{n-r}, a_{n+1-r}, a_{n+2-r}, \ldots, a_{2n-r}]
\]

where \( a_j = 0 \) if \( j \notin \{0, \ldots, n\} \).
Indeed, setting $\alpha$ The resulting vector $[c_0, \ldots, c_{2n}]$ is the convolution of $A$ and $B$.

**Question 26.4.4.** How to compute the convolution of two vectors of length $n$?

**Definition 26.4.5.** The resulting vector $[c_0, \ldots, c_{2n}]$ is the convolution of $A$ and $B$.

Let $p(x) = \sum_{i=0}^{n} a_i x^i$, and $q(x) = \sum_{i=0}^{n} b_i x^i$. The coefficient of $x^i$ in $r(x) = p(x)q(x)$ is

$$d_i = \sum_{j=0}^{i} a_j b_{i-j}.$$  

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 $a_i = a_i$ and $b_i = b_{n-i}$ 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 $p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$ and $q(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3$ 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 $a_i b_j$.

<table>
<thead>
<tr>
<th></th>
<th>$a_0$</th>
<th>$a_1 x$</th>
<th>$a_2 x^2$</th>
<th>$a_3 x^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_0$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$+ b_1 x$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$+ b_2 x^2$</td>
<td>$a_1 b_1 x^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$+ b_3 x^3$</td>
<td>$a_0 b_2 x^2$</td>
<td>$a_1 b_2 x^3$</td>
<td>$a_2 b_0 x^2$</td>
<td>$a_3 x^3$</td>
</tr>
</tbody>
</table>

**Theorem 26.4.6.** Given two vectors $A = [a_0, a_1, \ldots, a_n]$, $B = [b_0, \ldots, b_n]$ 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 $c_0, \ldots, c_{2n}$ be the coefficients of $r(x)$. It is easy to verify, as described above, that $[c_0, \ldots, c_{2n}]$ is the convolution of $A$ and $B$.  

---

### 26.4.1. 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 $\tau = r \cos \phi + ir \sin \phi = r(\cos \phi + i \sin \phi) = re^{i\phi}$, 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

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots,$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots,$$

and

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots.$$  

Since $i^2 = -1$, we have that

$$e^{ix} = 1 + i \frac{x}{1!} - \frac{x^2}{2!} - i \frac{x^3}{3!} + \frac{x^4}{4!} + i \frac{x^5}{5!} - \frac{x^6}{6!} - \cdots = \cos x + i \sin x.$$
Figure 26.2: (A) The 16 roots of unity. (B) The 8 roots of unity. (C) The 4 roots of unity.

The nice thing about polar form, is that given two complex numbers \( \tau = re^{i\phi} \) and \( \tau' = r'e^{i\phi'} \), multiplying them is now straightforward. Indeed, \( \tau \cdot \tau' = re^{i\phi} \cdot r'e^{i\phi'} = rr'e^{i(\phi+\phi')} \). Observe that the function \( e^{i\theta} \) is \( 2\pi \) 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 \( \tau = re^{i\phi} \) 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 \( j = 0, \ldots, n - 1 \), which are the \( n \) distinct roots of unity.

Chapter 27

Sorting Networks

27.1. 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 \( S = \{s_1, \ldots, s_n\} \) 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 \( i = 1, \ldots, n \). 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 it 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.

### 27.2. 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.

#### 27.2.1. Definitions

**Definition 27.2.1.** 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 27.2.2.** 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 27.2.3.** 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.
27.2.2. 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 27.1 (ii). The next natural question, is how much 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 27.1 (ii).

In particular, the above discussion implies the following result.

**Lemma 27.2.4.** The sorting network based on insertion sort has \( O(n^2) \) gates, and requires \( 2n - 1 \) time units to sort \( n \) numbers.

27.3. 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 27.3.1.** If a comparison network transforms the input sequence \( a = \langle a_1, a_2, \ldots, a_n \rangle \) into the output sequence \( b = \langle b_1, b_2, \ldots, b_n \rangle \), then for any monotonically increasing function \( f \), the network transforms the input sequence \( f(a) = \langle f(a_1), \ldots, f(a_n) \rangle \) into the sequence \( f(b) = \langle f(b_1), \ldots, f(b_n) \rangle \).

**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

\[
\max(f(x), f(y)) = f(\max(x, y)) \quad \text{and} \quad \min(f(x), f(y)) = f(\min(x, y)),
\]

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 \( a_1, \ldots, a_n \), then for the input \( f(a_1), \ldots, f(a_n) \) 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 27.3.2.** 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 \(a_1, \ldots, a_n\). Let \(b_1, \ldots, b_n\) 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 27.3.1), for the input

\[
\langle f(a_1), \ldots, f(a_n) \rangle,
\]

which is a binary sequence, the circuit would output \(\langle f(b_1), \ldots, f(b_n) \rangle\). But then, this sequence looks like

\[
000\ldots f(a_k)????f(a_i)??1111
\]

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., \(\langle f(b_1), \ldots, f(b_n) \rangle\)) for which the comparison network does not sort it correctly. A contradiction to our assumption.

27.4. A bitonic sorting network

Definition 27.4.1. A \textit{bitonic sequence} is a sequence which is first increasing and then decreasing, or can be circularly shifted to become so.

Example 27.4.2. 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 27.4.3. A binary bitonic sequence (i.e., bitonic sequence made out only of zeroes and ones) is either of the form \(0^i1^j0^k\) or of the form \(1^i0^j1^k\), where \(0^i\) (resp., \(1^i\)) denote a sequence of \(i\) zeros (resp., ones).

Definition 27.4.4. A \textit{bitonic sorter} is a comparison network that sorts all bitonic sequences correctly.

Definition 27.4.5. A \textit{half-cleaner} is a comparison network, connecting line \(i\) with line \(i + n/2\).

In particular, let \textbf{Half-Cleaner}[\(n\)] denote the half-cleaner with \(n\) inputs. Note, that the depth of a \textbf{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 27.4.6. 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 \(0^i1^j0^k\) 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 \(n/2 + \alpha\).
Figure 27.2: Depicted are the (i) recursive construction of BitonicSorter[n], (ii) opening up the recursive construction, and (iii) the resulting comparison network.

Figure 27.3: (i) Merger via flipping the lines of bitonic sorter. (ii) A BitonicSorter. (ii) The Merger after we “physically” flip the lines, and (iv) An equivalent drawing of the resulting Merger.

If $n/2 - \alpha > \beta$ 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 $1^i0^j1^k$.

This suggests a simple recursive construction of BitonicSorter[n], see Figure 27.2, and we have the following lemma.

**Lemma 27.4.7.** 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 $k = \lg n$.

### 27.4.1. 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 $a_1 \leq a_2 \leq \ldots \leq a_n$ and $b_1 \leq b_2 \leq \ldots \leq b_n$, observe that the sequence $a_1, a_2, \ldots, a_n, b_n, b_{n-1}, b_{n-2}, \ldots, b_2, b_1$ is bitonic.

Thus, to merge two sorted sequences of length $n/2$, just flip one of them, and use BitonicSorter[n], see Figure 27.3. 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 27.3. 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 27.4
Lemma 27.4.8. 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.

### 27.5. 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 27.5.1.** 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

\[
G(n) = 2G(n/2) + \text{Gates(Merger}[n]).
\]

Which is \(G(n) = 2G(n/2) + O(n \log n) = O(n \log^2 n)\).

As for the depth, we have that \(D(n) = D(n/2) + \text{Depth(Merger}[n]) = D(n/2) + O(\log(n))\), and thus \(D(n) = O(\log^2 n)\), as claimed.

### 27.6. 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 \(2^{8046}\).
Chapter 28

Union Find

28.1. Union-Find

28.1.1. Requirements from the data-structure

We want to maintain a collection of sets, under the following operations.

(i) \text{makeSet}(x) - creates a set that contains the single element \(x\).

(ii) \text{find}(x) - returns the set that contains \(x\).

(iii) \text{union}(A,B) - returns the set which is the union of \(A\) and \(B\). Namely \(A \cup B\).

28.1.2. 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 \textit{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 overall running time divided by number of operations.

28.1.3. 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 28.1 for an example of how such a reversed tree looks like.

We implement the operations of the Union-Find data structure as follows:

(A) \text{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 28.1: 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 28.2:** (a) The tree before performing $\text{find}(z)$, and (b) The reversed tree after performing $\text{find}(z)$ that uses path compression.

(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 $\text{find}$ 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 $\text{find}$ 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 $\text{find}$ 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 28.2. For the pseudo-code of the makeSet, union and find using path compression and union by rank, see Figure 28.3.

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.

28.2. Analyzing the Union-Find Data-Structure

Definition 28.2.1. A node in the union-find data-structure is a leader if it is the root of a (reversed) tree.

Lemma 28.2.2. 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 28.2.3. 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 28.2.4. 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 root(A) on 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 \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \). However, by induction, \( \text{rank}(v) \) is larger than the rank of all the other nodes of \( \pi \). In particular, all the nodes that get compressed, the rank of their new parent, is larger than their own rank.

Lemma 28.2.5. 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 28.2.6. 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 28.2.7. The time to perform a single \texttt{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 $\log n$ and thus the depth of a tree is at most $O(\log n)$. \qed

Surprisingly, we can do much better.

Theorem 28.2.8. 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 $\log$ 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 $\log^* 2^{2^{2^{2^2}}} = \log^*(65536) = 5$.

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 28.2.8 states (in the amortized sense), that the Union-Find data-structure takes constant time per operation (unless $n$ is larger than $\beta$ which is unlikely).

It would be useful to look on the inverse function to $\log^*$.

Definition 28.2.9. Let $\text{Tower}(b) = 2^{\text{Tower}(b-1)}$ and $\text{Tower}(0) = 1$.

So, $\text{Tower}(i)$ is just a tower of $2^{2^{-2}}$ of height $i$. Observe that $\log^* (\text{Tower}(i)) = i$.

Definition 28.2.10. 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,

$\text{Block}(0) = [0, 1]$
$\text{Block}(1) = [2, 2]$
$\text{Block}(2) = [3, 4]$
$\text{Block}(3) = [5, 16]$
$\text{Block}(4) = [17, 65536]$
$\text{Block}(5) = [65537, 2^{65536}]$

$\vdots$ 

The running time of \texttt{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:

$$x_1 = x, x_2 = \overline{p}(x) = \overline{p}(x_1), \ldots, x_i = \overline{p}(x_{i-1}), \ldots, x_m = \text{root of tree}.$$ 

Clearly, we have for this sequence: $\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m)$, and the time it takes to perform $\texttt{find}(x)$ is proportional to $m$, the length of the path from $x$ to the root of the tree containing $x$.

Definition 28.2.11. 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 \texttt{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 \( \text{Block}(c \log^* n) \), for \( c \) a constant sufficiently large.

In particular, consider a \textbf{find} \((x)\) operation, and let \( \pi \) be the path visited. Next, consider the ranks of the elements of \( \pi \), and imagine partitioning \( \pi \) into which blocks each element rank belongs to. An example of such a path is depicted on the right. The price of the \textbf{find} operation is the length of \( \pi \).

Formally, for a node \( x \), \( \nu = \text{index}_B(x) \) 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 \textbf{find} operation, since the ranks of the nodes we visit are monotone increasing, once we pass through from a node \( \nu \) 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 \( \pi \) that are next to a element in a different block, to the number of blocks (which is \( O(\log^* n) \)).

**Definition 28.2.12.** Consider a path \( \pi \) traversed by a \textbf{find} operation. Along the path \( \pi \), an element \( x \), such that \( \nu(x) \) is in a different block, is a \textit{jump between blocks}.

On the other hand, a jump during a \textbf{find} operation inside a block is called an \textit{internal jump}; that is, \( x \) and \( \nu(x) \) are in the same block.

**Lemma 28.2.13.** \textit{During a single \( \textbf{find}(x) \) operation, the number of jumps between blocks along the search path is \( O(\log^* n) \).}

**Proof:** Consider the search path \( \pi = x_1, \ldots, x_m \), and consider the list of numbers \( 0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \ldots \leq \text{index}_B(x_m) \). We have that \( \text{index}_B(x_m) = O(\log^* n) \). As such, the number of elements \( x \) in \( \pi \) such that \( \text{index}_B(x) \neq \text{index}_B(\nu(x)) \) is at most \( O(\log^* n) \). \hfill \Box

Consider the case that \( x \) and \( \nu(x) \) are both the same block (i.e., \( \text{index}_B(x) = \text{index}_B(\nu(x)) \)) and we perform a find operation that passes through \( x \). Let \( r_{\text{bef}} = \text{rank}(\nu(x)) \) before the \textbf{find} operation, and let \( r_{\text{aft}} = \text{rank}(\nu(x)) \) after the \textbf{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 28.2.14.** \textit{At most \( |\text{Block}(i)| \leq \text{Tower}(i) \) \textbf{find} operations can pass through an element \( x \), which is in the \( i \)th block (i.e., \( \text{index}_B(x) = i \)) before \( \nu(x) \) is no longer in the \( i \)th block. That is \( \text{index}_B(\nu(x)) > i \).}

**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 28.2.10. \hfill \Box

**Lemma 28.2.15.** \textit{There are at most \( n/\text{Tower}(i) \) nodes that have ranks in the \( i \)th block throughout the algorithm execution.}

**Proof:** By Lemma 28.2.6, we have that the number of elements with rank in the \( i \)th block is at most

\[
\sum_{k \in \text{Block}(i)} \frac{n}{2^k} \leq \sum_{k = \text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{n}{2^k} = n \cdot \sum_{k = \text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{1}{2^k} \leq \frac{n}{2^{\text{Tower}(i-1)}} = \frac{n}{\text{Tower}(i)}. \hfill \Box
\]

**Lemma 28.2.16.** \textit{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 28.2.14. There are at most \( n/\text{Tower}(i) \) elements with ranks in the \( i \)th block, throughout the algorithm execution, by Lemma 28.2.15. Thus, the total number of internal jumps is

\[
|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n. \]

We are now ready for the last step.

Lemma 28.2.17. 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 28.2.16. There are \( O(\log^* n) \) blocks. As such there are at most \( O(n \log^* n) \) internal jumps.

Lemma 28.2.18. The overall time spent on \( m \) \textbf{find} operations, throughout the lifetime of a union-find data-structure defined over \( n \) elements, is \( O((m + n) \log^* n) \).

Theorem 28.2.8 now follows readily from the above discussion.

Chapter 29

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 \textit{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.

29.1. Problem Statement

Given an undirected graph \( G = (V, E) \) and nonnegative weights \( \omega_{ij} \), for all \( i, j \in E \), the \textit{maximum cut problem} (\textbf{MAX CUT}) is that of finding the set of vertices \( S \) that maximizes the weight of the edges in the cut \( (S, \overline{S}) \); that is, the weight of the edges with one endpoint in \( S \) and the other in \( \overline{S} \). For simplicity, we usually set \( \omega_{ij} = O \) for \( i, j \notin E \) and denote the weight of a cut \( (S, \overline{S}) \) by \( w(S, \overline{S}) = \sum_{i \in S, j \in \overline{S}} \omega_{ij} \).

This problem is \textbf{NP-COMPLETE}, and hard to approximate within a certain constant.
Given a graph with vertex set \( V = \{1, \ldots, n\} \) and nonnegative weights \( \omega_{ij} \), the weight of the maximum cut \( w(S, \overline{S}) \) is given by the following integer quadratic program:

\[
\begin{align*}
\text{(Q)} \quad & \max \quad \frac{1}{2} \sum_{i<j} \omega_{ij}(1 - y_i y_j) \\
\text{subject to:} \quad & y_i \in \{-1, 1\} \quad \forall i \in V.
\end{align*}
\]

Indeed, set \( S = \{i \mid y_i = 1\} \). Clearly, \( \omega(S, \overline{S}) = \frac{1}{2} \sum_{i<j} \omega_{ij}(1 - y_i y_j) \).

Solving quadratic integer programming is of course \textbf{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:

\[
\begin{align*}
\text{(P)} \quad & \max \quad \gamma = \frac{1}{2} \sum_{i<j} \omega_{ij} \left(1 - \langle v_i, v_j \rangle \right) \\
\text{subject to:} \quad & v_i \in \mathbb{S}^{(n)} \quad \forall i \in V,
\end{align*}
\]

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 \( r \) on the unit sphere \( \mathbb{S}^{(n)} \). This induces a hyperplane \( h \) which passes through the origin and is orthogonal to \( r \). We next assign all the vectors that are on one side of \( h \) to \( S \), and the rest to \( \overline{S} \).

Summarizing, the algorithm is as follows: First, we solve (P), next, we pick a random vector \( r \) uniformly on the unit sphere \( \mathbb{S}^{(n)} \). Finally, we set

\[
S = \left\{ v_i \mid \langle v_i, r \rangle \geq 0 \right\}.
\]

### 29.1.1. 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 29.1.1.** We have \( \Pr[\text{sign}(v_i, r) \neq \text{sign}(v_j, r)] = \frac{1}{\pi} \arccos(\langle v_i, v_j \rangle) \).

**Proof:** Let us think about the vectors \( v_i, v_j \) and \( r \) 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 \( r \) matter, which can be decided by projecting \( r \) on \( g \), and normalizing it to have length \( 1 \). Now, the sphere is symmetric, and as such, sampling \( r \) 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, \( \text{sign}(\langle v_i, r \rangle) \neq \text{sign}(\langle v_j, r \rangle) \) happens only if \( r \) 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,

\[
\Pr[\text{sign}(\langle v_i, r \rangle) \neq \text{sign}(\langle v_j, r \rangle)] = \frac{2\tau}{2\pi} = \frac{1}{\pi} \cdot \arccos(\langle v_i, v_j \rangle),
\]

as claimed.
Theorem 29.1.2. Let $W$ be the random variable which is the weight of the cut generated by the algorithm. We have

$$
E[W] = \frac{1}{\pi} \sum_{i<j} \omega_{ij} \arccos\left(\langle v_i, v_j \rangle\right).
$$

Proof: Let $X_{ij}$ be an indicator variable which is 1 if and only if the edge $ij$ is in the cut. We have

$$
E[X_{ij}] = \Pr[\text{sign}(\langle v_i, r \rangle) \neq \text{sign}(\langle v_j, r \rangle)] = \frac{1}{\pi} \arccos\left(\langle v_i, v_j \rangle\right),
$$

by Lemma 29.1.1. Clearly, $W = \sum_{i<j} \omega_{ij} X_{ij}$, and by linearity of expectation, we have

$$
E[W] = \sum_{i<j} \omega_{ij} E[X_{ij}] = \frac{1}{\pi} \sum_{i<j} \omega_{ij} \arccos\left(\langle v_i, v_j \rangle\right).
$$

Lemma 29.1.3. For $-1 \leq y \leq 1$, we have

$$
\frac{\arccos(y)}{\pi} \geq \alpha \cdot \frac{1}{2}(1 - y),
$$

where $\alpha = \min_{0 \leq \psi \leq \pi} \frac{\psi}{\pi (1 - \cos \psi)}$.

