CS 583: Approximation Algorithms

Chandra Chekuri

September 23, 2021

\[1\text{Dept. of Computer Science, University of Illinois, Urbana, IL 61820. Email: chekuri@illinois.edu.}\]
Contents

1 Introduction 4
  1.1 Formal Aspects ................................. 7
    1.1.1 NP Optimization Problems .................... 7
    1.1.2 Relative Approximation ...................... 8
    1.1.3 Additive Approximation ...................... 8
    1.1.4 Hardness of Approximation ................. 9
  1.2 Designing Approximation Algorithms .......... 11

2 Covering Problems 13
  2.1 Greedy for Set Cover and Maximum Coverage ........ 14
    2.1.1 Greedy Algorithm .......................... 14
    2.1.2 Analysis of Greedy Cover .................. 15
    2.1.3 Dominating Set ............................. 18
  2.2 Vertex Cover ................................ 19
    2.2.1 A 2-approximation for Vertex Cover .......... 19
    2.2.2 Set Cover with small frequencies .......... 20
  2.3 Vertex Cover via LP ........................... 20
  2.4 Set Cover via LP ............................. 23
    2.4.1 Deterministic Rounding ..................... 23
    2.4.2 Randomized Rounding ........................ 24
    2.4.3 Dual-fitting .............................. 28
    2.4.4 Greedy for implicit instances of Set Cover 30
  2.5 Submodularity ................................ 30
    2.5.1 Submodular Set Cover ...................... 31
    2.5.2 Submodular Maximum Coverage ............. 32
  2.6 Covering Integer Programs (CIPs) ............. 32

3 Knapsack 34
  3.1 The Knapsack Problem .......................... 34
    3.1.1 A Greedy Algorithm ....................... 35
## CONTENTS

3.1.2 A Polynomial Time Approximation Scheme . . . . . . . . . 36
3.1.3 Rounding and Scaling . . . . . . . . . . . . . . . . . . . . . 38
3.2 Other Problems . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

4 Packing Problems . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.1 Maximum Independent Set Problem in Graphs . . . . . . . . . . 41
  4.1.1 Elimination Orders and MIS . . . . . . . . . . . . . . . . . 44
4.2 The efficacy of the Greedy algorithm for a class of Independence Families . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.3 Randomized Rounding with Alteration for Packing Problems . . 47
4.4 Packing Integer Programs (PIPs) . . . . . . . . . . . . . . . . . . 50
  4.4.1 Randomized Rounding with Alteration for PIPs . . . . . . 50

5 Load Balancing and Bin Packing . . . . . . . . . . . . . . . . . . . 56
5.1 Load Balancing / MultiProcessor Scheduling . . . . . . . . . . . 56
  5.1.1 Problem Description . . . . . . . . . . . . . . . . . . . . . . 57
  5.1.2 Greedy Algorithm . . . . . . . . . . . . . . . . . . . . . . . 57
  5.1.3 A PTAS for Multi-Processor Scheduling . . . . . . . . . . . 59
  5.1.4 Section Notes . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.2 Bin Packing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
  5.2.1 Problem Description . . . . . . . . . . . . . . . . . . . . . . 62
  5.2.2 Greedy Approaches . . . . . . . . . . . . . . . . . . . . . . . 62
  5.2.3 (Asymptotic) PTAS for Bin Packing . . . . . . . . . . . . . . 63
  5.2.4 Asymptotic PTAS for Bin Packing . . . . . . . . . . . . . . . 65
  5.2.5 Section Notes . . . . . . . . . . . . . . . . . . . . . . . . . . 66

6 Unrelated Machine Scheduling and Generalized Assignment . . . . 67
6.1 Scheduling on Unrelated Parallel Machines . . . . . . . . . . . . 67
6.2 Generalized Assignment Problem . . . . . . . . . . . . . . . . . . . 71
  6.2.1 Shmoys-Tardos Rounding . . . . . . . . . . . . . . . . . . . 73
  6.2.2 Iterative Rounding . . . . . . . . . . . . . . . . . . . . . . . 77
6.3 Maximization version of GAP . . . . . . . . . . . . . . . . . . . . 81
6.4 Bibliographic Notes . . . . . . . . . . . . . . . . . . . . . . . . . . 82