Proof: Set $y = \cos(\psi)$. The inequality now becomes

$$
\frac{\psi}{\pi} \geq \frac{1}{2}(1 - \cos \psi).\text{ Reorganizing, the inequality becomes}
\frac{2}{\pi} \frac{\psi}{1 - \cos \psi} \geq \alpha,
$$

which trivially holds by the definition of $\alpha$.

 Lemma 29.1.4. $\alpha > 0.87856$.

Proof: Using simple calculus, one can see that $\alpha$ achieves its value for $\psi = 2.331122\ldots$, the nonzero root of $\cos \psi + \psi \sin \psi = 1$.

Theorem 29.1.5. The above algorithm computes in expectation a cut with total weight $\alpha \cdot \text{Opt} \geq 0.87856 \text{Opt}$, where $\text{Opt}$ 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

$$
E[W] = \frac{1}{\pi} \sum_{i<j} \omega_{ij} \arccos\left(\langle v_i, v_j \rangle\right) \geq \sum_{i<j} \omega_{ij} \alpha \frac{1}{2}(1 - \langle v_i, v_j \rangle) = \alpha \gamma \geq \alpha \cdot \text{Opt},
$$

by Lemma 29.1.3.

29.2. 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 $i = 1, \ldots, n$. Let $V$ be the matrix having $v_1, \ldots, v_n$ 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

$$(SD) \quad \max \quad \frac{1}{2} \sum_{i<j} \omega_{ij}(1 - x_{ij})$$

subject to: $x_{ii} = 1$ for $i = 1, \ldots, n$

$$(x_{ij})_{i=1,\ldots,n,j=1,\ldots,n} \text{ is a positive semi-definite matrix.}$$

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 29.2.1. 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:

$$v^T (tA + (1-t)B)v = v^T (tAv + (1-t)Bv) \geq 0 + 0 \geq 0,$$

since $A$ and $B$ are positive semidefinite.

Positive semidefinite matrices corresponds to ellipsoids. Indeed, consider the set $x^T Ax = 1$: 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.

29.3. 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 $\alpha$, 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].

Chapter 30

The Perceptron Algorithm

30.1. 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 30.1. That is, we are trying to learn a line $\ell$ 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 $\ell$? 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 $a = (a_1, \ldots, a_d), b = (b_1, \ldots, b_d) \in \mathbb{R}^2$, 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 outside the class, if it is positive it is inside.

A set of training examples is a set of pairs $S = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $x_i \in \mathbb{R}^d$ and $y_i \in \{-1, 1\}$, for $i = 1, \ldots, n$.

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 $(w, b)$, such that for an $(x_i, y_i)$ we have $\text{sign}(\langle w, x_i \rangle + b) = y_i$, which is

$$\langle w, x_i \rangle + b \geq 0 \quad \text{if } y_i = 1,$$

and

$$\langle w, x_i \rangle + b \leq 0 \quad \text{if } y_i = -1.$$  

Or equivalently, let $x_i = \left(x_i^1, \ldots, x_i^d\right) \in \mathbb{R}^d$, for $i = 1, \ldots, m$, and let $w = \left(w^1, \ldots, w^d\right)$, then we get the linear constraint

$$\sum_{k=1}^d w_k x_i^k + b \geq 0 \quad \text{if } y_i = 1,$$

and

$$\sum_{k=1}^d w_k x_i^k + b \leq 0 \quad \text{if } y_i = -1.$$
Algorithm **perceptron**$(S$: a set of $l$ examples$)$

$$
\begin{align*}
\mathbf{w}_0 &\leftarrow 0 , k \leftarrow 0 \\
R &\leftarrow \max_{(x,y) \in S} \|x\|. \\
\text{repeat} & \quad \text{for } (x,y) \in S \\
& \quad \text{if } \text{sign}(\langle \mathbf{w}_k , x \rangle) \neq y \\
& \quad \quad \mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + y \cdot x \\
& \quad \quad k \leftarrow k + 1 \\
\text{until no mistakes are made in the classification} & \quad \text{return } \mathbf{w}_k \text{ and } k
\end{align*}
$$

Figure 30.2: The **perceptron** algorithm.

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 30.2.

Why does the **perceptron** algorithm converges to the right solution? Well, assume that we made a mistake on a sample $(x, y)$ and $y = 1$. Then, $\langle \mathbf{w}_k , x \rangle < 0$, and

$$
\langle \mathbf{w}_{k+1} , x \rangle = \langle \mathbf{w}_k + y \cdot x , x \rangle = \langle \mathbf{w}_k , x \rangle + y \langle x , x \rangle = \langle \mathbf{w}_k , x \rangle + y \|x\| > \langle \mathbf{w}_k , x \rangle.
$$

Namely, we are “walking” in the right direction, in the sense that the new value assigned to $x$ by $\mathbf{w}_{k+1}$ is larger (“more positive”) than the old value assigned to $x$ by $\mathbf{w}_k$.

**Theorem 30.1.1.** Let $S$ be a training set of examples, and let $R = \max_{(x,y) \in S} \|x\|$. Suppose that there exists a vector $\mathbf{w}_{opt}$ such that $\|\mathbf{w}_{opt}\| = 1$, and a number $\gamma > 0$, such that

$$
y \langle \mathbf{w}_{opt}, x \rangle \geq \gamma \quad \forall (x,y) \in S.
$$

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 $\mathbf{w}_{opt}$. To see that, let us define the distance between $\mathbf{w}_{opt}$ and the weight vector in the $k$th update:

$$
\alpha_k = \left\| \mathbf{w}_k - \frac{R^2}{\gamma} \mathbf{w}_{opt} \right\|^2.
$$

We next quantify the change between $\alpha_k$ and $\alpha_{k+1}$ (the example being misclassified is $(x,y)$):

$$
\begin{align*}
\alpha_{k+1} &= \left\| \mathbf{w}_{k+1} - \frac{R^2}{\gamma} \mathbf{w}_{opt} \right\|^2 \\
&= \left\| \mathbf{w}_k + yx - \frac{R^2}{\gamma} \mathbf{w}_{opt} \right\|^2 \\
&= \left\| \mathbf{w}_k - \frac{R^2}{\gamma} \mathbf{w}_{opt} + yx \right\|^2 \\
&= \left( \mathbf{w}_k - \frac{R^2}{\gamma} \mathbf{w}_{opt} \right) + yx.
\end{align*}
$$
Expanding this we get:

\[ \alpha_{k+1} = \left( \left( w_k - \frac{R^2}{\gamma} w_{opt} \right) \cdot \left( w_k - \frac{R^2}{\gamma} w_{opt} \right) \right) + 2y \left( \left( w_k - \frac{R^2}{\gamma} w_{opt} \right) , x \right) + \langle x, x \rangle \]

As \((x, y)\) is misclassified, it must be that sign \((y \langle w_k, x \rangle)\) \(=\) \(-1\); that is \(y \langle w_k, x \rangle < 0\).
Now, since \(\|x\| \leq R\), we have

\[ \alpha_{k+1} \leq \alpha_k + 2y \left( w_k - \frac{R^2}{\gamma} w_{opt}, x \right) \leq \alpha_k + R^2 + \frac{2R^2}{\gamma} \langle w_{opt}, x \rangle. \]

Next, since \(y \langle w_{opt}, x \rangle \geq \gamma\) for \(\forall (x, y) \in S\), we have that

\[ \alpha_{k+1} \leq \alpha_k + R^2 - \frac{2R^2}{\gamma} \]

We have: \(\alpha_{k+1} \leq \alpha_k - R^2\), and

\[ \alpha_0 = \left\| 0 - \frac{R^2}{\gamma} w_{opt} \right\|^2 = \frac{R^4}{\gamma^2} \left\| w_{opt} \right\|^2 = \frac{R^4}{\gamma^2}. \]

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.

### 30.2. 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 \(\sigma\)?

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 30.2.1.** 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

\[
-2ax - 2by + \left(x^2 + y^2\right) \leq r^2 - a^2 - b^2.
\]

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 \( (x, y, x^2 + y^2) \in B \) 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

\[
ax + by + c\left(x^2 + y^2\right) \leq -d.
\]

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 show 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**.

### 30.3. 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 $X = \{p_1, p_2, \ldots, p_m\}$ 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

$$r(X) = (b_1, \ldots, b_m)$$

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.

### 30.3.1. 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?

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 30.3.1 (Sauer Lemma).** If $R$ has VC dimension $d$ then $|U(X, R)| = O(m^d)$, where $m$ is the size of $X$. 

190
Chapter 31

Huffman Coding

31.1. 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, were 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 $S$ (···) includes the code for $E$ (·) as a prefix. (Hopefully the receiver knows that when it gets ··· 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 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 output. Thus, the binary string ’010100’ encodes the string “bda”.

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 $f[1 \ldots n]$, we want to compute a prefix-free binary code that minimizes the
Figure 31.1: 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.

<table>
<thead>
<tr>
<th>char</th>
<th>frequency</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘A’</td>
<td>48165</td>
<td>1110</td>
</tr>
<tr>
<td>‘B’</td>
<td>8414</td>
<td>10100</td>
</tr>
<tr>
<td>‘C’</td>
<td>13896</td>
<td>00100</td>
</tr>
<tr>
<td>‘D’</td>
<td>28041</td>
<td>0011</td>
</tr>
<tr>
<td>‘E’</td>
<td>74809</td>
<td>011</td>
</tr>
<tr>
<td>‘F’</td>
<td>13559</td>
<td>11111</td>
</tr>
<tr>
<td>‘G’</td>
<td>12530</td>
<td>11110</td>
</tr>
<tr>
<td>‘H’</td>
<td>38961</td>
<td>1001</td>
</tr>
</tbody>
</table>

Figure 31.2: The resulting prefix code for the frequencies of Figure 31.1. Here, for the sake of simplicity of exposition, the code was constructed only for the A—Z characters.

<table>
<thead>
<tr>
<th>char</th>
<th>frequency</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘I’</td>
<td>41005</td>
<td>1011</td>
</tr>
<tr>
<td>‘J’</td>
<td>710</td>
<td>111101110</td>
</tr>
<tr>
<td>‘K’</td>
<td>4782</td>
<td>111110111</td>
</tr>
<tr>
<td>‘L’</td>
<td>22030</td>
<td>10101</td>
</tr>
<tr>
<td>‘M’</td>
<td>15298</td>
<td>01000</td>
</tr>
<tr>
<td>‘N’</td>
<td>42380</td>
<td>110</td>
</tr>
<tr>
<td>‘O’</td>
<td>46499</td>
<td>1101</td>
</tr>
<tr>
<td>‘P’</td>
<td>9957</td>
<td>101001</td>
</tr>
<tr>
<td>‘Q’</td>
<td>667</td>
<td>1111011101</td>
</tr>
</tbody>
</table>

Figure 31.2: The resulting prefix code for the frequencies of Figure 31.1. Here, for the sake of simplicity of exposition, the code was constructed only for the A—Z characters.

The total encoded length of the message. That is we would like to compute a tree $T$ that minimizes

$$\text{cost}(T) = \sum_{i=1}^{n} f[i] \ast \text{len(code}(i)), \quad (31.1)$$

where code($i$) is the binary string encoding the $i$th character and len($s$) is the length (in bits) of the binary string $s$.

As a concrete example, consider Figure 31.1, 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.

```
M U  ⇒  M
     /    |
    U
```

Similarly, we can put together two subtrees.
31.1. The algorithm to build Hoffman’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 31.2 and Figure 31.3.

31.1.2. Analysis

**Lemma 31.1.** 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. (31.1).
As for the second claim, consider a leaf \( u \) with maximum depth \( d \) in \( \mathcal{T} \), and consider it parent \( v = \overline{p}(u) \). 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 31.1.2.** 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 31.1.1.

Now, suppose those two leaves are not \( x \) and \( y \), but some other characters \( \alpha \) and \( \beta \). Let \( \mathcal{T}' \) be the code tree obtained by swapping \( x \) and \( \alpha \). The depth of \( x \) increases by some amount \( \Delta \), and the depth of \( \alpha \) 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 \( \alpha \) is not, which implies that \( f[\alpha] > f[x] \). Thus, swapping \( x \) and \( \alpha \) does not increase the total cost of the code. Since \( \mathcal{T} \) was an optimal code tree, swapping \( x \) and \( \alpha \) 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 \( \beta \) must give yet another optimal code tree. In this final optimal code tree, \( x \) and \( y \) as maximum-depth siblings, as required.

**Theorem 31.1.3.** 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 \( f[1 \ldots n] \) 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 \( f[1..n] \) 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 it leaves. We remain with a tree \( \mathcal{T}_{opt}' \) that has as leaves the characters \( 3, \ldots, n \) 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

\[
\text{cost}(\mathcal{T}_{opt}) = \sum_{i=1}^{n} f[i]\text{depth}_{\mathcal{T}_{opt}}(i) = \sum_{i=3}^{n+1} f[i]\text{depth}_{\mathcal{T}_{opt}}(i) + f[1]\text{depth}_{\mathcal{T}_{opt}}(1) + f[2]\text{depth}_{\mathcal{T}_{opt}}(2) = \text{cost}(\mathcal{T}_{opt}') + (f[1] + f[2])\text{depth}(\mathcal{T}_{opt}) - (f[1] + f[2])\left(\text{depth}(\mathcal{T}_{opt}) - 1\right) = \text{cost}(\mathcal{T}_{opt}') + f[1] + f[2].
\]

This implies that minimizing the cost of \( \mathcal{T}_{opt} \) is equivalent to minimizing the cost of \( \mathcal{T}_{opt}' \). In particular, \( \mathcal{T}_{opt}' \) must be an optimal coding tree for \( f[3 \ldots n + 1] \). Now, consider the Huffman tree \( \mathcal{T}_H \) constructed for \( f[3, \ldots, n + 1] \) and the overall Huffman tree \( \mathcal{T}_H \) constructed for \( f[1, \ldots, n] \). 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 \( f[3, \ldots, n + 1] \) is optimal; namely, \( \text{cost}(\mathcal{T}'_{opt}) = \text{cost}(\mathcal{T}'_H) \). As such, arguing as above, we have

\[
\]

by Eq. (31.2). Namely, the Huffman tree has the same cost as the optimal tree.

**31.1.3. 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.

(This 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.)
31.1.4. 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 31.1.4.** Let \( 1, \ldots, n \) 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

\[
\mathbb{H}(X) = \sum_i \Pr[X = i] \lg \frac{1}{\Pr[X = i]},
\]

which is the **entropy** of \( X \).

---

**Chapter 32**

**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 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.

### 32.1. Entropy

**Definition 32.1.1.** The **entropy** in bits of a discrete random variable \( X \) is given by

\[
\mathbb{H}(X) = -\sum_x \Pr[X = x] \lg \Pr[X = x].
\]
The binary entropy function $H(p)$ for a random binary variable that is 1 with probability $p$, is $H(p) = -p \log p - (1 - p) \log(1 - p)$. We define $H(0) = H(1) = 0$. See Figure 32.1.

The function $H(p)$ is a concave symmetric around $1/2$ on the interval $[0, 1]$ and achieves its maximum at $1/2$. For a concrete example, consider $H(3/4) \approx 0.8113$ and $H(7/8) \approx 0.5436$. 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 $H'(p) = -\log p + \log(1 - p) = \log \frac{1 - p}{p}$ and $H''(p) = \frac{p}{1 - p} \left( -\frac{1}{p^2} \right) = -\frac{1}{p(1 - p)}$. Thus, $H''(p) \leq 0$, for all $p \in (0, 1)$, and the $H(\cdot)$ is concave in this range. Also, $H'(1/2) = 0$, which implies that $H(1/2) = 1$ is a maximum of the binary entropy. Namely, a balanced coin has the largest amount of randomness in it.

Example 32.1.2. A random variable $X$ that has probability $1/n$ to be $i$, for $i = 1, \ldots, n$, has entropy $H(X) = -\sum_{i=1}^{n} \frac{1}{n} \log \frac{1}{n} = \log n$.

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 32.1.3. Let $X$ and $Y$ be two independent random variables, and let $Z$ be the random variable $(X, Y)$. Then $H(Z) = H(X) + H(Y)$.

Proof: In the following, summation are over all possible values that the variables can have. By the independence of $X$
and $Y$ we have
\[
\mathbb{H}(Z) = \sum_{x,y} \Pr[X, Y = (x, y)] \frac{1}{\Pr(X, Y = (x, y))} \\
= \sum_{x,y} \Pr[X = x] \Pr[Y = y] \frac{1}{\Pr[X = x] \Pr[Y = y]} \\
= \sum_x \sum_y \Pr[X = x] \Pr[Y = y] \frac{1}{\Pr[X = x]} \\
+ \sum_y \sum_x \Pr[X = x] \Pr[Y = y] \frac{1}{\Pr[Y = y]} \\
= \sum_x \Pr[X = x] \frac{1}{\Pr[X = x]} + \sum_y \Pr[Y = y] \frac{1}{\Pr[Y = y]} \\
= \mathbb{H}(X) + \mathbb{H}(Y).
\]

\[\text{Lemma 32.1.4. Suppose that } nq \text{ is integer in the range } [0, n]. \text{ Then } \frac{2^n \mathbb{H}(q)}{n + 1} \leq \binom{n}{nq} \leq 2^n \mathbb{H}(q).\]

\[\text{Proof: This trivially holds if } q = 0 \text{ or } q = 1, \text{ so assume } 0 < q < 1. \text{ We know that } \]
\[
\binom{n}{nq} q^{nq} (1 - q)^{n-nq} \leq (q + (1-q))^n 1.
\]

As such, since $q^{-nq}(1 - q)^{-(1-q)n} = 2^n (-q \log q - (1-q) \log(1-q)) = 2^n \mathbb{H}(q)$, 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
\[
\binom{n}{nq} \geq \frac{1}{n+1} q^{-nq}(1 - q)^{-(n-nq)} = \frac{1}{n+1} 2^n \mathbb{H}(q).
\]

\[\text{Lemma 32.1.4 can be extended to handle non-integer values of } q. \text{ This is straightforward, and we omit the easy but tedious details.}\]
Corollary 32.1.5. We have:

(i) $q \in [0, 1/2] \Rightarrow \left( \frac{n}{nq} \right) \leq 2^{n\mathbb{H}(q)}$. (ii) $q \in [1/2, 1] \Rightarrow \left( \frac{n}{nq} \right) \leq 2^{n\mathbb{H}(q)}$.

(iii) $q \in [1/2, 1] \Rightarrow \frac{2^{n\mathbb{H}(q)}}{n+1} \leq \left( \frac{n}{nq} \right)$. (iv) $q \in [0, 1/2) \Rightarrow \frac{2^{n\mathbb{H}(q)}}{n+1} \leq \left( \frac{n}{nq} \right)$

The bounds of Lemma 32.1.4 and Corollary 32.1.5 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 $\left( \frac{n}{nq} \right) \approx 2^{n\mathbb{H}(q)}$ possible sequences that have similar probability. As such, $\mathbb{H}(Y) \approx \lg \left( \frac{n}{nq} \right) = n\mathbb{H}(q)$, by Example 32.1.2, a fact that we already know from Lemma 32.1.3.

32.1.1. Extracting randomness

Entropy can be interpreted as the amount of unbiased random coin flips can be extracted from a random variable.

Definition 32.1.6. An extraction function $\text{Ext}$ takes as input the value of a random variable $X$ and outputs a sequence of bits $y$, such that $Pr[\text{Ext}(X) = y \mid |y| = k] = \frac{1}{2^k}$, whenever $Pr[|y| = k] > 0$, where $|y|$ denotes the length of $y$.

As a concrete (easy) example, consider $X$ to be a uniform random integer variable out of $0, \ldots, 7$. All that $\text{Ext}(X)$ has to do in this case, is to compute the binary representation of $x$. However, note that Definition 32.1.6 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 $0, \ldots, 11$, the function $\text{Ext}(x)$ 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 32.1.6 holds for this extraction function, since $Pr[\text{Ext}(X) = 00 \mid |\text{Ext}(X)| = 2] = \frac{1}{4}$, as required. This scheme can be of course extracted for any range.

The following is obvious, but we provide a proof anyway.

Lemma 32.1.7. Let $x/y$ be a faction, 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 size is equal to $i(x - y)$, but since $y > x$ (as $x/y < 1$), this quantity is negative, as required. •

Theorem 32.1.8. Suppose that the value of a random variable $X$ is chosen uniformly at random from the integers $\{0, \ldots, m-1\}$, then there is an extraction function for $X$ that outputs on average at least $[\lg m] - 1 = [\mathbb{H}(X)] - 1$ 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 $\{0, \ldots, m-1\}$ 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 32.1.6.

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 $u = \left[ \lg(m - 2^k) \right] < k$. 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, $2^k + 2^u > m$, 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 32.1.7, since $\frac{m-2^k}{m} < 1$, we have

$$\frac{m - 2^k}{m} \leq \frac{m - 2^k + \left(2^{u+1} + 2^k - m\right)}{m} = \frac{2^{u+1}}{2^{u+1} + 2^k}.$$
Thus, by induction (we assume the claim holds for all integers smaller than \( m \)), we have

\[
E[Y] \geq \frac{2^k}{m} k + \frac{m - 2^k}{m} \left( \left\lfloor \frac{\lg (m - 2^k)}{m} \right\rfloor - 1 \right) = \frac{2^k}{m} k + \frac{m - 2^k}{m} \left( k - k + u - 1 \right)
\]

\[
= k + \frac{m - 2^k}{m} (u - k - 1)
\]

\[
\geq k + \frac{2^{u+1}}{2^{u+1} + 2^k} (u - k - 1) = k - \frac{2^{u+1}}{2^{u+1} + 2^k} (1 + k - u),
\]

since \( u - k - 1 \leq 0 \) as \( k > u \). If \( u = k - 1 \), then \( E[Y] \geq k - \frac{1}{2} \cdot 2 = k - 1 \), as required. If \( u = k - 2 \) then \( E[Y] \geq k - \frac{1}{3} \cdot 3 = k - 1 \). Finally, if \( u < k - 2 \) then

\[
E[Y] \geq k - \frac{2^{u+1}}{2^k} (1 + k - u) = k - \frac{k - u + 1}{2^{k-u-1}} = k - \frac{2 + (k - u - 1)}{2^{k-u-1}} \geq k - 1,
\]

since \( (2 + i)/2^i \leq 1 \) for \( i \geq 2 \).

Chapter 33

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 Assualtman 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.

33.1. Extracting randomness

33.1.1. 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 \( 0, \ldots, \binom{n}{j} - 1 \).

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 \( \pi_v \) from the root of \( \mathcal{T} \) to \( v \) is of length \( k \). Furthermore, assume there are \( m \) ones written on the path \( \pi_v \). 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 $0, \ldots, \binom{n}{j} - 1$, and this mapping is one-to-one. Let $\text{EnumBinomCoeffAlg}$ denote this procedure.

### 33.1.2. Extracting randomness

**Theorem 33.1.1.** 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 $S_j = \{0, \ldots, \binom{n}{j} - 1\}$. Note, that this, conditional distribution on $j$, is uniform on this set, and we can apply the extraction algorithm of Theorem 32.1.8 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

$$E[B] = \sum_{k=0}^{n} \Pr[Z = k] E[B \mid Z = k],$$

and by Theorem 32.1.8, we have $E[B \mid Z = k] \geq \left\lceil \log \binom{n}{k} \right\rceil - 1$. Let $\epsilon < p - 1/2$ be a constant to be determined shortly. For $n(p - \epsilon) \leq k \leq n(p + \epsilon)$, we have

$$\binom{n}{k} \geq \left( \frac{n}{\lceil n(p + \epsilon) \rceil} \right)^{\lceil n(p + \epsilon) \rceil} \geq \frac{2^{n \mathbb{H}(p + \epsilon)}}{n + 1},$$

by Corollary 32.1.5 (iii). We have

$$E[B] \geq \sum_{k=n(p - \epsilon)}^{\lceil n(p + \epsilon) \rceil} \Pr[Z = k] E[B \mid Z = k] \geq \sum_{k=n(p - \epsilon)}^{\lceil n(p + \epsilon) \rceil} \Pr[Z = k] \left( \left\lceil \log \binom{n}{k} \right\rceil - 1 \right)$$

$$\geq \sum_{k=n(p - \epsilon)}^{\lceil n(p + \epsilon) \rceil} \Pr[Z = k] \left( \log \frac{2^{n \mathbb{H}(p + \epsilon)}}{n + 1} - 2 \right)$$

$$= \left( n \mathbb{H}(p + \epsilon) - \log(n + 1) - 2 \right) \Pr[Z - np] \leq \epsilon n$$

$$\geq \left( n \mathbb{H}(p + \epsilon) - \log(n + 1) - 2 \right) \left( 1 - 2 \exp \left( -\frac{n \epsilon^2}{4p} \right) \right),$$

since $\mu = E[Z] = np$ and $\Pr[Z - np] \geq \frac{\epsilon}{p} np \leq 2 \exp \left( -\frac{np}{2} \left( \frac{\epsilon}{p} \right)^2 \right) = 2 \exp \left( -\frac{n \epsilon^2}{4p} \right)$, by the Chernoff inequality. In particular, fix $\epsilon > 0$, such that $\mathbb{H}(p + \epsilon) > (1 - \delta/4) \mathbb{H}(p)$, and since $p$ is fixed $n \mathbb{H}(p) = \Omega(n)$, in particular, for $n$ sufficiently large, we have $-\log(n + 1) \geq -\frac{\delta}{4} n \mathbb{H}(p)$. Also, for $n$ sufficiently large, we have $2 \exp \left( -\frac{n \epsilon^2}{4p} \right) \leq \frac{\delta}{10}$. Putting it together, we have that for $n$ large enough, we have

$$E[B] \geq \left( 1 - \frac{\delta}{4} - \frac{\delta}{10} \right) n \mathbb{H}(p) \left( 1 - \frac{\delta}{10} \right) \geq (1 - \delta)n \mathbb{H}(p),$$
as claimed.

As for the upper bound, observe that if an input sequence \( x \) has probability \( \Pr[X = x] \), then the output sequence \( y = \text{Ext}(x) \) has probability to be generated which is at least \( \Pr[X = x] \). Now, all sequences of length \(|y|\) have equal probability to be generated. Thus, we have the following (trivial) inequality

\[
2^{|\text{Ext}(x)|} \Pr[X = x] \leq 2^{|\text{Ext}(x)|} \Pr[y = \text{Ext}(x)] \leq 1,
\]

implying that \(|\text{Ext}(x)| \leq \lg(1/\Pr[X = x])\). Thus,

\[
E[B] = \sum_x \Pr[X = x] |\text{Ext}(x)| \leq \sum_x \Pr[X = x] \lg \frac{1}{\Pr[X = x]} = H(X).
\]

33.2. Bibliographical Notes

The presentation here follows [MU05, Sec. 9.1-Sec 9.3].

Chapter 34

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.

34.1. 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 34.1.1. The input to a binary symmetric channel with parameter \( p \) is a sequence of bits \( x_1, x_2, \ldots, \) and the output is a sequence of bits \( y_1, y_2, \ldots, \) such that \( \Pr[x_i = y_i] = 1 - p \) 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, it’s 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 34.1.2. A \((k, n)\) encoding function \(\text{Enc} : \{0, 1\}^k \to \{0, 1\}^n\) takes as input a sequence of \(k\) bits and outputs a sequence of \(n\) bits. A \((k, n)\) decoding function \(\text{Dec} : \{0, 1\}^n \to \{0, 1\}^k\) 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\(^{\text{iii}}\) in 1948 states exactly how much information can be sent on such a channel.

**Theorem 34.1.3 (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 \(\gamma\) 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 \(\gamma\) for a \(k\)-bit input message chosen uniformly at random.

### 34.1.0.1. Intuition behind Shannon’s theorem

Let assume the senders has sent a string \(S = s_1s_2\ldots s_n\). The receiver got a string \(T = t_1t_2\ldots t_n\), where \(p = \Pr[t_i \neq s_i]\), for all \(i\). In particular, let \(U\) be the Hamming distance between \(S\) and \(T\); that is, \(U = \sum s_i \neq t_i\). Under our assumptions \(E[U] = pn\), and \(U\) is a binomial variable. By Chernoff inequality, we know that \(U \in [(1 - \delta)np, (1 + \delta)np]\) with high probability, where \(\delta\) 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

\[
\sum_{i = (1 - \delta)np}^{(1 + \delta)np} \binom{n}{i} \leq 2 \binom{n}{(1 + \delta)np} \leq \alpha = 2 \cdot 2^{nH((1 + \delta)p)}.
\]

Let us pick as many rings as possible in the hypercube so that they are disjoint: \(R_1, \ldots, R_{\kappa}\). 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 \(\kappa\) we would pick would be at least

\[
\kappa \geq \frac{2^n}{|R|} \geq \frac{2^n}{2 \cdot 2^{nH((1 + \delta)p)}} \approx 2^{n(1 - H((1 + \delta)p))}.
\]

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

\[
k = \lfloor \log \kappa \rfloor = n(1 - H((1 + \delta)p)) = n(1 - H(p)),
\]

as desired.

### 34.1.0.2. 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.

---

\(^{\text{iii}}\)Claude Elwood Shannon (April 30, 1916 - February 24, 2001), an American electrical engineer and mathematician, has been called “the father of information theory”.

202
34.2. Proof of Shannon’s theorem

The proof is not hard, but requires some care, and we will break it into parts.

34.2.1. How to encode and decode efficiently

34.2.1.1. The scheme

Our scheme would be simple. Pick \( k \leq n(1 - \frac{H(p)}{n}) - \delta \). For any number \( i = 0, \ldots, \hat{K} = 2^{k+1} - 1 \), randomly generate a binary string \( Y_i \) made out of \( n \) bits, each one chosen independently and uniformly. Let \( Y_0, \ldots, Y_{\hat{K}} \) denote these code words. Here, we have \( \hat{K} = 2^{n(1 - H(p) - \delta)} \).

For each of these codewords we will compute the probability that if we send this codeword, the receiver would fail. Let \( X_0, \ldots, X_K \), 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.

34.2.1.2. 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 \( Y_0, \ldots, Y_{\hat{K}} \)). 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

\[
W_i = \sum_{r \notin S_i} \Pr[r \text{ received when } X_i \text{ was sent}].
\]

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

\[
\Pr[r \text{ received when } X_i \text{ was sent}] = w(X_i, r).
\]

Let \( S_i \) be an indicator variable which is 1 if \( r \notin S_i \). We have that

\[
W_i = \sum_{r \notin S_i} \Pr[r \text{ received when } X_i \text{ was sent}] = \sum_{r \notin S_i} w(X_i, r) = \sum_r S_i \cdot w(X_i, r).
\]

The value of \( W_i \) is a random variable of our choice of \( Y_0, \ldots, Y_{\hat{K}} \). As such, its natural to ask what is the expected value of \( W_i \).

Consider the ring

\[
R(r) = \left\{ x \left| (1 - \varepsilon)n p \leq \Delta(x, r) \leq (1 + \varepsilon)n p \right. \right\},
\]

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 34.2.1. 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

$$\tau = \Pr\left[ r \notin S_i \left| r \in R(Y_i) \right. \right] \leq \frac{\gamma}{8},$$

where $\gamma$ is the parameter of Theorem 34.1.3.

Proof: The decoder fails here, only if $R(r)$ contains some other codeword $Y_j (j \neq i)$ in it. As such,

$$\tau = \Pr\left[ r \notin S_i \left| r \in R(Y_i) \right. \right] \leq \Pr\left[ Y_j \in R(r), \text{ for any } j \neq i \right] \leq \sum_{j \neq i} \Pr\left[ Y_j \in R(r) \right].$$

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

$$\Pr\left[ Y_j \in R(r) \right] = \sum_{m=(1-\varepsilon)n^p}^{(1+\varepsilon)n^p} \frac{n^m}{2^n} \left( \frac{n}{2} \right)^{\left( 1+\varepsilon \right)n^p},$$

since $(1+\varepsilon)p < 1/2$ (for $\varepsilon$ sufficiently small), and as such the last binomial coefficient in this summation is the largest. By Corollary 32.1.5 (i), we have

$$\Pr\left[ Y_j \in R(r) \right] \leq \frac{n}{2^n} \left( \frac{n}{2} \right)^{\left( 1+\varepsilon \right)n^p} = n2^{n(1+\varepsilon)p-1}.$$

As such, we have

$$\tau = \Pr\left[ r \notin S_i \left| r \in R(Y_i) \right. \right] \leq \sum_{j \neq i} \Pr\left[ Y_j \in R(r) \right] \leq K \Pr\left[ Y_1 \in R(r) \right] \leq 2^{k+1} \cdot n2^{n(1+\varepsilon)p-1} \leq n2^{n(1-\mathbb{H}(\varepsilon p) - \mathbb{H}(\varepsilon p) - \delta) + \mathbb{H}(\varepsilon p) - \delta)} + 1$$

since $k \leq n(1 - \mathbb{H}(\varepsilon p) - \delta)$. Now, we choose $\varepsilon$ to be a small enough constant, so that the quantity $\mathbb{H}(1+\varepsilon p) - \mathbb{H}(\varepsilon p) - \delta$ is equal to some (absolute) negative (constant), say $-\beta$, where $\beta > 0$. Then, $\tau \leq n2^{\beta+1}$, and choosing $n$ large enough, we can make $\tau$ smaller than $\gamma/2$, as desired. As such, we just proved that

$$\tau = \Pr\left[ r \notin S_i \left| r \in R(Y_i) \right. \right] \leq \frac{\gamma}{2}.$$  ■

Lemma 34.2.2. We have, that $\sum_{r \in R(Y_i)} w(Y_i, r) \leq \gamma/8$, where $\gamma$ is the parameter of Theorem 34.1.3.

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 $\varepsilon pn$ bits were flipped. But this quantity can be bounded using the Chernoff inequality. Let $Z = \Delta(Y_i, r)$, and observe that $E[Z] = pn$, and it is the sum of $n$ independent indicator variables. As such

$$\sum_{r \in R(Y_i)} w(Y_i, r) = \Pr\left[ |Z - E[Z]| > \varepsilon pn \right] \leq 2 \exp\left( -\frac{\varepsilon^2}{4} pn \right) < \frac{\gamma}{4},$$

since $\varepsilon$ is a constant, and for $n$ sufficiently large.  ■

Lemma 34.2.3. For any $i$, we have $\mu = E[W_i] \leq \gamma/4$, where $\gamma$ is the parameter of Theorem 34.1.3.
Proof: By linearity of expectations, we have
\[
\mu = E[W_i] = E\left[\sum_r S_{i,r}w(Y_i, r)\right] = \sum_r E[S_{i,r}w(Y_i, r)] \\
= \sum_r E[S_{i,r}] w(Y_i, r) = \sum_r \Pr[x \notin S_i] w(Y_i, r),
\]

since \(S_{i,r}\) is an indicator variable. Setting, \(\tau = \Pr[r \notin S_i \mid r \in R(Y_i)]\) and since \(\sum_r w(Y_i, r) = 1\), we get
\[
\mu = \sum_{r \in R(Y_i)} \Pr[x \notin S_i \mid r \in R(Y_i)] w(Y_i, r) + \sum_{r \notin R(Y_i)} \Pr[x \notin S_i] w(Y_i, r) \\
= \sum_{r \in R(Y_i)} \Pr[x \notin S_i \mid r \in R(Y_i)] w(Y_i, r) + \sum_{r \notin R(Y_i)} \Pr[x \notin S_i] w(Y_i, r) \\
\leq \sum_{r \in R(Y_i)} \tau \cdot w(Y_i, r) + \sum_{r \notin R(Y_i)} w(Y_i, r) \leq \tau + \sum_{r \notin R(Y_i)} w(Y_i, r) \leq \frac{\gamma}{4} + \frac{\gamma}{4} = \frac{\gamma}{2}.
\]

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,
\[
\tau = \Pr[r \notin S_i \mid r \in R(Y_i)] \leq \Pr[Y_j \in R(r), \text{ for any } j \neq i] \leq \sum_{j \neq i} \Pr[Y_j \in R(r)].
\]

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
\[
\Pr[Y_j \in R(r)] = \sum_{m=(1+\varepsilon)n p}^{\binom{n}{r}} \binom{n}{m} \leq \frac{n}{2^n} \binom{n}{(1+\varepsilon)n p}.
\]

since \((1+\varepsilon)p < 1/2\) (for \(\varepsilon\) sufficiently small), and as such the last binomial coefficient in this summation is the largest. By Corollary 32.1.5 (i), we have
\[
\Pr[Y_j \in R(r)] \leq \frac{n}{2^n} \binom{n}{(1+\varepsilon)n p} \leq \frac{n}{2^n} 2^{nH((1+\varepsilon)p)} = n 2^{nH((1+\varepsilon)p) - 1)}.
\]

As such, we have
\[
\tau = \Pr[r \notin S_i \mid r \in R(Y_i)] \leq \sum_{j \neq i} \Pr[Y_j \in R(r)] \cdot \leq K \Pr[Y_1 \in R(r)] \leq 2^{k+1} n 2^{nH((1+\varepsilon)p) - 1)}
\]
\[
\leq n 2^{(1-H(p) - \delta) + 1 + nH((1+\varepsilon)p) - 1)} \leq n 2^{(H((1+\varepsilon)p) - H(p) - \delta) + 1}
\]
since \(k \leq n(1 - H(p) - \delta)\). Now, we choose \(\varepsilon\) to be a small enough constant, so that the quantity \(H((1+\varepsilon)p) - H(p) - \delta\) is negative (constant). Then, choosing \(n\) large enough, we can make \(\tau\) smaller than \(\gamma/2\), as desired. As such, we just proved that
\[
\tau = \Pr[r \notin S_i \mid r \in R(Y_i)] \leq \frac{\gamma}{2}.
\]

In the following, we need the following trivial (but surprisingly deep) observation.

**Observation 32.4.** For a random variable \(X\), if \(E[X] \leq \psi\), then there exists an event in the probability space, that assigns \(X\) a value \(\leq \mu\).

This holds, since \(E[X]\) 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.

205
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 34.2.5.** For the codewords \(X_0, \ldots, X_K\), the probability of failure in recovering them when sending them over the noisy channel is at most \(\gamma\).