7 Congestion Minimization in Networks . . . . . . . . . . . . . . . . . 83
7.1 Congestion Minimization and VLSI Routing . . . . . . . . . . . . 83
7.2 Min-max Integer Programs . . . . . . . . . . . . . . . . . . . . . . 89

A Basic Feasible Solutions to LPs and the Rank Lemma . . . . . . . . 97
  A.0.1 Some Examples . . . . . . . . . . . . . . . . . . . . . . . . . . 98
  A.0.2 Connection to Caratheodary’s Theorem . . . . . . . . . . . . 99
B Probabilistic Inequalities 100
Chapter 1

Introduction

These are lecture notes for a course on approximation algorithms.

Course Objectives

1. To appreciate that not all intractable problems are the same. NP optimization problems, identical in terms of exact solvability, can appear very different from the approximation point of view. This sheds light on why, in practice, some optimization problems (such as Knapsack) are easy, while others (like Clique) are extremely difficult.

2. To learn techniques for design and analysis of approximation algorithms, via some fundamental problems.

3. To build a toolkit of broadly applicable algorithms/heuristics that can be used to solve a variety of problems.

4. To understand reductions between optimization problems, and to develop the ability to relate new problems to known ones.

The complexity class \( P \) contains the set of problems that can be solved in polynomial time. From a theoretical viewpoint, this describes the class of tractable problems, that is, problems that can be solved efficiently. The class \( \text{NP} \) is the set of problems that can be solved in non-deterministic polynomial time, or equivalently, problems for which a solution can be verified in polynomial time. \( \text{NP} \) contains many interesting problems that often arise in practice, but there is good reason to believe \( P \neq \text{NP} \). That is, it is unlikely that there exist algorithms to solve \( \text{NP} \) optimization problems efficiently, and so we often resort to heuristic methods to solve these problems.

Heuristic approaches include backtrack search and its variants, mathematical programming methods, local search, genetic algorithms, tabu search, simulated
annealing etc. Some methods are guaranteed to find an optimal solution, though they may take exponential time; others are guaranteed to run in polynomial time, though they may not return a (optimal) solution. Approximation algorithms are (typically) polynomial time heuristics that do not always find an optimal solution but they are distinguished from general heuristics in providing guarantees on the quality of the solution they output.

**Approximation Ratio:** To give a guarantee on solution quality, one must first define what we mean by the quality of a solution. We discuss this more carefully later. For now, note that each instance of an optimization problem has a set of feasible solutions. The optimization problems we consider have an objective function which assigns a (real/rational) number/value to each feasible solution of each instance $I$. The goal is to find a feasible solution with minimum objective function value or maximum objective function value. The former problems are minimization problems and the latter are maximization problems.

For each instance $I$ of a problem, let $\text{OPT}(I)$ denote the value of an optimal solution to instance $I$. We say that an algorithm $A$ is an $\alpha$-approximation algorithm for a problem if, for every instance $I$, the value of the feasible solution returned by $A$ is within a (multiplicative) factor of $\alpha$ of $\text{OPT}(I)$. Equivalently, we say that $A$ is an approximation algorithm with approximation ratio $\alpha$. For a minimization problem we would have $\alpha \geq 1$ and for a maximization problem we would have $\alpha \leq 1$. However, it is not uncommon to find in the literature a different convention for maximization problems where one says that $A$ is an $\alpha$-approximation algorithm if the value of the feasible solution returned by $A$ is at least $\frac{1}{\alpha} \cdot \text{OPT}(I)$; the reason for using convention is so that approximation ratios for both minimization and maximization problems will be $\geq 1$. In this course we will for the most part use the convention that $\alpha \geq 1$ for minimization problems and $\alpha \leq 1$ for maximization problems.