*Proof:* We just proved that when using \(Y_0, \ldots, Y_{\tilde{K}}\), the expected probability of failure when sending \(Y_i\) is \(\mathbb{E}[W_i] \leq \gamma\), where \(\tilde{K} = 2^{k+1} - 1\). As such, the expected total probability of failure is

\[
\mathbb{E}\left[\sum_{i=0}^{\tilde{K}} W_i\right] = \sum_{i=0}^{\tilde{K}} \mathbb{E}[W_i] \leq \frac{\gamma}{2} 2^{k+1} = \gamma 2^k,
\]

by Lemma 34.2.3 (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 34.2.4, there exist a choice of \(Y_i\)s, such that

\[
\sum_{i=0}^{\tilde{K}} W_i \leq 2^k \gamma.
\]

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 \(\gamma\), because in the summation, we will get that

\[
\sum_{i=0}^{\tilde{K}} W_i > 2^k \gamma.
\]

Which is a contradiction. As such, there are \(2^k\) codewords with failure probability smaller than \(\gamma\). 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 \(\gamma\).

**Lemma 34.2.5** concludes the proof of the constructive part of Shannon’s theorem.

### 34.2.2. Lower bound on the message size

We omit the proof of this part.

### 34.3. Bibliographical Notes

The presentation here follows [MU05, Sec. 9.1-Sec 9.3].

---

**Chapter 35**

**Exercises - Entropy**

### 35.0.1. Compress a sequence.

We wish to compress a sequence of independent, identically distributed random variables \(X_1, X_2, \ldots\). Each \(X_j\) takes on one of \(n\) values. The \(i\)th value occurs with probability \(p_i\), where \(p_1 \geq p_2 \geq \ldots \geq p_n\). 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 \([\lg(1/p_i)]\) bits of \( T_i \). Start with an empty string, and consider \( xj \) in order. If \( X_j \) takes on the \( i \)th value, append the \( i \)th codeword to the end of the string.

(A) Show that no codeword is the prefix of any other codeword.

(B) Let \( Z \) be the average number of bits appended for each random variable \( X_j \). Show that

\[ H(X_j) \leq z \leq H(X_j) + 1. \]

35.0.2. 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 \( X = (X_1, X_2, \ldots, X_n) \), where each \( X_i \) is independently 0 with probability \( p \) and 1 with probability \( 1 - p \). The sequences can be ordered lexicographically, so for \( x = (x_1, x_2, \ldots, x_n) \) and \( y = (y_1, y_2, \ldots, y_n) \), 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

\[ q(x) = \sum_{y < x} p(y). \]

(A) Suppose we are given \( X = (X_1, X_2, \ldots, X_n) \). Explain how to compute \( q(X) \) in time \( O(n) \) (assume that any reasonable operation on real numbers takes constant time).

(B) Argue that the intervals \([q(x), q(x) + p(x))\] are disjoint subintervals of \([0, 1)\).

(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 \( [\lg(1/p(X))] + 1 \) binary decimal digits to represent \( X \) in such a way that no codeword is the prefix of any other codeword.

(D) Given a codeword chosen as in (C), explain how to decompress it to determine the corresponding sequence \((X_1, X_2, \ldots, X_n)\).

(E) Using the Chernoff inequality, argue that \( \lg(1/p(X)) \) is close to \( nH(p) \) with high probability. Thus, this approach yields an effective compression scheme.

35.0.3. Computing entropy.

1. Let \( S = \sum_{i=1}^{10} 1/i^2 \). Consider a random variable \( X \) such that \( \Pr[X = i] = 1/(S i^2) \), for \( i = 1, \ldots, 10 \). Compute \( H(X) \).
2. Let \( S = \sum_{i=1}^{10} 1/i^3 \). Consider a random variable \( X \) such that \( \Pr[X = i] = 1/(S i^3) \), for \( i = 1, \ldots, 10 \). Compute \( H(X) \).
3. Let \( S(\alpha) = \sum_{i=1}^{10} 1/i^\alpha \), for \( \alpha > 1 \). Consider a random variable \( X \) such that \( \Pr[X = i] = 1/(S(\alpha) i^\alpha) \), for \( i = 1, \ldots, 10 \). Prove that \( H(X) \) is either increasing or decreasing as a function of \( \alpha \) (you can assume that \( \alpha \) is an integer).

35.0.4. When is entropy maximized?

Consider an \( n \)-sided die, where the \( j \)th face comes up with probability \( p_j \). Show that the entropy of a die roll is maximized when each face comes up with equal probability \( 1/n \).
35.0.5. Condition entropy.

The **conditional entropy** $H(Y|X)$ is defined by

$$H(Y|X) = \sum_{x,y} \Pr[(X = x) \cap (Y = y)] \log \frac{1}{\Pr[Y = y|X = x]}.$$  

If $Z = (X, Y)$, prove that

$$H(Z) = H(X) + H(Y|X).$$

35.0.6. 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 $\{0, \ldots, m - 1\}$. It follows that if we have $k$ numbers chosen independently and uniformly at random from $\{0, \ldots, m - 1\}$ 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.

35.0.7. Kraft inequality.

Assume you have a (valid) prefix code with $n$ codewords, where the $i$th codeword is made out of $\ell_i$ bits. Prove that

$$\sum_{i=1}^{n} \frac{1}{2^{\ell_i}} \leq 1.$$
Chapter 36

Matchings

36.1. Definitions and basic properties

36.1.1. Definitions

Definition 36.1.1. 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 36.1.2. 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 36.1.3. 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.

Definition 36.1.4. 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.

Problem 36.1.5. Given a graph \( G \) and a weight function on the edges, compute the maximum weight matching in \( G \).

Remark 36.1.6. 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.

36.1.2. 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.
Figure 36.1: (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.

Definition 36.1.7. A path $\pi = v_1v_2, \ldots, v_{2k+2}$ is an augmenting path for a matching $M$ in a graph $G$:
(i) $\pi$ 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) $e_1, e_3, \ldots, e_{2k+1} \notin M$, and
(v) $e_2, e_4, \ldots, e_{2k} \in M$.

Lemma 36.1.8. If $M$ is a matching and $\pi$ 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 $\pi$ from the graph all together. What is left of $M$, is a matching of size $|M| - |M \cap \pi|$. Now, add back $\pi$ and alternate the edges of the matching $M$ with the free edges of $\pi$. Clearly, the new set of edges is a matching, since $\pi$ 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 36.1.9. Let $M$ be a matching, and $T$ be a maximum matching, and $k = |T| - |M|$. Then $M$ has $k$ 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 $H = (V, E')$. 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 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).

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 (vertex disjoint) augmenting paths were of length $\geq u/k$ then the total number of vertices in $H$ would be at least $(u/k + 1)u > u$, since a path of length $\ell$ has $\ell + 1$ vertices. A contradiction.

The lemma readily implies:

Corollary 36.1.10. A matching $M$ is maximum $\iff$ there is no augmenting path for $M$. 

210
36.2. Unweighted matching in bipartite graph

36.2.1. The slow algorithm; \texttt{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 \texttt{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 \texttt{BFS}, compute the shortest path $\pi_i$ from a vertex of $L_i$ to a vertex of $R_i$. If there is no such path, the algorithm stops and reports that the current matching as 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 36.1.10, 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, the algorithm can be easily implemented in linear time. We thus have the following result:

\textbf{Lemma 36.2.1.} 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.

36.2.2. The Hopcroft-Karp algorithm

We next improve the running time – this requires quite a bit of work, but hopefully exposes some interesting properties of matching in bipartite graphs.

36.2.2.1. 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 36.2.2 below.

(B) As such, if we always augment along shortest paths, then the augmenting paths get longer as the algorithm progress, see Corollary 36.2.3 below.

(C) Furthermore, all the augmenting paths of the same length used by the algorithm are vertex-disjoint (!). See Lemma 36.2.4 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.)

\textbf{Lemma 36.2.2.} Let $M$ be a matching, and $\pi$ be the shortest augmenting path for $M$, and let $\pi'$ be any augmenting path for $M' = M \oplus \pi$. Then $|\pi'| \geq |\pi|$. Specifically, we have $|\pi'| \geq |\pi| + 2 |\pi \cap \pi'|$.

\textbf{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 $\sigma_1$ and $\sigma_2$ (relative to $M$). Observe that $M \oplus N = \pi \oplus \pi'$, and as such

$$|\pi \oplus \pi'| = |M \oplus N| \geq |\sigma_1| + |\sigma_2|.$$
Since \( \pi \) was the shortest augmenting path for \( M \), it follows that \( |\sigma_1| \geq |\pi| \) and \( |\sigma_2| \geq |\pi| \). We such,

\[
|\pi \oplus \pi'| \geq |\sigma_1| + |\sigma_2| \geq |\pi| + |\pi| = 2|\pi|.
\]

By definition, we have that \( |\pi \oplus \pi'| = |\pi| + |\pi'| - 2|\pi \cap \pi'| \). Combining with the above, we have

\[
|\pi| + |\pi'| - 2|\pi \cap \pi'| \geq 2|\pi| \quad \Rightarrow \quad |\pi'| \geq |\pi| + 2|\pi \cap \pi'|.
\]

The above lemma immediately implies the following.

**Corollary 36.2.3.** For the sequence of augmenting paths used by the algorithm of Section 36.2.1 (which we remind the reader always augment the matching along the shortest augmenting path), we have that \( |\pi_1| \leq |\pi_2| \leq \ldots \leq |\pi_t| \), where \( t \) is the number of augmenting paths computed by the algorithm, and \( \pi_1, \pi_2, \ldots, \pi_t \) are the sequence of augmenting paths used by the algorithm.

**Lemma 36.2.4.** For all \( i \) and \( j \), such that \( |\pi_i| = \cdots = |\pi_j| \), we have that the paths \( \pi_i \) and \( \pi_j \) are vertex disjoint.

**Proof:** Assume for contradiction, that that \( |\pi_i|=|\pi_j|, i<j, \) and \( \pi_i \) and \( \pi_j \) are not vertex disjoint, and assume that \( j-i \) is minimal. As such, for any \( k \), such that \( i<k<j \), we have that \( \pi_k \) is disjoint from \( \pi_i \) and \( \pi_j \).

Now, let \( M_i \) be the matching after \( \pi_i \) was applied. We have that \( \pi_j \) is not using any of the edges of \( \pi_{i+1}, \ldots, \pi_{j-1} \). As such, \( \pi_j \) is an augmenting path for \( M_i \). Now, \( \pi_j \) and \( \pi_i \) share vertices. It definitely cannot be the two endpoints of \( \pi_j \) (since they are free) - so it must be some interval vertex of \( \pi_j \). But then, \( \pi_i \) and \( \pi_j \) 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 \( \pi_j \) - since it contains the only matching edge adjacent to \( v \) in \( M_j \). Namely, \( |\pi_i \cap \pi_j| \geq 1 \). Now, by Lemma 36.2.2, we conclude that \( |\pi_j| \geq |\pi_i| + 2|\pi_i \cap \pi_j| > |\pi_i| \). A contradiction.

**36.2.2.2. 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 \( k = 1, 3, 5, \ldots \). Specifically, we apply this extraction algorithm, till \( k = 1 + 2\lfloor \sqrt{n} \rfloor \). 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 36.2.5 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.

**Analysis.**

**Lemma 36.2.5.** 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 \( \geq \sqrt{n} \). By Lemma 36.1.9, 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 \( n \geq 4 \).
36.2.2.3. 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 path. 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!)

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 $T$ be the resulting tree.

Let $L_1, R_1, L_2, R_2, L_3, \ldots$ be the layers of the BFS. By assumption, the first free vertex below $L_1$ encountered in the tree is of level $R_\tau$, 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 $T$. 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). 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 $T$). Compute also the reverse graph $J^{\text{rev}}$ (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 $J^{\text{rev}}$ 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 $F_\tau$, till we visit all such vertices.

This algorithm results in a collection of augmenting paths $P_\tau$, which are vertex disjoint. We claim that $P_\tau$ is the desired set maximal cardinality disjoint set of augmenting paths of length $k$.

**Analysis.** Building the initial graphs $J$ and $J^{\text{rev}}$ 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 $J^{\text{rev}}$ is going to be considered only once by the algorithm. As such, the running time of the algorithm is $O(n + m)$ as desired.

**Lemma 36.2.6.** 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 $\sigma$ of length $k$ that is disjoint from the vertices of the paths of $P_k$. But then, we could traverse $\sigma$ in $J$, and this would go through unused vertices. Indeed, if any of the vertices of $\sigma$ were used
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 36.3: Extracting augmenting paths from the reverse layered graph.
Figure 36.4: (A) A bipartite graph and its current matching. (B) Augmenting paths computed using the layered graph (see Figure ??). (C) The new matching after we apply the augmenting paths.

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 \(\sigma\) is supposedly disjoint from the paths of \(P_k\).

36.2.2.4. The result

**Theorem 36.2.7.** 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 \(\text{algMatching}_{HK}\) algorithm is described in Section 36.2.2.2, 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 36.2.1. Indeed, the \(\text{algMatching}_{HK}\) algorithm computes a sequence of sets of augmenting paths \(P_1, P_3, P_5, \ldots\). 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 36.2.6 each set \(P_k\) contains a maximal set of such vertex-disjoint augmenting paths of length \(k\). By Lemma 36.2.4, all augmenting paths of length \(k\) computed are vertex disjoint.

As such, now by induction, we can argue that if \(\text{algMatching}_{HK}\) simulates correctly \(\text{algSlowMatch}\), for the augmenting paths in \(P_1 \cup P_3 \cup \ldots \cup P_i\), then it simulates it correctly for \(P_1 \cup P_3 \cup \ldots \cup P_i \cup P_{i+1}\), and we are done.

36.3. Bibliographical notes

The description here follows the original paper of Hopcroft and Karp [HK73].
Chapter 37

Matchings II

37.1. Maximum Weight Matchings in A Bipartite Graph

37.1.1. On the structure of the problem

For an alternating path/cycle $\pi$, its weight, in relation to a matching $M$, is

$$\gamma(\pi, M) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in \pi \cap M} w(e). \quad (37.1)$$

Namely, it is the total weight of the free edges in $\pi$ minus the weight of the matched edges. This is a natural concept because of the following lemma.

Lemma 37.1.1. Let $M$ be a matching, and let $\pi$ 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 $w(M')$ is bigger; namely, $w(M') > w(M)$.

Proof: We have that

$$w(M') - w(M) = \sum_{e \in M'} w(e) - \sum_{e \in M} w(e) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in M \setminus \pi} w(e) = \gamma(\pi, M).$$

Just observe that $w(M') = w(M) + \gamma(\pi, M)$. ■

Definition 37.1.2. An alternating path is augmenting if it starts and ends in a free vertex.

Observation 37.1.3. If $M$ has an augmenting path $\pi$ then $M$ is not of maximum size matching (this is for the unweighted case), since $M \oplus \pi$ is a larger matching.

Theorem 37.1.4. Let $M$ be a matching of maximum weight among matchings of size $|M|$. Let $\pi$ be an augmenting path for $M$ of maximum weight, and let $T$ be the matching formed by augmenting $M$ using $\pi$. 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 $\sigma$ in $H$. The weight $\gamma(\sigma, M)$ (see Eq. (37.1)) 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 $\sigma$ 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 $S$.  

216
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 \(\mu\) in \(U_S\) which is not paired, and it must be that \(\gamma(\mu, M) = w(S) - w(M)\) (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 \(\sigma'\) for \(M\) such that \(w(M \oplus \sigma') > w(S)\). 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.

### 37.2. Maximum Weight Matchings in A Bipartite 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 \(M\) we define the graph \(G_M\) to be the directed graph, where if \(rl \in M\), \(l \in L\) and \(r \in R\) then we add \((r, l)\) to \(E(G_M)\) with weight \(a((r, l)) = w(rl)\). Similarly, if \(rl \in E \setminus M\) then add the edge \((l \rightarrow r)\) to \(E(G_M)\) to \(G_M\) and set \(a((l, r)) = -w(rl)\).

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 \(\pi\) in \(G\) must have an odd number of edges. Since \(G\) is bipartite, \(\pi\) must have one endpoint on the left side and one endpoint on the right side. Observe, that a path \(\pi\) in \(G_M\) has weight \(a(\pi) = -\gamma(\pi, M)\).

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 \(\pi\) in \(G_M\) starting \(U_L\) going to \(U_R\) with maximum weight \(\gamma(\pi)\), namely with minimum weight \(a(\pi)\).

**Lemma 37.2.1.** If \(M\) is a maximum weight matching with \(k\) edges in \(G\), than there is no negative cycle in \(G_M\) where \(a(\cdot)\) is the associated weight function.

**Proof:** Assume for the sake of contradiction that there is a cycle \(C\), and observe that \(\gamma(C) = -a(C) > 0\). Namely, \(M \oplus C\) is a new matching with bigger weight and the same number of edges. A contradiction to the maximality of \(M\).

**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 just computing a shortest path in the graph \(G_M\) which does not have negative cycles, and this can just 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 37.2.2.** Given a bipartite graph \(G\) and a maximum weight matching \(M\) of size \(k\) one can find a maximum weight augmenting path for \(G\) 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 37.2.3.** 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.
37.2.1. Faster Algorithm

It turns out, in fact, that the graph here is very special, and one can use the Dijkstra algorithm. We omit any further details, and just state the result. The interested student can figure out the details (warning: this is not easy).

**Theorem 37.2.4.** 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.

37.3. 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 \( \infty \) (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 claim.

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.

37.4. 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 \( \tau_{xy} \) denote the path in \( \mathcal{T} \) between \( x \) and \( y \). For two paths \( \pi \) and \( \pi' \) that share an endpoint, let \( \pi \parallel \pi' \) denotes the path resulting from concatenating \( \pi \) to \( \pi' \). For a path \( \pi \), let \(|\pi|\) denote the number of edges in \( \pi \).

37.4.1. 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 $M$.

We start by collapsing the unmatched vertices to a single vertex $s$, and let $H$ be the resulting graph. 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 $T$ denote the resulting tree.

An augmenting path in $G$ corresponds to an odd cycle in $H$ with passing through the vertex $s$.

Definition 37.4.1. An edge $uv \in E(G)$ is a bridge if the following conditions are met: (i) $u$ and $v$ are children of the same vertex in $T$, (ii) if the depth of $u$ in $T$ is even then $uv$ is free (i.e., $uv \notin M$, and (iii) if the depth of $u$ in $T$ 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 $T$ and $G$.

The following is an easy technical lemma.

Lemma 37.4.2. Let $v$ be a vertex of $G$, $M$ a matching in $G$, and let $\pi$ be the shortest alternating path between $s$ and $v$ in $G$. Furthermore, assume that for any vertex $w$ of $\pi$ the shortest alternating path between $w$ and $s$ is the path along $\pi$.

Then, the depth $d_T(v)$ of $v$ in $T$ 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 $T$.

For $|\pi| = k$, consider the vertex just before $v$ on $\pi$, and let us denote it by $u$. By induction, the depth of $u$ in $T$ 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 37.4.3. 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 $T$.

Proof: Let $\pi$ be an augmenting path in $G$. The path $\pi$ corresponds to a an odd length alternating cycle in $H$. Let $\sigma$ be the shortest odd length alternating cycle in $G$ (note that both edges in $\sigma$ that are adjacent to $s$ are unmatched).

For a vertex $x$ of $\sigma$, 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 $x$ and $s$ along $\sigma$. Clearly, $d(x) \leq d'(x)$, but we claim that in fact $d(x) = d'(x)$, for all $x \in \sigma$. Indeed, assume for the sake of contradiction that $d(x) < d'(x)$, and let $\pi_1, \pi_2$ be the two paths from $x$ to $s$ formed by $\sigma$. Let $\eta$ 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 $\sigma$ involving $s$, which is a contradiction.

But then, take the two vertices of $\sigma$ furthest away from $s$. Clearly, both of them have the same depth in $T$, since $d(u) = d'(u) = d'(v) = d(v)$. By Lemma 37.4.2, we now have that $d_T(u) = d(u) = d(v) = d_T(v)$. Establishing the first part of the claim. See Figure 37.1.

As for the second claim, observe that it easily follows as $\sigma$ is created from an alternating path.

Thus, we can do the following: Compute the alternating BFS $T$ 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 37.4.3 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 $T$, and glue them together with the edge $uv$ to form an odd cycle $\mu$ in $H$; namely, $\mu = \tau_{su} \parallel uv \parallel \tau_{sv}$. If $\mu$ 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.
But $\mu$, in fact, might have common edges. In particular, let $\pi_{su}$ and $\pi_{sv}$ be the two paths from $s$ to $u$ and $v$, respectively. Let $w$ be the lowest vertex in $T$ that is common to both $\pi_{su}$ and $\pi_{sv}$.

**Definition 37.4.4.** 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 37.4.5.** Consider a bridge edge $uv \in G$, and let $w$ be the least common ancestor (LCA) of $u$ and $v$ in $T$. Consider the path $\pi_{sw}$ together with the cycle $C = \pi_{wu} \ || \ uv \ || \ \pi_{wv}$. Then $\pi_{sw}$ and $C$ together form a flower.

**Proof:** Since only the even depth nodes in $T$ have more than one child, $w$ must be of even depth, and as such $\pi_{sw}$ 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 $T$. In particular, $|C| = |\pi_{wu}| + |\pi_{wv}| + 1 = 2\alpha + 1$, which is an odd number. 

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 $w \ldots u \ldots v \ldots w$ 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 37.4.6.** 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 37.4.7.** 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 $\pi$ which is an even length alternating path starting from 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 37.3. 

**Theorem 37.4.8.** 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 $G/C$. 

220
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 $\pi$ 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 $\pi$ 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 $\pi$ 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 $\pi$ starts and ends in two different free vertices and $C$ has only one free vertex, it follows that $\pi$ has an endpoint outside $C$. Let $v$ be this endpoint of $\pi$ and let $u$ be the first vertex of $\pi$ that belongs to $C$. Let $\sigma$ 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 $\pi$ 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 $\pi$ is unmatched. Thus, rotate the matching $M$ around $C$ such that $u$ becomes free. Clearly, then $\sigma$ is now an augmenting path in $G$ (for the rotated matching) and also an augmenting path in $G/C$.

Corollary 37.4.9. 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 $G/C$.

37.4.2. 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 $\rho$ 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 $\pi$ 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.

37.4.2.1. 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 37.4.10. Given a graph $G$ with $n$ vertices and $m$ edges, computing a maximum size matching in $G$ can be done in $O(n^2 m)$ time.
37.5. 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

The description above also follows loosely the same book.

Chapter 38

Lower Bounds

38.1. 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 $\text{compare}(i, j)$ 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 38.1.1. 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., $\text{compare}$) a deterministic sorting algorithm have to perform?

38.1.1. 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 38.1. The algorithm starts at the root.

But what is a sorting algorithm? The output of a sorting algorithm is the input elements is a certain order. That is, a sorting algorithm for $n$ elements outputs a permutation $\pi$ of $[n] = \{1, \ldots, n\}$. Formally, if the input is $x_1, \ldots, x_n$ the output is a permutation $\pi$ of $[n]$, such that $x_{\pi(1)} < x_{\pi(2)} < \ldots < x_{\pi(n)}$.

Initially all $n!$ permutations are possible, but as the algorithm performs comparisons, and as the algorithm descend 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 38.1 have 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 algorithm decides that $x_1 < x_2$, then the algorithm descends into the
Figure 38.1: A decision tree for sorting three elements.

node $u$, and the possible ordering 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 38.1.2. 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 38.1.3. Given a permutation $\pi$ of $[n]$, an input that is sorted in the ordering specified by $\pi$ is the following: $x_i = \pi^{-1}(i)$, for $i = 1, \ldots, n$.

Proof: The input we construct would be made out of the numbers of $[n]$. 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 38.1.4. A convenient way to do the above transformation is the following. Write the permutation as a function $\pi$ by writing it as matrix with two rows, the top row having 1, $\ldots, n$, 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 38.1.5. 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 38.1.6. 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 $u \in V(\mathcal{T})$.

Imagine, that the algorithm performed $k$ comparisons, and it is currently at a node $v_j$ 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 $v_i$, 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 $v_1, \ldots, v_k$ 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 $1 = |\Phi(v_k)| \geq \frac{|\Phi(v_{k-1})|}{2} \geq \ldots \geq \frac{|\Phi(v_1)|}{2^{k-1}}$. 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.

\[ \blacksquare \]

### 38.1.2. An easier direct argument

**Proof:** (Proof of Lemma 38.1.6.) Consider the set $\Pi$ of all permutations of $[n]$ (each can be interpreted as a sequence of the $n$ numbers $1, \ldots, n$). We treat an element $(x_1, \ldots, x_n) \in \Pi$ 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 $n!$ leafs, 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.

\[ \blacksquare \]

The reader might wonder why we bothered to show the original proof of Lemma 38.1.6. 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 gave against an adversary – this is a rather powerful and useful idea.

### 38.2. Uniqueness

Problem 38.2.1 (Uniqueness). Given an input of $n$ real numbers $x_1, \ldots, x_n$. 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 38.2.2.** 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 38.2.3.** 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 $\sigma$ and $\sigma'$ be any two different permutations in $\Phi(v)$, and let $X = x_1, \ldots, x_n$ be an input realizing $\sigma$, and let $Y = y_1, \ldots, y_n$ be an input realizing $\sigma'$. Let $Z(t) = z_1(t), \ldots, z_n(t)$ be an input where $z_i(t) = tx_i + (1-t)y_i$. Clearly, $Z(0) = x_1, \ldots, x_n$ and $Z(1) = y_1, \ldots, y_n$.

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 $ith$ element of the input to the $jth$ 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 $\mathcal{T}$.

For $t = \alpha$, 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 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 $v$ cannot be a leaf. 

**Proof: (of Theorem 38.2.2)** We apply the same argument as in Lemma 38.1.6. 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 $\Omega(n \log n)$. 

---

### 38.3. Other lower bounds

#### 38.3.1. 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 38.3.1 (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 38.3.2 (Jeff Erickson and Raimund Seidel [ES95]).** Solving the degenerate points problem requires $\Omega(n^d)$ time in a “reasonable” model of computation.

#### 38.3.2. 3Sum-Hard

**Problem 38.3.3 (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 $a + b = c$. 

---

225
We leave the following as an exercise to the interested reader.

**Lemma 38.3.4.** 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 39**

**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.

### 39.1. How many times can the minimum change?

Let $\Pi = \pi_1 \ldots \pi_n$ be a random permutation of $\{1, \ldots, n\}$. Let $\mathcal{E}_i$ be the event that $\pi_i$ is the minimum number seen so far as we read $\Pi$; that is, $\mathcal{E}_i$ is the event that $\pi_i = \min_{k=1}^{i-1} \pi_k$. 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 $E[X_i] = 1/i$. We are interested in how many times the minimum might change$^\dagger$; that is $Z = \sum_i X_i$, and how concentrated is the distribution of $Z$. The following is maybe surprising.

**Lemma 39.1.1.** The events $\mathcal{E}_1, \ldots, \mathcal{E}_n$ are independent (as such, variables $X_1, \ldots, X_n$ 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 $\pi_n$. We then, again, pick a random permutation of the remaining numbers and set the first number as the penultimate number (i.e., $\pi_{n-1}$) in the output permutation. We repeat this process till we generate the whole permutation.

Now, consider $1 \leq i_1 < i_2 < \ldots < i_k \leq n$, and observe that $\Pr[\mathcal{E}_{i_1} \cap \mathcal{E}_{i_2} \cap \ldots \cap \mathcal{E}_{i_k}] = \Pr[\mathcal{E}_{i_1}]$, since by our thought experiment, $\mathcal{E}_{i_1}$ is determined after all the other variables $\mathcal{E}_{i_2}, \ldots, \mathcal{E}_{i_k}$. In particular, the variable $\mathcal{E}_{i_1}$ is inherently not effected by these events happening or not. As such, we have

$$
\Pr[\mathcal{E}_{i_1} \cap \mathcal{E}_{i_2} \cap \ldots \cap \mathcal{E}_{i_k}] = \Pr[\mathcal{E}_{i_1}] \Pr[\mathcal{E}_{i_2} \cap \ldots \cap \mathcal{E}_{i_k}] = \Pr[\mathcal{E}_{i_1}] \prod_{j=1}^{k} \Pr[\mathcal{E}_{i_j}] = \prod_{j=1}^{k} \frac{1}{i_j}.
$$

$\dagger$The answer, my friend, is blowing in the permutation.

226
by induction.

**Theorem 39.1.2.** Let \( \Pi = \pi_1 \ldots \pi_n \) be a random permutation of \( 1, \ldots, n \), and let \( Z \) be the number of times, that \( \pi_i \) is the smallest number among \( \pi_1, \ldots, \pi_i \), for \( i = 1, \ldots, n \). Then, we have that for \( t \geq 2e \) that \( \Pr[Z > t \ln n] \leq 1/n^{4/5} \), and for \( t \in [1, 2e] \), we have that \( \Pr[Z > t \ln n] \leq 1/n^{(t-1)^2/4} \).

**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

\[
\mu = E[Z] = \sum_i E[X_i] = n \sum_{i=1}^{n-1} \frac{1}{i} \geq \int_{x=1}^{n+1} \frac{1}{x} \, dx = \ln(n + 1) \geq \ln n.
\]

Next, we set \( \delta = t - 1 \), and use Chernoff inequality.

---

**39.2. 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 \( \langle p_1, \ldots, p_n \rangle \) 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 \( P_i = \{p_1, \ldots, p_i\} \) of pivots has already been handled by the algorithm. That is, the algorithm have these pivots in sorted orders \( p_1^i < p_2^i < \ldots < p_{i}^i \). In addition, the numbers that were not handled yet \( P \setminus P_i \), are partitions into sets \( Q_0, \ldots, Q_i \), where all the numbers in \( P \setminus P_i \) between \( p_{i}^i \) and \( p_{i+1}^i \) 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 39.2.1.** 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 \( P_i = \{p_1, \ldots, p_i\} \) 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 \( E[\sum_i X_i] \leq \sum_{i=1}^{n} 2/i = O(\log n) \), as desired. This also implies that QuickSort takes \( O(n \log n) \) time in expectation.

**Exercise 39.2.2.** Prove using backward analysis that QuickSort takes \( O(n \log n) \) with high probability.

It is not true that the indicator variables \( X_1, X_2, \ldots \) are independent (this is quite subtle and not easy to see, as such extending directly the proof of Theorem 39.1.2 for this case does not work.)
39.3. Closest pair: Backward analysis in action

We are interested in solving the following problem:

Problem 39.3.1. 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 \( CP(P) = \min_{p \neq q, p, q \in P} \|p - q\| \).

39.3.1. Definitions

Definition 39.3.2. For a real positive number \( \Delta \) and a point \( p = (p_1, \ldots, p_d) \in \mathbb{R}^d \), define \( G_{\Delta}(p) \) to be the grid point \((\lfloor p_1 / \Delta \rfloor \Delta, \ldots, \lfloor p_d / \Delta \rfloor \Delta)\).

We call \( \Delta \) the width or sidelength of the grid \( G_{\Delta} \). Observe that \( G_{\Delta} \) partitions \( \mathbb{R}^d \) into cubes, which are grid cells. The grid cell of \( p \) is uniquely identified by the integer point \( \text{id}(p) = (\lfloor p_1 / \Delta \rfloor, \ldots, \lfloor p_d / \Delta \rfloor) \).

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.

39.3.2. Back to the problem

The following is an easy standard packing argument that underlines, under various disguises, many algorithms in computational geometry.

Lemma 39.3.3. Let \( P \) be a set of points contained inside a square \( \square \), such that the sidelength of \( \square \) is \( \alpha = CP(P) \). Then \( |P| \leq 4 \).

Proof: Partition \( \square \) into four equal squares \( \square_1, \ldots, \square_4 \), 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 \( \alpha \) 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 39.3.4. Given a set \( P \) of \( n \) points in the plane and a distance \( \alpha \), one can verify in linear time whether \( CP(P) < \alpha \), \( CP(P) = \alpha \), or \( CP(P) > \alpha \).

Proof: Indeed, store the points of \( P \) in the grid \( G_{\alpha} \). 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 \( CP(P) < \alpha \), by Lemma 39.3.3.
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 \( G_\alpha \) that intersects the disk \( D = \text{disk}(p, \alpha) \) centered at \( p \) with radius \( \alpha \); see the figure on the right. If there is a point closer to \( p \) than \( \alpha \) 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 39.3.3 (otherwise, we would already have stopped since the \( CP(\cdot) \) of the inserted points is smaller than \( \alpha \)). 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 \( d(p, S) < \alpha \), we stop; otherwise, we continue to the next point.

Overall, this takes at most linear time.

As for correctness, observe that the algorithm returns ‘\( CP(P) < \alpha \)’ only after finding a pair of points of \( P \) with distance smaller than \( \alpha \). So, assume that \( p \) and \( q \) are the pair of points of \( P \) realizing the closest pair and that \( \|p - q\| = CP(P) < \alpha \).

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 ‘\( CP(P) < \alpha \)’. Similar argumentations work for the case that \( CP(P) = \alpha \). Thus if the algorithm returns ‘\( CP(P) > \alpha \)’, it must be that \( CP(P) \) is not smaller than \( \alpha \) or equal to it. Namely, it must be larger. Thus, the algorithm output is correct.

Remark 39.3.5. Assume that \( CP(P \setminus \{p\}) \geq \alpha \), but \( CP(P) < \alpha \). Furthermore, assume that we use Lemma 39.3.4 on \( P \), where \( p \in P \) is the last point to be inserted. When \( p \) is being inserted, not only do we discover that \( CP(P) < \alpha \), 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 takes only constant time.

39.3.3. Slow algorithm

Lemma 39.3.4 provides a natural way of computing \( CP(P) \). Indeed, permute the points of \( P \) in an arbitrary fashion, and let \( P = \langle p_1, \ldots, p_n \rangle \). Next, let \( \alpha_{i-1} = CP(\{p_1, \ldots, p_{i-1}\}) \). We can check if \( \alpha_i < \alpha_{i-1} \) by using the algorithm of Lemma 39.3.4 on \( P_i \) and \( \alpha_{i-1} \). In fact, if \( \alpha_i < \alpha_{i-1} \), the algorithm of Lemma 39.3.4 would return ‘\( CP(P) = \alpha_{i-1} \)’ and the two points of \( P_i \) realizing \( \alpha_i \).

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 39.3.4 to store \( P_i = \langle p_1, \ldots, p_i \rangle \). 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 insert all the points of \( P_i = \langle p_1, \ldots, p_i \rangle \) into the new grid \( G_{\alpha_i}(P_i) \). This takes \( O(i) \) time.

In the end of this process, we output the number \( \alpha_n \), together with the two points of \( P \) that realize the closest pair.

Observation 39.3.6. If the closest pair distance, in the sequence \( \alpha_1, \ldots, \alpha_n \), 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.

39.3.4. Linear time algorithm

Surprisingly\(^a\), 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 \( \langle p_1, \ldots, p_n \rangle \) be this permutation. Let \( \alpha_2 = \|p_1 - p_2\| \), and start inserting the points into the data-structure of Lemma 39.3.4. We will keep the invariant that \( \alpha_i \) would be the closest pair distance in the set \( P_i \), for \( i = 2, \ldots, n \).

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 \( \alpha_i \) (see Remark 39.3.5), rebuild the grid, and insert the \( i \) points of \( P_i \) from scratch into the grid \( G_{\alpha_i} \). This rebuilding of \( G_{\alpha_i}(P_i) \) takes \( O(i) \) time.

\(^a\)Surprise 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".

229
Finally, the algorithm returns the number $\alpha_n$ and the two points of $P_n$ realizing it, as the closest pair in $P$.

**Lemma 39.3.7.** Let $t$ be the number of different values in the sequence $\alpha_2, \alpha_3, \ldots, \alpha_n$. Then $E[t] = O(\log n)$. 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 $E[X_i] = Pr[X_i = 1]$ (as $X_i$ is an indicator variable) and $t = \sum_{i=3}^{n} X_i$.

To bound $Pr[X_i = 1] = Pr[\alpha_i < \alpha_{i-1}]$, we (conceptually) fix the points of $P_i$ and randomly permute them. A point $q \in P_i$ is critical if $CP(P_i \setminus \{q\}) > CP(P_i)$. If there are no critical points, then $\alpha_{i-1} = \alpha_i$ and then $Pr[X_i = 1] = 0$ (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 $Pr[X_i = 1] = 1/i$, 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 $CP(P_i)$. The quantity $\alpha_i$ is smaller than $\alpha_{i-1}$ 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 $CP(P_i \setminus \{s\}) = ||p - q||$, and hence the point $s$ is not critical.

Thus, $Pr[X_i = 1] = Pr[\alpha_i < \alpha_{i-1}] \leq 2/i$, and by linearity of expectations, we have that $E[t] = E[\sum_{i=3}^{n} X_i] = \sum_{i=3}^{n} E[X_i] \leq \sum_{i=3}^{n} 2/i = O(\log n)$.

Lemma 39.3.7 implies that, in expectation, the algorithm rebuilds the grid $O(\log n)$ times. By Observation 39.3.6, 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).

**Theorem 39.3.8.** 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

$$E[R] = E\left[1 + \sum_{i=3}^{n} (1 + X_i \cdot i)\right] \leq n + \sum_{i=3}^{n} E[X_i] \cdot i \leq n + \sum_{i=3}^{n} i \cdot Pr[X_i = 1]$$

$$\leq n + \sum_{i=3}^{n} i \cdot \frac{2}{i} \leq 3n,$$

by linearity of expectation and since $E[X_i] = Pr[X_i = 1]$ and since $Pr[X_i = 1] \leq 2/i$ (as shown in the proof of Lemma 39.3.7). Thus, the expected running time of the algorithm is $O(E[R]) = O(n)$.

**Theorem 39.3.8** 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 39.3.8 is considerably stronger, using hashing, randomization, and the floor function.
39.4. Computing a good ordering of the vertices of a graph

We are given a graph \( G = (V, E) \) be an edge-weighted graph with \( n \) vertices and \( m \) edges. The task is to compute an ordering \( \pi = (\pi_1, \ldots, \pi_n) \) of the vertices, and for every vertex \( v \in V \), the list of vertices \( L_v \), such that \( \pi_i \in L_v \), if \( \pi_i \) is the closest vertex to \( v \) in the \( i \)th prefix \( (\pi_1, \ldots, \pi_i) \).