**Remarks:**

1. The approximation ratio of an algorithm for a minimization problem is the maximum (or supremum), over all instances of the problem, of the ratio between the values of solution returned by the algorithm and the optimal solution. Thus, it is a bound on the worst-case performance of the algorithm.

2. The approximation ratio $\alpha$ can depend on the size of the instance $I$, so one should technically write $\alpha(|I|)$.

3. A natural question is whether the approximation ratio should be defined in an additive sense. For example, an algorithm has an $\alpha$-approximation for a minimization problem if it outputs a feasible solution of value at most
OPT(I) + \alpha for all I. This is a valid definition and is the more relevant one in some settings. However, for many \textbf{NP} problems it is easy to show that one cannot obtain any interesting additive approximation (unless of course \(P = NP\)) due to scaling issues. We will illustrate this via an example later.

\textbf{Pros and cons of the approximation approach: } Some advantages to the approximation approach include:

1. It explains why problems can vary considerably in difficulty.

2. The analysis of problems and problem instances distinguishes easy cases from difficult ones.

3. The worst-case ratio is \textit{robust} in many ways. It allows \textit{reductions} between problems.

4. Approximation algorithmic ideas/tools/relaxations are valuable in developing heuristics, including many that are practical and effective.

5. Quantification of performance via a concrete metric such as the approximation ratio allows for innovation in algorithm design and has led to many new ideas.

As a bonus, many of the ideas are beautiful and sophisticated, and involve connections to other areas of mathematics and computer science.

Disadvantages include:

1. The focus on \textit{worst-case measures} risks ignoring algorithms or heuristics that are practical or perform well on \textit{average}.

2. Unlike, for example, integer programming, there is often no incremental/continuous tradeoff between the running time and quality of solution.

3. Approximation algorithms are often limited to cleanly stated problems.

4. The framework does not (at least directly) apply to decision problems or those that are inapproximable.

\textbf{Approximation as a broad lens}

The use of approximation algorithms is not restricted solely to \textbf{NP}-Hard optimization problems. In general, ideas from approximation can be used to solve many problems where finding an exact solution would require too much of any resource.
A resource we are often concerned with is time. Solving NP-Hard problems exactly would (to the best of our knowledge) require exponential time, and so we may want to use approximation algorithms. However, for large data sets, even polynomial running time is sometimes unacceptable. As an example, the best exact algorithm known for the MATCHING problem in general graphs requires $O(m\sqrt{n})$ time; on large graphs, this may not be practical. In contrast, a simple greedy algorithm takes near-linear time and outputs a matching of cardinality at least $1/2$ that of the maximum matching; moreover there have been randomized sub-linear time algorithms as well.

Another often limited resource is space. In the area of data streams/streaming algorithms, we are often only allowed to read the input in a single pass, and given a small amount of additional storage space. Consider a network switch that wishes to compute statistics about the packets that pass through it. It is easy to exactly compute the average packet length, but one cannot compute the median length exactly. Surprisingly, though, many statistics can be approximately computed.

Other resources include programmer time (as for the MATCHING problem, the exact algorithm may be significantly more complex than one that returns an approximate solution), or communication requirements (for instance, if the computation is occurring across multiple locations).

1.1 Formal Aspects

1.1.1 NP Optimization Problems

In this section, we cover some formal definitions related to approximation algorithms. We start from the definition of optimization problems. A problem is simply an infinite collection of instances. Let $\Pi$ be an optimization problem. $\Pi$ can be either a minimization or maximization problem. Instances $I$ of $\Pi$ are a subset of $\Sigma^*$ where $\Sigma$ is a finite encoding alphabet. For each instance $I$ there is a set of feasible solutions $S(I)$. We restrict our attention to real/rational-valued optimization problems; in these problems each feasible solution $S \in S(I)$ has a value $val(S, I)$. For a minimization problem $\Pi$ the goal is, given $I$, find $\text{OPT}(I) = \min_{S \in S(I)} val(S, I)$.