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 \( \mathcal{L} = \sum_{v \in V} |L_v| \).

39.4.1. The algorithm

Take a random permutation \( \pi_1, \ldots, \pi_n \) 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 \( \pi_i \). 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 \( \pi_i \) 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.

39.4.2. Analysis

Lemma 39.4.1. The above algorithm computes a permutation \( \pi \), 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 \( v \in V = \{v_1, \ldots, v_n\} \). Consider the set of \( n \) numbers \( \{d_G(v, v_1), \ldots, d_G(v, v_n)\} \). Clearly, \( d_G(v, \pi_1), \ldots, d_G(v, \pi_n) \) is a random permutation of this set, and by Lemma 39.1.1 the random permutation \( \pi \) 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.

39.5. Computing nets

39.5.1. Basic definitions

39.5.1.1. Metric spaces

Definition 39.5.1. A metric space is a pair \( (X, d) \) where \( X \) is a set and \( d : X \times X \to [0, \infty) \) is a metric satisfying the following axioms: (i) \( d(x, y) = 0 \) if and only if \( x = y \), (ii) \( d(x, y) = d(y, x) \), and (iii) \( d(x, y) + d(y, z) \geq d(x, z) \) (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 \( d_M \). Namely, given two points \( p, q \in X \), we assume that \( d(p, q) \) can be computed in constant time.

Another standard example for a finite metric space is a graph \( G \) with non-negative weights \( w(\cdot) \) 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.
39.5.2. Nets

Definition 39.5.2. For a point set $P$ in a metric space with a metric $d$, and a parameter $r > 0$, an $r$-net of $P$ is a subset $C \subseteq P$, such that (i) for every $p, q \in C$, $p \neq q$, we have that $d(p, q) \geq r$, and (ii) for all $p \in P$, we have that $\min_{q \in C} d(p, q) < r$.

Intuitively, an $r$-net represents $P$ in resolution $r$.

39.5.2. 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, $p$, add $p$ to $C$, and mark it and all other points in distance $< \Delta = \Theta$ from $p$. 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 — it was also described by the authors in [ERH12].

Lemma 39.5.3. 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.

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, 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 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 $N_{\mathcal{G}}(p)$ 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 $c \in N_{\mathcal{G}}(p)$. 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 39.5.4. 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 39.5.4 implies that for a weighted point set one can compute the following quantity in linear time.

Algorithm 39.5.5 (net). Given a weighted point set $P \subseteq \mathbb{R}^d$, let $N(r, P)$ 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 $N(r, P)$ to designate the algorithm computing this net, which has linear running time.

Specifically, the algorithm of Har-Peled [Har04] is considerably more complicated than Lemma 39.5.3, and does not work in this settings, as the number of clusters it can handle is limited to $O(n^{1/6})$. Lemma 39.5.3 has no such restriction.
39.5.3. Computing an \( r \)-net in a sparse graph

Given a graph \( G = (V, E) \) 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 39.5.2.

39.5.3.1. 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 \( \pi_i \) be the \( i \)th vertex in a random permutation \( \pi \) 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 \( \pi_i \) to the resulting net \( N \).
- (B) Set \( \delta(\pi_i) \) to zero.
- (C) Perform Dijkstra’s algorithm starting from \( \pi_i \), 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 \( \pi_i \), and relax the edges adjacent to \( u \) in the graph.

39.5.3.2. Analysis

While the analysis here does not directly uses backward analysis, it is inspired to a large extent by such an analysis as in Section 39.4.

Lemma 39.5.6. The set \( N \) 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 \( N \) is at most \( \ell \). Indeed, for the sake of contradiction, let \( j \) be the (end of) the first iteration where this claim is false. It must be that \( \pi_j \in N \), and it is the nearest vertex in \( N \) to \( v \). But then, consider the shortest path between \( \pi_j \) 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 \( N \) have distance \( \geq r \). Indeed, when the algorithm handles vertex \( v \in N \), its distance from all the vertices currently in \( N \) is \( \geq r \), implying the claim. \( \square \)

Lemma 39.5.7. 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 \( \Pi_i = \{\pi_1, \ldots, \pi_i\} \).
- (B) If \( \pi_{i+1} = x \) then \( \delta(v) \) would change in the \((i + 1)\)th iteration.

Let \( s_i = |S_i| \). Observe that \( S_1 \supseteq S_2 \supseteq \cdots \supseteq S_n \), and \( |S_n| = 0 \).

In particular, let \( E_{i+1} \) 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 \( \pi_{i+1} \). However, \( \pi_{i+1} \) has a uniform distribution over the vertices of \( S_i \), and in particular, if \( E_{i+1} \) 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 \( E_{i+1} \) 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 \( \lfloor \log_2 n \rfloor \) iterations were lucky with high probability, implying the claim. Here \( c \) is a sufficiently large constant. \( \square \)
Interestingly, in the above proof, all we used was the monotonicity of the sets $S_1, \ldots, S_n$, 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 39.5.8.** 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 39.5.7, the two $\delta$ 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. \hfill \blacksquare

### 39.6. 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 39.1 is by now folklore.

The closet-pair result is Section 39.3 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 39.4 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 39.5.2, is from Har-Peled and Raichel [HR13].

Computing a net in a sparse graph, Section 39.5.3, is from [EHS14]. While backwards analysis fails to hold in this case, it provide a good intuition for the analysis, which is slightly more complicated and indirect.
This chapter includes problems that are perquisite. Their main purpose is to check whether you are ready 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.

40.1. Graph Problems

40.1.1. 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 → 4 → 5 → 2 → 3 → 1.

40.1.2. Graphs! Graphs!

(20 pts.)
A coloring of a graph $G$ by $\alpha$ colors is an assignment to each vertex of $G$ a color which is an integer between 1 and $\alpha$, such that no two vertices that are connected by an edge have the same color.
(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.

(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.)

(C) (5 pts.) A directed graph $G = (V, E)$ is a neat graph if there exist an ordering of the vertices of the graph $V(G) = \langle v_1, v_2, \ldots, v_n \rangle$ 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.

(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.

40.1.3. 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 than 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.

40.1.4. Connectivity and walking.

(10 pts.)

(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 $v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$, in which $e_l$ has endpoints $v_{l-1}$ and $v_l$.

(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.

40.1.5. 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.
40.1.6. Coloring

(10 pts.)

(A) (5 pts.) Let $T_1, T_2$ and $T_3$ be three trees defined over the set of vertices $\{v_1, \ldots, v_n\}$. 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 $T_3$).

(B) (5 pts.) Describe an efficient algorithm for computing this coloring. What is the running time of your algorithm?

40.1.7. 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.

40.2. Recurrences

40.2.1. 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.

(A) (2 pts.) $A(n) = A(\sqrt{n}/3 + \lfloor \log n \rfloor) + n$

(B) (2 pts.) $B(n) = \min_{0 < k < n} (3 + B(k) + B(n - k))$.

(C) (2 pts.) $C(n) = 3C(\lfloor n/2 \rfloor - 5) + n/ \log n$

(D) (2 pts.) $D(n) = \frac{n}{n-3}D(n-1) + 1$

(E) (2 pts.) $E(n) = E(\lfloor 3n/4 \rfloor) + \sqrt{n}$

(F) (2 pts.) $F(n) = F(\lfloor \log n \rfloor) + \log n$ (HARD)

(G) (2 pts.) $G(n) = n + \lfloor \sqrt{n} \rfloor \cdot G(\lfloor \sqrt{n} \rfloor)$

(H) (2 pts.) $H(n) = \log(H(n-1)) + 1$

(I) (2 pts.) $I(n) = 5I(\lfloor \sqrt{n} \rfloor) + 1$

(J) (2 pts.) $J(n) = 3J(n/4) + 1$
40.2.2. 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.

(A) (1 pts.) $A(n) = A(n/3 + 5 + \lfloor \log n \rfloor) + n \log \log n$

(B) (1 pts.) $B(n) = \min_{0 < k < n} (3 + B(k) + B(n - k))$.

(C) (1 pts.) $C(n) = 3C([n/2] - 5) + n/ \log n$

(D) (1 pts.) $D(n) = 5D(n/5) + 1$

(E) (1 pts.) $E(n) = E([3n/4]) + 1/ \sqrt{n}$

(F) (1 pts.) $F(n) = F(\lfloor \log^2 n \rfloor) + \log n$ (HARD)

(G) (1 pts.) $G(n) = n + 7 \sqrt{n} \cdot G(\lfloor \sqrt{n} \rfloor)$

(H) (1 pts.) $H(n) = \log^2 (H(n - 1)) + 1$

(I) (1 pts.) $I(n) = I(\lfloor n^{1/4} \rfloor) + 1$

(J) (1 pts.) $J(n) = J(n - \lfloor n/ \log n \rfloor) + 1$

40.2.3. 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.

(A) $A(n) = A(n/2) + n$

(B) $B(n) = 2B(n/2) + n$

(C) $C(n) = n + \frac{1}{2}(C(n - 1) + C(3n/4))$

(D) $D(n) = \max_{n/3 < k < 2n/3} (D(k) + D(n - k) + n)$ (HARD)

(E) $E(n) = 2E(n/2) + n/ \log n$ (HARD)

(F) $F(n) = \frac{F(n-1)}{F(n-2)}$, where $F(1) = 1$ and $F(2) = 2$. (HARD)

(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)

(H) $H(n) = n + \sqrt{n} \cdot H(\sqrt{n})$ (HARD)

(I) $I(n) = (n - 1)(I(n - 1) + I(n - 2))$, where $F(0) = F(1) = 1$ (HARD)

(J) $J(n) = 8J(n - 1) - 15J(n - 2) + 1$
40.2.4. 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 \( f(n) \).

(A) (2 pts.) \[ \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=j}^{i} \frac{1}{i} \]

(HARD)

(B) (2 pts.) \[ \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=j}^{i} \frac{1}{j} \]

(C) (2 pts.) \[ \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=j}^{i} \frac{1}{k} \]

(D) (2 pts.) \[ \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} \frac{1}{k} \]

(E) (2 pts.) \[ \sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=1}^{j} \frac{1}{j \cdot k} \]

40.2.5. 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.

(A) Suppose \( \alpha_0 = p, \alpha_1 = q, \) and \( \alpha_n = (\alpha_{n-2} \land \alpha_{n-1}) \) for all \( n \geq 2 \). Simplify \( \alpha_n \) as much as possible. [Hint: What is \( \alpha_5 \)?]

(B) Suppose \( \beta_0 = p, \beta_1 = q, \) and \( \beta_n = (\beta_{n-2} \equiv \beta_{n-1}) \) for all \( n \geq 2 \). Simplify \( \beta_n \) as much as possible. [Hint: What is \( \beta_5 \)?]

(C) Suppose \( \gamma_0 = p, \gamma_1 = q, \) and \( \gamma_n = (\gamma_{n-2} \Rightarrow \gamma_{n-1}) \) for all \( n \geq 2 \). Simplify \( \gamma_n \) as much as possible. [Hint: What is \( \gamma_5 \)?]

(D) Suppose \( \delta_0 = p, \delta_1 = q, \) and \( \delta_n = (\delta_{n-2} \ast \delta_{n-1}) \) for all \( n \geq 2 \), where \( \ast \) is some boolean function with two arguments. Find a boolean function \( \ast \) such that \( \delta_n = \delta_m \) if and only if \( n - m \) is a multiple of 4. [Hint: There is only one such function.]

40.3. Counting

40.3.1. Counting dominos

(A) A domino is a \( 2 \times 1 \) or \( 1 \times 2 \) 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.

(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 \times 10 rectangle filled with ten dominos, and a 2 \times 2 \times 10 box filled with ten slabs.
40.4. O notation and friends

40.4.1. 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.

\[
\begin{align*}
&n^{4.5} - (n - 1)^{4.5} & n & n^{2.1} & \lg^* (n/8) & 1 + \lg \lg \lg n \\
&\cos n + 2 & \lg (\lg^* n) & (\lg n)! & (\lg^* n)^{\lg n} & n^5 \\
&\lg^* 2^{2^n} & 2^{\lg n} & \sqrt{n^e} & \sum_{i=1}^n i & \sum_{i=1}^n i^2 \\
&n^{7/(2n)} & n^{3/(2\lg n)} & 12 + [\lg \lg (n)] & (\lg (2 + n))^{\lg n} & (1 + \frac{1}{154})^{15n} \\
&n^{1/\lg \lg n} & n^{1/2^{\lg n}} & \lg^{(201)} n & n^{1/125} & n (\lg n)^4 \\
\end{align*}
\]

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 \( n^2, n, \left(\frac{n}{2}\right), n^3 \) could be sorted either as \( n \ll n^2 \equiv \left(\frac{n}{2}\right) \ll n^3 \) or as \( n \ll \left(\frac{n}{2}\right) \equiv n^2 \ll n^3 \). [Hint: When considering two functions \( f(\cdot) \) and \( g(\cdot) \) it is sometime useful to consider the functions \( \ln f(\cdot) \) and \( \ln g(\cdot) \).]

40.4.2. 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.

\[
\begin{align*}
&n^{5.5} - (n - 1)^{5.5} & n & n^{2.2} & \lg^* (n/7) & 1 + \lg \lg n \\
&\cos n + 2 & \lg (\lg^* n) & (\lg n)! & (\lg^* n)^{\lg n} & n^4 \\
&\lg^* 2^{2^n} & 2^{\lg n} & e^{\sqrt{n}} & \sum_{i=1}^n \frac{1}{i} & \sum_{i=1}^n \frac{1}{i^2} \\
&n^{3/(2n)} & n^{3/(2\lg n)} & [\lg \lg (n!)] & (\lg (7 + n))^{\lg n} & (1 + \frac{1}{154})^{154n} \\
&n^{1/\lg \lg n} & n^{1/2^{\lg n}} & \lg^{(200)} n & n^{1/1234} & n (\lg n)^3 \\
\end{align*}
\]

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 \( n^2, n, \left(\frac{n}{2}\right), n^3 \) could be sorted either as \( n \ll n^2 \equiv \left(\frac{n}{2}\right) \ll n^3 \) or as \( n \ll \left(\frac{n}{2}\right) \equiv n^2 \ll n^3 \).

40.4.3. O notation revisited.

(10 pts.)

(A) Let \( f_i(n) \) be a sequence of functions, such that for every \( i \), \( f_i(n) = o(\sqrt{n}) \) (namely, \( \lim_{n \to \infty} \frac{f_i(n)}{\sqrt{n}} = 0 \)). Let \( g(n) = \sum_{i=1}^n f_i(n) \). Prove or disprove: \( g(n) = o(n^{3/2}) \).

(B) If \( f_1(n) = O(g_1(n)) \) and \( f_2(n) = O(g_2(n)) \). Prove or disprove:

- \( f_1(n) + f_2(n) = O(g_1(n) + g_2(n)) \)
- \( f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n)) \)
- \( f_1(n) f_2(n) = O(g_1(n) g_2(n)) \)
40.4.4. Some proofs required.

(A) Prove that $2^{\lceil \log n \rceil} + \lfloor \log n \rfloor = \Theta(n^2)$.

(B) Prove or disprove: $2^{\lfloor \log n \rfloor} = \Theta(2^{\lceil \log n \rceil})$.

(C) Prove or disprove: $2^{2^{\lfloor \log \log n \rfloor}} = \Theta(2^{2^{\lceil \log \log n \rceil}})$.

(D) Prove or disprove: If $f(n) = O(g(n))$, then $\log(f(n)) = O(\log(g(n)))$.

(E) Prove or disprove: If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.

(F) Prove that $\log_k n = o(n^{1/k})$ for any positive integer $k$.

40.5. Probability

40.5.1. 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.

(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 $X$?

(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.)

(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?

(D) (4 pts.) What is the probability that there are exactly $k$ matches? (1 ≤ $k$ < $n$)

[Hint: If you have to appeal to “intuition” or “common sense”, your answers are probably wrong!]

40.5.2. Idiotic Sort

(20 pts.)

There is an array $A$ with $n$ unsorted distinct numbers in it. IdioticSort($A$) sorts the array using an iterative algorithm. In each iteration, it picks randomly (and uniformly) two indices $i, j$ in the ranges $\{1, \ldots, n\}$. 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.

(A) (5 pts.) Prove that after (at most) $n!$ swaps performed by the algorithm, the array $A$ is sorted.

(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.)

(C) (5 pts.) Prove that if $A$ is not sorted, than the probability for a swap in the next iteration is at least $\geq 2/n^2$.

(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.]

(E) (5 pts.) Prove that the expected number of iterations performed by the algorithm is $O(n^5)$. [Hint: Use linearity of expectation.]
40.5.3. 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 $v_0, v_1, \ldots, v_m$ is of length $m$.

(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 $P_u(n-1, v) > 0$.

(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 $\lim_{m \to \infty} P_u(m, v) = 1$.

40.5.4. 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: $T_1, \ldots, T_n$. 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.)

40.5.5. 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.

40.5.6. 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, \ldots, 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.

(A) On average, how many cards does Penn give Teller?

(B) On average, what is the smallest-numbered card that Penn gives Teller? (HARD)

(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!

40.5.7. 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!
40.6. Basic data-structures and algorithms

40.6.1. 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.

40.6.2. 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.)

(A) (2 pts.) Find the smallest element stored in $T$ in $O(h)$ time.

(B) (2 pts.) Given a query $k$, find the $k$-th smallest element stored in $T$ in $O(h)$ time.

(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.

(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.

40.6.3. 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 WEIRD EUCLID would be preferable to FAST EUCLID?

\begin{verbatim}
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)
\end{verbatim}
**40.6.4. 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?

(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\(^\text{a} \) that rearranges the items into four clusters in \( O(n) \) time using only \( O(1) \) extra space.

(B) Describe an algorithm for the case where there are \( k \) possible values (i.e., 1, 2, \ldots, \( k \)) that rearranges the items using only \( O(\log k) \) extra space. How fast is your algorithm? (A faster algorithm would get more credit)

(C) Describe a faster algorithm (if possible) for the case when \( O(k) \) extra space is allowed. How fast is your algorithm?

(D) (HARD) Provide a fast algorithm that uses only \( O(1) \) additional space for the case where there are \( k \) possible values.

\( ^{\text{a}} \) Since you’ve read the Homework Instructions, you know what the phrase ‘describe an algorithm’ means. Right?
40.6.5. 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 code\(^\text{a}\), so that the last node has a pointer back to some other node in the list instead.

![Diagram of a standard linked list and 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).

40.7. General proof thingies

40.7.1. 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 \(i = 1, \ldots, n\).

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.

(A) (5 pts.) Prove that the game terminates if you start from a configuration where all the cobs are in a single sack.

(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.

(C) (5 pts.) (hard) Prove that the game terminates if you start from an arbitrary configuration where the cobs might be in several sacks.

(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.

40.7.2. 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 \(m\).

(A) \(F_n\) is even if and only if \(n\) is divisible by 3.

(B) \(\sum_{i=0}^{n} F_i = F_{n+2} - 1\)

(C) \(F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}\) (Really HARD)

(D) If \(n\) is an integer multiple of \(m\), then \(F_n\) is an integer multiple of \(F_m\).

\(^{a}\)After all, your code is always completely 100% bug-free. Isn’t that right, Mr. Gates?
40.7.3. Some binomial identities.

(A) Prove the following identity by induction:

\[
\binom{2n}{n} = \sum_{k=0}^{n} \binom{n}{k} \binom{n}{n-k}.
\]

(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.

40.8. Miscellaneous

40.8.1. 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.

\[\begin{array}{c}
\text{t=0} \\
\text{t=1} \\
\text{t=2}
\end{array}\]

Every second, the ant walks an inch, and then the rubber band is stretched an inch longer.

(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?]

(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 \(f(n)\).
41.1. Data structures

41.1.1. Furthest Neighbor

(20 pts.)
Let \( P = \{p_1, \ldots, p_n\} \) be a set of \( n \) points in the plane.

(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\(^\text{\textsuperscript{0}}\) algorithm to compute \( m = O(\log n) \) partitions \( \mathcal{P}_1, \ldots, \mathcal{P}_m \) 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 \( O(n \log n) \).

(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 \( X \), 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 \( P \setminus \{p\} \).

41.1.2. Free lunch.

(10 pts.)

1. (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. (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

\[
\max_{(a, b) \in S, a \in [\alpha, \beta]} b_i,
\]

where \( S \) is the current set of pairs.

3. (3 pts.) Using (b), describe an \( O(n \log n) \) time algorithm for computing the longest ascending subsequence given a sequence of \( n \) numbers.

\(^0\)There is a very nice and simple randomized algorithm for this problem, you can think about it if you are interested.
41.2. Divide and Conqueror

41.2.1. Divide-and-Conquer Multiplication

1. (5 pts.) Show how to multiply two linear polynomials \( ax + b \) and \( cx + d \) using only three multiplications. (Hint: One of the multiplications is \((a + b) \cdot (c + d)\).)

2. (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. (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

41.3.1. 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 \( 10n \).)

41.3.2. Common subsequence

Given two sequences, \( a_1, \ldots, a_n \) and \( b_1, \ldots, b_m \) of real numbers, We want to determine whether there is an \( i \geq 0 \), such that \( b_1 = a_{i+1}, b_2 = a_{i+2}, \ldots, b_m = a_{i+m} \). Show how to solve this problem in \( O(n \log n) \) time with high probability.

41.3.3. 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 \( p(x) \mod q(x) \) in \( O(n \log n) \) time. The remainder of \( r(x) = p(x) \mod q(x) \) is the unique polynomial of degree smaller than this of \( q(x) \), such that \( p(x) = q(x) \cdot d(x) + r(x) \), where \( d(x) \) is a polynomial.

Let \( p(x) = \sum_{i=0}^{n-1} a_i x^i \) be a given polynomial.

1. (4 pts.) Prove that \( p(x) \mod (x - z) = p(z) \), for all \( z \).

2. (4 pts.) We want to evaluate \( p(\cdot) \) on the points \( x_0, x_1, \ldots, x_{n-1} \). Let

\[
P_{ij}(x) = \prod_{k=i}^{j} (x - x_k)
\]

and

\[
Q_{ij}(x) = p(x) \mod P_{ij}(x).
\]

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. (4 pts.) Prove that for $i \leq k \leq j$, we have

$$\forall x \quad Q_k(x) = Q_j(x) \mod P_k(x)$$

and

$$\forall x \quad Q_{kj}(x) = Q_{ij}(x) \mod P_{kj}(x).$$

4. (8 pts.) Given an $O(n \log^2 n)$ time algorithm to evaluate $p(x_0), \ldots, p(x_{n-1})$. Here $x_0, \ldots, x_{n-1}$ are $n$ given real numbers.

### 41.4. Union-Find

#### 41.4.1. Linear time Union-Find,

(20 pts.)

1. (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?

2. (2 pts.) Same question, for rank $\lfloor (\lg n)/2 \rfloor$.

3. (4 pts.) Prove that in a set of $n$ elements, a sequence of $n$ consecutive Find operations take $O(n)$ time in total.

4. (2 pts.) Write a non-recursive version of Find with path compression.

5. (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.

6. (4 pts.) What happens in the same situation if only the path compression is used?

#### 41.4.2. Off-line Minimum

(20 pts.)

The off-line minimum problem asks us to maintain a dynamic set $T$ of elements from the domain $\{1, 2, \ldots, n\}$ 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 $\{1, 2, \ldots, n\}$ 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 $extracted[1 \ldots m]$, where for $i = 1, 2, \ldots, m$, $extracted[i]$ 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. (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.

2. (8 pts.)

   To develop an algorithm for this problem, we break the sequence $S$ into homogeneous subsequences. That is, we represent $S$ by

   $I_1, E, I_2, E, I_3, \ldots, I_m, E, I_{m+1},$

   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.
Argue that the array extracted returned by Off-Line-Minimum is correct.

3. (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.

4.1.4.3. 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(root[T]). Each node is assumed to be colored white prior to the walk.

```
LCA(u)
1 MAKESET(u)
2 ancestor[FIND(u)] ← u
3 for each child v of u in T
4    do LCA(v)
5    UNION(u, v)
6    ancestor[FIND(u)] ← u
7 color[u] ← BLACK
8 for each node v such that \{u, v\} ∈ P
9    do if color[v] = BLACK
10    then print “The least common ancestor of” u “and” v “is” ancestor[FIND(v)]
```

1. (4 pts.) Argue that line 10 is executed exactly once for each pair \{u, v\} ∈ P.
2. (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 $T$.
3. (6 pts.) Prove that LCA correctly prints the least common ancestor of $u$ and $v$ for each pair \{u, v\} ∈ P.
4. (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.
41.4.4. 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 \( A(x) = A_4(x) \). 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 \( \lim_{n \to \infty} \frac{\alpha(n)}{\log^*(n)} = 0 \).

4. (4 pts.) We define

\[
\log^{**} n = \min \left\{ i \geq 1 \left| \log^{*} \ldots \log^{*} n \leq 2 \right. \right. \right. \right. \right. i \text{times} \}
\]

(i.e., how many times do you have to take \( \log^* \) of a number before you get a number smaller than 2). Prove that \( \lim_{n \to \infty} \frac{\alpha(n)}{\log^{**}(n)} = 0 \).

5. (4 pts.) Prove that \( \log(\alpha(n)) \leq \alpha(\log^{**} n) \) for \( n \) large enough.

41.5. Lower bounds

41.5.1. Sort them Up

(20 pts.)

A sequence of real numbers \( x_1, \ldots, x_n \) is \( k \)-mixed, if there exists a permutation \( \pi \), such that \( x_{\pi(i)} \leq x_{\pi(i+1)} \) and \( |\pi(i) - i| \leq k \), for \( i = 1, \ldots, n - 1 \).

1. (10 pts.) Give a fast algorithm for sorting \( x_1, \ldots, x_n \).

2. (10 pts.) Prove a lower bound in the comparison model on the running time of your algorithm.

41.5.2. Another Lower Bound

(20 pts.)

Let \( b_1 \leq b_2 \leq b_3 \leq \ldots \leq b_k \) be \( k \) given sorted numbers, and let \( A \) be a set of \( n \) arbitrary numbers \( A = \{a_1, \ldots, a_n\} \), such that \( b_1 < a_i < b_k \), for \( i = 1, \ldots, n \).

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 \( r(a_1), \ldots, r(a_n) \) must take \( \Omega(n \log k) \) running time in the worst case.
41.6. Number theory

41.6.1. Some number theory.

(10 pts.)
1. (5 pts.) Prove that if gcd(m, n) = 1, then \(m^{\phi(n)} + n^{\phi(m)} \equiv 1 \pmod{mn}\).
2. (5 pts.) Give two distinct proofs that there are an infinite number of prime numbers.

41.6.2. Even More Number Theory

(10 pts.)
Prove that \(|P(n)| = \Omega(n^2)\), where \(P(n) = \{(a, b) \mid a, b \in \mathbb{Z}, 0 < a < b \leq n, \gcd(a, b) = 1\}\).

41.6.3. 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 \((2^m)^m\).
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 \(\binom{2m}{m} \geq 2^m/(2m)\).)

41.7. Sorting networks

41.7.1. 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 \(i = 1, 2, ..., n - 1\).

41.7.2. First sort, then partition

Suppose that we have \(2n\) elements \(< a_1, a_2, ..., a_{2n}>\) 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 \(< a_1, a_2, ..., a_n>\) and \(< a_{n+1}, a_{n+2}, ..., a_{2n}>\).

41.7.3. 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.)
41.7.4. Matrix Madness

(20 pts.) We can sort the entries of an $m \times m$ matrix by repeating the following procedure $k$ times:

1. Sort each odd-numbered row into monotonically increasing order.
2. Sort each even-numbered row into monotonically decreasing order.
3. Sort each column into monotonically increasing order.

1. (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.

2. (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.]

3. (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 you bound).

41.8. Max Cut

41.8.1. Splitting and splicing

Let $G = (V, E)$ be a graph with $n$ vertices and $m$ edges. A split 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 $\bar{\chi}(G) = \max_{V_1} m(V_1, V_2)$ 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 $m(V_1, V_2) \geq \bar{\chi}(G)/2$. [Hint: Start from an arbitrary split, and continue in a greedy fashion to improve it.]

Chapter 42

Exercises - Computational Geometry

This chapter include problems that are related to computational geometry.
42.1. Misc

42.1.1. 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 $q$.)

42.1.2. Sweeping

(20 pts.)

(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 $X$.

(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 $\bigcap_{h \in \mathcal{H}}$ in $O(n \log n)$ time. (Hint: use (a).)

(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)?

42.1.3. 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 form $q$. 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.
42.1.4. 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)?

42.1.5. Convexity revisited.

(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 $CH(S_1) \cap CH(S_2) \neq \emptyset$.

(b) Prove that any point $x$ which is a convex combination of $n$ points $p_1, \ldots, p_n$ in the plane, can be defined as a convex combination of three points of $p_1, \ldots, p_n$. (Hint: use (a) and induction on the number of points.)

(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 $CH(S_1) \cap CH(S_2) \neq \emptyset$, $S = S_1 \cup S_2$, and $S_1 \cap S_2 = \emptyset$. (Hint: Use (a) and induction on the dimension.)

42.1.6. 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.

42.1.7. 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 43

Fall 2014

43.1. Homeworks

43.1.1. Homework 0

1. (60 pts.) The numbers dance.
   (A) (30 pts.) The input is a multiset $X$ of $n$ positive integer numbers. Consider the following famous algorithm:
   \[
   \text{PlayItBen}(X) :
   \]
   \[
   \text{while } X \text{ contains more than two elements do}
   \]
   \[
   \text{two distinct elements } x_1, x_2 \text{ are chosen arbitrarily from } X,
   \]
   \[
   \text{such that } x_1 \leq x_2
   \]
   \[
   \text{if } x_1 = x_2 \text{ or } x_1 + 1 = x_2 \text{ then}
   \]
   \[
   X \leftarrow (X \setminus \{x_1, x_2\}) \cup \{x_1 + x_2\}
   \]
   \[
   \text{else}
   \]
   \[
   X \leftarrow (X \setminus \{x_1, x_2\}) \cup \{x_1 + 1, x_2 - 1\}
   \]
   Prove (maybe using induction, but you do not have to) that \text{PlayItBen} always terminates.
   (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 \text{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 $v_0, v_1, \ldots, v_m$ is of length $m$.
   (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 $P_u(n - 1, v) > 0$.
   (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 $\lim_{m \to \infty} P_u(m, v) = 1$. 

256
43.1.2. 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.

(A) (10 pts.)

**NOT SET COVER**

**Instance:** Collection $C$ of subsets of a finite set $S$.
**Target:** Compute the maximum $k$, and the sets $S_1, \ldots, S_k$ in $C$, such that $S \not\subseteq \bigcup_{j=1}^{k} S_j$.

(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 $U_1, \ldots, U_k$, such that the sum of the sizes of the items inside each $U_i$ is $B$ or more.

(C) (10 pts.)

**DOUBLE HITTING SET**

**Instance:** A ground set $U = \{1, \ldots, n\}$, and a set $\mathcal{F} = \{U_1, \ldots, U_m\}$ 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 $U_i$.

(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.

(E) (10 pts.)

**Cover by paths (edge disjoint).**

**Instance:** Graph $G = (V, E)$.
**Target:** Compute the minimum number $k$ of paths $\pi_1, \ldots, \pi_k$ that are edge disjoint, and their union cover all the edges in $G$.

(F) (10 pts.)

**Cover by paths (vertex disjoint).**

**Instance:** Graph $G = (V, E)$.
**Target:** Compute the minimum number $k$ of paths $\pi_1, \ldots, \pi_k$ that are vertex disjoint, and their union cover all the vertices in $G$. 

257
43.1.3. 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 $n$).

   (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 $[N] \times [N]$, where $N = 4 \lceil n^{2/3} \rceil$ and $[N] = \{1, \ldots, N\}$. Then solve the case the vertices of $G$ are contained in $[N] \times [n]$, and finally solve the general case where the vertices are contained in $[n] \times [n]$.)

   (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.)

   (C) (10 pts.) Given a breakable graph $G$, provide a polynomial time constant approximation algorithm for the largest independent set in $G$.

   (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: $v_1, \ldots, v_n$. Let $G_i$ be the induced subgraph of $G$ on $v_1, \ldots, v_i$. Formally, $G_i = (V_i, E_i)$, where $V_i = \{v_1, \ldots, v_i\}$ 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.

   (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 $V_1, \ldots, V_k$, 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 $\pi : V \to \{1, \ldots, n\}$ (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.
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 $1, \ldots, k_{i-1}$ (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 $\left( V^{(k)}, E^{(k)} \right)$, where $V^{(k)} = V \times V \times \cdots \times V$ is the set of all ordered $k$-tuples of vertices from $V$ and $E^{(k)}$ is defined so that $(v_1, v_2, \ldots, v_k)$ is adjacent to $(w_1, w_2, \ldots, w_k)$ if and only if for each $i$ (for $i = 1, \ldots, k$) either vertex $v_i$ is adjacent to $w_i$ in $G$, or else $v_i = w_i$.

(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 $\alpha$, then the largest clique in $G^{(k)}$ is $\alpha^k$, and vice versa.

(B) (10 pts.) Show an algorithm that is given a clique of size $\beta$ in $G^{(k)}$ and outputs a clique of size $\lceil \beta^{1/k} \rceil$ in $G$.

(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).)

(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.\(^3\)

43.1.4. Homework 3

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 $T$ 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

$$f(T) = \max_{i=1}^{n} (\|r_i - g_i\| + \|g_i - b_i\| + \|b_i - r_i\|)$$

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 $\alpha_i$ clients, and each client has to be in distance at most $r_i$ from it, for $i = 1, \ldots, m$. 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.

(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).

\(^3\)Can one prove that there is FPTAS in this case? I do not know.
(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 $C = \max_{(u,v) \in E} c(u, v)$.
(A) (3 pts.) Argue that a minimum cut of $G$ has capacity at most $C|E|$.
(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 $G$. 

260
maxFlow-By-Scaling($G, s, t$)
1 $C \leftarrow \max_{(u, v) \in E} c(u, v)$
2 initialize flow $f$ to 0
3 $K \leftarrow 2^\lfloor \log C \rfloor$
4 while $K \geq 1$ do {
5 \hspace{1em} while (there exists an augmenting path $p$ of capacity at least $K$) do {
6 \hspace{2em} augment flow $f$ along $p$
7 \hspace{1em} }\hspace{1em} $K \leftarrow K/2$
8 \hspace{1em} }
9 return $f$

(C) (3 pts.) Argue that maxFlow-By-Scaling returns a maximum flow.

(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.

(E) (4 pts.) Argue that the inner while loop of lines 5-6 is executed $O(|E|)$ times for each value of $K$.

(F) (4 pts.) Conclude that maxFlow-By-Scaling can be implemented so that it runs in $O(E^2 \log C)$ time.

43.1.5. Homework 4

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.) Tiedious Computations
Provide detailed solutions for the following problems, showing each pivoting stage separately.

(A) (5 pts.)
maximize $6x_1 + 3x_2 + 5x_3 + 2x_4$
subject to
$x_1 + x_2 + x_3 + x_4 = 1$
$x_1, x_2, x_3, x_4 \geq 0.$

(B) (5 pts.)
maximize $2x_1 + 4x_2$
subject to
$2x_1 + x_2 \leq 4$
$2x_1 + 3x_2 \leq 3$
$4x_1 + x_2 \leq 5$
$x_1 + 5x_2 \leq 1$
$x_1, x_2 \geq 0.$

(C) (5 pts.)
maximize $6x_1 + 2x_2 + 8x_3 + 9x_4$
subject to
$2x_1 + x_2 + x_3 + 3x_4 \leq 5$
$x_1 + 3x_2 + x_3 + 2x_4 \leq 3$
$x_1, x_2, x_3, x_4 \geq 0.$
minimize $2x_{12} + 8x_{13} + 3x_{14} + 2x_{23} + 7x_{24} + 3x_{34}$

subject to

$x_{12} + x_{13} + x_{14} \geq 1$
$-x_{12} + x_{23} + x_{24} = 0$
$-x_{13} - x_{23} + x_{34} = 0$
$x_{14} + x_{24} + x_{34} \leq 1$
$x_{12}, x_{13}, \ldots, x_{34} \geq 0.$

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, $K_1, K_2, \ldots, K_k$, 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

(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?

(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?

(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:

i. Those traveling from Ithaca to Newark (god only knows why).
ii. Those traveling from Newark to Boston (a very good idea).
iii. 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:

i. Y class: full coach.
ii. B class: nonrefundable.
iii. 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:

<table>
<thead>
<tr>
<th></th>
<th>Ithaca-Newark</th>
<th>Newark-Boston</th>
<th>Ithaca-Boston</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>300</td>
<td>160</td>
<td>360</td>
</tr>
<tr>
<td>B</td>
<td>220</td>
<td>130</td>
<td>280</td>
</tr>
<tr>
<td>M</td>
<td>100</td>
<td>80</td>
<td>140</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th></th>
<th>Ithaca-Newark</th>
<th>Newark-Boston</th>
<th>Ithaca-Boston</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>8</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>M</td>
<td>22</td>
<td>20</td>
<td>18</td>
</tr>
</tbody>
</table>

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 place 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.

(A) (5 pts.) What is the dual of the following LP?

\[
\begin{align*}
\text{maximize} & \quad x_1 - 2x_2 \\
\text{subject to} & \quad x_1 + 2x_2 - x_3 + x_4 \geq 0 \\
& \quad 4x_1 + 3x_2 + 4x_3 - 2x_4 \leq 3 \\
& \quad -x_1 - x_2 + 2x_3 + x_4 = 1 \\
& \quad x_2, x_3 \geq 0
\end{align*}
\]

(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?

(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 \( P \) denote the set of all the directed (simple) paths from \( s \) to \( t \) in \( G \).