Now let us formally define NP optimization (NPO) which is the class of optimization problems corresponding to $NP$.

Definition 1.1. $\Pi$ is in NPO if

- Given $x \in \Sigma^*$, there is a polynomial-time algorithm that decide if $x$ is a valid instance of $\Pi$. That is, we can efficiently check if the input string is well-formed. This is a basic requirement that is often not spelled out.
For each $I$, and $S \in S(I)$, $|S| \leq \text{poly}(|I|)$. That is, the solution are of size polynomial in the input size.

There exists a poly-time decision procedure that for each $I$ and $S \in \Sigma^*$, decides if $S \in S(I)$. This is the key property of NP; we should be able to verify solutions efficiently.

$\text{val}(I, S)$ is a polynomial-time computable function.

We observe that for a minimization NPO problem $\Pi$, there is a associated natural decision problem $L(\Pi) = \{(I, B) : \text{OPT}(I) \leq B\}$ which is the following: given instance $I$ of $\Pi$ and a number $B$, is the optimal value on $I$ at most $B$? For maximization problem $\Pi$ we reverse the inequality in the definition.

**Lemma 1.1.** $L(\Pi)$ is in NP if $\Pi$ is in NPO.

### 1.1.2 Relative Approximation

When $\Pi$ is a minimization problem, recall that we say an approximation algorithm $A$ is said to have approximation ratio $\alpha$ iff

- $A$ is a polynomial time algorithm
- for all instance $I$ of $\Pi$, $A$ produces a feasible solution $A(I)$ s.t. $\text{val}(A(I), I) \leq \alpha \text{ val}(\text{OPT}(I), I)$. (Note that $\alpha \geq 1$.)

Approximation algorithms for maximization problems are defined similarly. An approximation algorithm $A$ is said to have approximation ratio $\alpha$ iff

- $A$ is a polynomial time algorithm
- for all instance $I$ of $\Pi$, $A$ produces a feasible solution $A(I)$ s.t. $\text{val}(A(I), I) \geq \alpha \text{ val}(\text{OPT}(I), I)$. (Note that $\alpha \leq 1$.)

For maximization problems, it is also common to see use $1/\alpha$ (which must be $\geq 1$) as approximation ratio.

### 1.1.3 Additive Approximation

Note that all the definitions above are about relative approximations; one could also define additive approximations. $A$ is said to be an $\alpha$-additive approximation algorithm, if for all $I$, $\text{val}(A(I)) \leq \text{OPT}(I) + \alpha$. Most NPO problems, however, do not allow any additive approximation ratio because $\text{OPT}(I)$ has a scaling property.
To illustrate the scaling property, let us consider Metric-TSP. Given an instance $I$, let $I_\beta$ denote the instance obtained by increasing all edge costs by a factor of $\beta$. It is easy to observe that for each $S \in \mathcal{S}(I) = \mathcal{S}(I_\beta)$, $val(S, I_\beta) = \beta \cdot val(S, I_\beta)$ and $\text{OPT}(I_\beta) = \beta \cdot \text{OPT}(I)$. Intuitively, scaling edge by a factor of $\beta$ scales the value by the same factor $\beta$. Thus by choosing $\beta$ sufficiently large, we can essentially make the additive approximation (or error) negligible.

**Lemma 1.2.** Metric-TSP does not admit an $\alpha$ additive approximation algorithm for any polynomial-time computable $\alpha$ unless $P = NP$.