Consider the following (very large) integer program:

\[
\begin{align*}
\text{minimize} & \quad \sum_{e \in E(G)} c_e x_e \\
\text{subject to} & \quad x_e \in \{0, 1\} \quad \forall e \in E(G) \\
& \quad \sum_{e \in \pi} x_e \geq 1 \quad \forall \pi \in P.
\end{align*}
\]

(A) (5 pts.) What does this IP computes?
(B) (5 pts.) Write down the relaxation of this IP into a linear program.

(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)?

(D) (5 pts.) The strong duality theorem states the following.

**Theorem 43.1.1.** If the primal LP problem has an optimal solution \( x^* = (x_1^*, \ldots, x_n^*) \) then the dual also has an optimal solution, \( y^* = (y_1^*, \ldots, y_m^*) \), 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.)

### 43.1.6. Homework 5

1. (20 pts.) Sorting networks stuff
   (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 \( i = 1, 2, \ldots, n - 1 \).
   (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.]
   (C) (3 pts.)
   Suppose that we have \( 2n \) elements \( (a_1, a_2, \ldots, a_{2n}) \) 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 \( (a_1, a_2, \ldots, a_n) \) and \( (a_{n+1}, a_{n+2}, \ldots, a_{2n}) \).
   (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 \( p(x) \mod q(x) \) in \( O(n \log n) \) time. The **remainder** of \( r(x) = p(x) \mod q(x) \) is the unique polynomial of degree smaller than this of \( q(x) \), such that \( p(x) = q(x) \ast d(x) + r(x) \), where \( d(x) \) is a polynomial.
   Let \( p(x) = \sum_{i=0}^{n-1} a_i x^i \) be a given polynomial.
   (A) (2 pts.) Prove that \( p(x) \mod (x - z) = p(z) \), for all \( z \).
   (B) (2 pts.) We want to evaluate \( p(\cdot) \) on the points \( x_0, x_1, \ldots, x_{n-1} \). Let

\[
P_{ij}(x) = \prod_{k=i}^{j} (x - x_k)
\]

and

\[
Q_{ij}(x) = p(x) \mod P_{ij}(x).
\]

Observe that the degree of \( Q_{ij} \) is at most \( j - i \).
Prove that, for all \( x, Q_{i,k}(x) = p(x_k) \) and \( Q_{0,n-1}(x) = p(x) \).
(C) (4 pts.) Prove that for \( i \leq k \leq j \), we have
\[
\forall x \quad Q_{i:k}(x) = Q_{i:j}(x) \mod P_{i:k}(x)
\]
and
\[
\forall x \quad Q_{k:j}(x) = Q_{i:j}(x) \mod P_{k:j}(x).
\]

(D) (4 pts.) Given an \( O(n \log^2 n) \) time algorithm to evaluate \( p(x_0), \ldots, p(x_{n-1}) \). Here \( x_0, \ldots, x_{n-1} \) are \( n \) given real numbers.

3. (20 pts.) Linear time Union-Find.

(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 \([\log n - 5]\), where there are \( n \) elements stored in the data-structure?

(B) (2 pts.) Same question, for rank \([\log n]/2\).

(C) (4 pts.) Prove that in a set of \( n \) elements, a sequence of \( n \) consecutive Find operations take \( O(n) \) time in total.

(D) (4 pts.) Write a non-recursive version of Find with path compression.

(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.

(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 \( X_1, X_2, \ldots \). Each \( X_j \) takes on one of \( n \) values. The \( i \)th value occurs with probability \( p_i \), where \( p_1 \geq p_2 \geq \ldots \geq p_n \). 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 \( \lceil \log(1/p_i) \rceil \) 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.

(A) Show that no codeword is the prefix of any other codeword.

(B) Let \( Z \) be the average number of bits appended for each random variable \( X_j \). Show that
\[
H(X_j) \leq z \leq H(X_j) + 1.
\]

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 \( X = (X_1, X_2, \ldots, X_n) \), where each \( X_i \) is independently 0 with probability \( p \) and 1 with probability \( 1 - p \). The sequences can be ordered lexicographically, so for \( x = (x_1, x_2, \ldots, x_n) \) and \( y = (y_1, y_2, \ldots, y_n) \), 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
\[
q(x) = \sum_{y < x} p(y).
\]

(A) Suppose we are given \( X = (X_1, X_2, \ldots, X_n) \). Explain how to compute \( q(X) \) in time \( O(n) \) (assume that any reasonable operation on real numbers takes constant time).

(B) Argue that the intervals \([q(x), q(x) + p(x)]\) are disjoint subintervals of \([0, 1]\).

(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 \([\log(1/p(X))] + 1\) binary digits to represent \( X \) in such a way that no codeword is the prefix of any other codeword.

(D) Given a codeword chosen as in (C), explain how to decompress it to determine the corresponding sequence \((X_1, X_2, \ldots, X_n)\).
(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.

(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 \( 1/n \).

(B) (5 pts.) Extraction to the limit,

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 \( \{0, \ldots, m - 1\} \). It follows that if we have \( k \) numbers chosen independently and uniformly at random from \( \{0, \ldots, m - 1\} \) 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.

(C) (2 pts.) Assume you have a (valid) prefix code with \( n \) codewords, where the \( i \)th codeword is made out of \( \ell_i \) bits. Prove that

\[
\sum_{i=1}^{n} \frac{1}{2^{\ell_i}} \leq 1.
\]

(D) (2 pts.) Let \( S = \sum_{i=1}^{10} 1/i^2 \). Consider a random variable \( X \) such that \( \Pr[X = i] = 1/(S i^2) \), for \( i = 1, \ldots, 10 \). Compute \( \mathbb{H}(X) \).

(E) (2 pts.) Let \( S = \sum_{i=1}^{10} 1/i^3 \). Consider a random variable \( X \) such that \( \Pr[X = i] = 1/(S i^3) \), for \( i = 1, \ldots, 10 \). Compute \( \mathbb{H}(X) \).

(F) (2 pts.) Let \( S(\alpha) = \sum_{i=1}^{10} 1/i^\alpha \), for \( \alpha > 1 \). Consider a random variable \( X \) such that \( \Pr[X = i] = 1/(S(\alpha) i^\alpha) \), for \( i = 1, \ldots, 10 \). Prove that \( \mathbb{H}(X) \) is either increasing or decreasing as a function of \( \alpha \) (you can assume that \( \alpha \) is an integer).

(G) (2 pts.) The conditional entropy \( \mathbb{H}(Y|X) \) is defined by

\[
\mathbb{H}(Y|X) = \sum_{x,y} \Pr[(X = x) \cap (Y = y)] \frac{1}{\Pr[Y = y|X = x]} \lg \Pr[Y = y|X = x].
\]

If \( Z = (X, Y) \), prove that

\[
\mathbb{H}(Z) = \mathbb{H}(X) + \mathbb{H}(Y|X).
\]
43.2. Midterm

1. **Coloring graphs. (25 pts.)**

For a set $S$, a balanced coloring $\phi$ 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 $\phi$ 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 $S$).

(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?

(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 $\mu = E[X]$?

(C) (5 pts.) Prove that $\Pr[X \geq 2\mu] \leq 1/2$.

(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 $C = \{c_1, \ldots, c_n\}$ 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 $Y = \{y_1, \ldots, y_k\}$ 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:

(A) (5 pts.) In the beginning of the $i$th iteration, if there are $\beta_i$ sets in $\mathcal{F}$ not hit yet, then there is an element in $S$ that hits at least $\beta_i/k$ of these sets.

(B) (10 pts.) Prove that for any $i$, we have that $\beta_{i+1} \leq \beta_i/c$, where $c > 1$ is some positive constant (what is the value of $c$ - provide a reasonable lower bound).

(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.

(D) (2 pts.) What is the quality of approximation provided by **GreedyHit**?
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:

   \[
   \text{minimize } \sum_{e \in E(G)} \text{cost}(e)x_e \\
   \text{subject to } x_e \in \{0, 1\} \quad \forall e \in E(G) \\
   \sum_{e \in \pi} x_e \geq 1 \quad \forall \pi \in \mathcal{P}.
   \]

   (A) (5 pts.) What does this IP compute?

   (B) (5 pts.) Write down the relaxation of this IP into a linear program.

   (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)?

   (D) (5 pts.) The strong duality theorem states the following.

   **Theorem 43.3.1.** If the primal LP problem has an optimal solution \( x^* = (x_1^*, \ldots, x_n^*) \) then the dual also has an optimal solution, \( y^* = (y_1^*, \ldots, y_m^*) \), 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 \( X = \langle X_1, X_2, \ldots, X_n \rangle \) be a sequence of \( n \) numbers generated by picking each \( X_i \) independently and uniformly from the range \( \{1, \ldots, n\} \).

   (A) (5 pts.) What is the entropy of \( X \)?

   (B) (5 pts.) Consider the sequence \( Y = \langle Y_1, \ldots, Y_n \rangle \) that results from sorting the sequence \( X \) in increasing order. For example, if \( X = (4, 1, 4, 1) \) then \( Y = (1, 1, 4, 4) \).

   Describe an encoding scheme that takes the sequence \( Y \) 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 \( Y \)? (Hint: Consider the differences sequence \( Y_1, Y_2 - Y_1, Y_3 - Y_2, \ldots, Y_n - Y_{n-1} \). And do not use Huffman’s encoding.)

   Demonstrate how your encoding scheme works for the sequence \( Y = (1, 1, 4, 6, 6, 6) \).

   (C) (5 pts.) Consider the set \( U \) of all sequences \( Y \) 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.)

   (D) (5 pts.) **Prove** an upper bound (as low as possible) on the entropy of \( Y \). (Proving a lower bound here seems quite hard and you do not have to do it.)
3. **Find kth smallest number. (20 pts.)**

This question asks you to design and analyze a randomized incremental algorithm to select the kth 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 $x_1, x_2, \ldots, x_n$. After any prefix $x_1, \ldots, x_{i-1}$ has been read, the algorithm has computed the kth smallest element in $x_1, \ldots, x_{i-1}$ (which is undefined if $i \leq k$), or if appropriate, some other invariant from which the kth 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.

**(A) (5 pts.)** Describe an efficient incremental algorithm for computing the kth smallest element (the more efficient it is, the better).

**(B) (5 pts.)** How many comparisons does your algorithm perform in the worst case?

**(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 $R = \{R_1, \ldots, R_n\}$ of n rectangles in the plane, and a set $P = \{p_1, \ldots, p_n\}$ 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 $R$ there is at least one point of $X$ that is contained in $R$.

Under the assumption that no rectangle of $R$ 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


271


Index

$(k, n)$ decoding function, 202
$(k, n)$ encoding function, 202
0/1-flow, 105

\[\text{FastEuclid, 243}\]
\[\text{FastExp, 42}\]
\[\text{FastExp, 42}\]

\[\text{Feasible, 143, 144}\]

\[\text{FibDP, 41}\]
\[\text{FibI, 41}\]
\[\text{FibR, 41}\]
\[\text{find, 177–182}\]
\[\text{FibI, 41}\]
\[\text{FlipCleaner, 175, 176}\]
\[\text{FlipCoins, 75}\]
\[\text{FlipCoins, 75}\]

\[\text{Ford-Fulkerson, 125}\]
\[\text{fpVertexCoverInner, 52}\]
\[\text{GreedyHit, 268}\]

\[\text{GreedySetCover, 59, 60}\]
\[\text{GreedyVertexCover, 49, 50}\]

\[\text{GreedySetCover, 59}\]
\[\text{Half-Cleaner, 174}\]
\[\text{IsMatch, 48}\]

\[\text{LPStartSolution, 143, 144}\]

\[\text{makeSet, 177–179}\]
\[\text{MatchNutsAndBolts, 74, 79}\]
\[\text{maxFlow-By-Scaling, 261}\]
\[\text{Merger, 175, 176}\]
\[\text{MinCut, 88, 89, 93}\]
\[\text{MinCutRep, 89, 90, 92}\]
\[\text{mtdFordFulkerson, 100–102}\]

\[\text{Partitions, 39}\]
\[\text{PartitionsI, 39}\]
\[\text{PartitionsI, 39}\]
\[\text{PartitionsI_C, 39}\]
\[\text{PartitionS_C, 39}\]

\[\text{perceptron, 185, 187, 188}\]
\[\text{PlayItBen, 256}\]
\[\text{QuickSelect, 76, 77}\]

\[\text{QuickSort, 74–79, 83, 227, 234}\]
\[\text{RandBit, 75}\]

randomized, 57
Relax, 218
RotateLeft, 84
RotateRight, 84
RotateUp, 84
Simplex, 139–141, 143, 144, 146, 153–156
SimplexInner, 144, 145
SlowEuclid, 243
SolveSubsetSum, 64
Sorter, 176
Trim, 66, 67
union, 177–179
WeirdEuclid, 244
WGreedySetCover, 60, 61
alternating BFS, 219
alternating cycle, 209
alternating path, 209
approximation
algorithm
minimization, 50
maximization problem, 57
Arithmetic coding, 207, 265
augmenting, 210
augmenting path, 99
average optimal cost, 60
average price, 60
backwards analysis, 226
bad, 267
balance, 267
balanced coloring, 267
basic solution, 146
Bin Packing
next fit, 68
bin packing
first fit, 68
binary code, 191
binary entropy, 196
binary symmetric channel, 201
bitonic sequence, 174
black-box access, 231
Bland’s rule, 146
blossom, 220
breakable, 258
capacity, 96, 100
cell, 228
Chernoff inequality, 80
Cholesky decomposition, 184
circulation, 107
classification, 186
classifying, 186
clause gadget, 21
clique, 17
clustering
k-center, 63
price, 62
CNF, 15, 16
collapsible, 166
coloring, 20
comparison network, 172
depth, 172
depth wire, 172
half-cleaner, 174
sorting network, 172
conditional entropy, 208, 266
conditional probability, 86
configurations, 44
congestion, 159
contraction
directed edge, 86
convex, 46
convex polygon, 46
convex programming, 182
convolution, 170
cost, 120
critical, 230
cut, 85, 100
minimum, 86
cuts, 85
Cycle
directed, 117
Eulerian, 54
DAG, 44, 172, 213
decision problem, 13
diagonal, 46
digraph, 117
directed, 96
directed acyclic graph, 44
directed cut, 100
directed Cycle
average cost, 117
disjoint paths
directed edge, 105
distance
point from set, 62
dot product, 186
double hitting set, 257
dynamic programming, 40
edge cover, 150
edge disjoint
paths, 105
edit distance, 42
entering, 145
entropy, 195
   binary, 196
Eulerian, 54
Eulerian cycle, 22, 54
expectation, 72
   conditional, 73
   linearity, 72
facility location problem, 62
Fast Fourier Transform, 166
feasible, 113
feasible permutations, 223
finite metric, 231
fixed parameter tractable, 37, 52
Flow
   capacity, 96
   Integrality theorem, 102
   value, 97
flow, 96, 97
   0/1-flow, 105
   circulation
      minimum-cost, 119
      valid, 120
      cost, 119
      efficient, 125
      min cost
      reduced cost, 123
      minimum-cost, 119
      residual graph, 98
      residual network, 98
flow network, 96
flower, 220
flow
   flow across a cut, 100
Ford-Fulkerson method, 100
FPTAS, 65
fully polynomial time approximation scheme, 65
gadget, 20
   color generating, 20
Galton-Watson processes, 93
greedy algorithms, 49
grid, 228, 258
ground set, 58, 257
Hamiltonian cycle, 22
high probability, 75
Huffman coding, 193
hypergraph, 58
image segmentation problem, 111
imaginary, 170
independent, 86, 267
independent set, 19, 150
induced subgraph, 17, 51
isolator, 258
kernel technique, 189
labeled, 186
leader, 179
learning, 186
leaving, 145
length, 209
lexicographically, 154
linear classifier $h$, 186
linear program
   dual, 148
Linear programming
   standard form, 139
linear programming, 138
   constraint, 138
   dual, 147
   dual program, 148
   entering variable, 145
   pivoting, 142
   primal, 147
   primal problem, 148
   primal program, 148
   slack form, 140
   slack variable, 140
   variable
      basic, 140
      non-basic, 140
Linearity of expectations, 57
linearization, 189
longest ascending subsequence, 48
LP, 138–154, 156–159
lucky, 233
Macaroni sort, 171
Markov's Inequality, 80
matched, 209
matching, 54, 104, 209
   bridge, 219
   free edge, 209
   free vertex, 209
   matching edge, 209
   maximum matching, 104
Vertex Cover, 19
Min, 49–53
Minimization, 49
Weight Set Cover, 60

problem
2SAT, 29, 61
3CNF, 24
3COLORABLE, 35
3Colorable, 20, 21
3DM, 26
3SAT, 15, 16, 18, 20, 24, 25, 29, 56, 58
3SUM, 225, 226
A, 18
AFWLB, 125, 126
Bin Packing, 68
Circuit Satisfiability, 13–15
Clique, 18, 19
CSAT, 15, 16
EDGE COVER, 35
EXACT-COVER-BY-3-SETS, 29
formula satisfiability, 14, 15
Hamiltonian Cycle, 22, 53
Hamiltonian Path, 125
Hamiltonian path, 126
Independent Set, 19, 20, 36
LONGEST PATH, 35
Max 3SAT, 56, 57
MAX CUT, 35, 182
MAX SAT, 34
MAX SPANNING TREE, 35
MaxClique, 17, 18
MIN CUT, 35
MIN EQUIVALENT DIGRAPH, 35
MIN SPANNING TREE, 35
minimization, 49
Not-3SAT, 30
NPC, 24
Partition, 26, 27, 68
PRIMALITY, 35
PROB, 65
reduction, 27
SAT, 14–16
Set Cover, 50, 58, 60, 150, 157
SetCover, 60
SHORTEST PATH, 35
Sorting Nuts and Bolts, 73
Subgraph Isomorphism, 27
Subset Sum, 24–26, 29, 64, 65
TRANSITIVE REDUCTION, 35
TRAVELING SALESMAN PROBLEM, 35

TSP, 23, 53, 54
Uniqueness, 224, 225
uniqueness, 230
VACATION TOUR PROBLEM, 35
Vec Subset Sum, 24, 25
VERTEX COVER, 35
Vertex Cover, 20, 22, 23, 36, 49, 69, 155
VertexCover, 69
VertexCover, 50
Weighted Vertex Cover, 155, 157
X, 18
profit, 113
PTAS, 65

Quick Sort
lucky, 78

quotation
– A Scanner Darkly, Philip K. Dick, 201
– Defeat, Arkady and Boris Strugatsky, 199
– J.M. Coetzee, 165
– The first world war, John Keegan., 38
– The roots of heaven, Romain Gary, 44
–Romain Gary, The talent scout., 195
A confederacy of Dunces, John Kennedy Toole, 11
Foundations, Leopold Staff, 85

random variable, 72
random variables
independent, 72
rank, 74, 76
real, 170
rearrangeable, 131
reduced cost, 123
relaxation, 157
remainder, 248, 264
residual capacity, 98, 99
residual graph, 120
rounding, 156
running-time
amortized, 177
expected, 74

see, 58
set system, 58
shortcutting, 54
sidelength, 228
sink, 96, 97
slack form, 140
sorting network
bitonic sorter, 174
running time, 172