**Proof.** For simplicity, suppose every edge has integer cost. For the sake of contradiction, suppose there exists an additive $\alpha$ approximation $\mathcal{A}$ for Metric-TSP. Given $I$, we run the algorithm on $I_\beta$ and let $S$ be the solution, where $\beta = 2\alpha$. We claim that $S$ is the optimal solution for $I$. We have $val(S, I) = val(S, I_\beta)/\beta \leq OPT(I_\beta)/\beta + \alpha/\beta = OPT(I) + 1/2$, as $\mathcal{A}$ is $\alpha$-additive approximation. Thus we conclude that $\text{OPT}(I) = val(S, I)$, since $\text{OPT}(I) \leq val(S, I)$, and $\text{OPT}(I), val(S, I)$ are integers. This is impossible unless $P = NP$. ■

Now let us consider two problems which allow additive approximations. In the Planar Graph Coloring, we are given a planar graph $G = (V, E)$. We are asked to color all vertices of the given graph $G$ such that for any $vw \in E$, $v$ and $w$ have different colors. The goal is to minimize the number of different colors. It is known that to decide if a planar graph admits 3-coloring is NP-complete [62], while one can always color any planar graph $G$ with using 4 colors (this is the famous 4-color theorem) [4, 64]. Further, one can efficiently check whether a graph is 2-colorable (that is, if it is bipartite). Thus, the following algorithm is a 1-additive approximation for Planar Graph Coloring: If the graph is bipartite, color it with 2 colors; otherwise, color with 4 colors.

As a second example, consider the Edge Coloring Problem, in which we are asked to color edges of a given graph $G$ with the minimum number of different colors so that no two adjacent edges have different colors. By Vizing’s theorem [66], we know that one can color edges with either $\Delta(G)$ or $\Delta(G) + 1$ different colors, where $\Delta(G)$ is the maximum degree of $G$. Since $\Delta(G)$ is a trivial lower bound on the minimum number, we can say that the Edge Coloring Problem allows a 1-additive approximation. Note that the problem of deciding whether a given graph can be edge colored with $\Delta(G)$ colors is NP-complete [32].

### 1.1.4 Hardness of Approximation

Now we move to hardness of approximation.
**Definition 1.2** (Approximability Threshold). Given a minimization optimization problem $\Pi$, it is said that $\Pi$ has an approximation threshold $\alpha^*(\Pi)$, if for any $\epsilon > 0$, $\Pi$ admits a $\alpha^*(\Pi) + \epsilon$ approximation but if it admits a $\alpha^*(\Pi) - \epsilon$ approximation then $P = NP$.

If $\alpha^*(\Pi) = 1$, it implies that $\Pi$ is solvable in polynomial time. Many NPO problems $\Pi$ are known to have $\alpha^*(\Pi) > 1$ assuming that $P \neq NP$. We can say that approximation algorithms try to decrease the upper bound on $\alpha^*(\Pi)$, while hardness of approximation attempts to increase lower bounds on $\alpha^*(\Pi)$.

To prove hardness results on NPO problems in terms of approximation, there are largely two approaches; a direct way by reduction from NP-complete problems and an indirect way via gap reductions. Here let us take a quick look at an example using a reduction from an NP-complete problem.

In the (metric) $k$-center problem, we are given an undirected graph $G = (V, E)$ and an integer $k$. We are asked to choose a subset of $k$ vertices from $V$ called centers. The goal is to minimize the maximum distance to a center, i.e. $\min_{S \subseteq V, |S| = k} \max_{v \in V} \text{dist}_G(v, S)$, where $\text{dist}_G(v, S) = \min_{u \in S} \text{dist}_G(u, v)$.

The $k$-center problem has approximation threshold 2, since there are a few 2-approximation algorithms for $k$-center and there is no $2 - \epsilon$ approximation algorithm for any $\epsilon > 0$ unless $P = NP$. We can prove the inapproximability using a reduction from the decision version of Dominating Set: Given an undirected graph $G = (V, E)$ and an integer $k$, does $G$ have a dominating set of size at most $k$? A set $S \subseteq V$ is said to be a dominating set in $G$ if for all $v \in V$, $v \in S$ or $v$ is adjacent to some $u$ in $S$. Dominating Set is known to be NP-complete.

**Theorem 1.3** ([34]). Unless $P = NP$, there is no $2 - \epsilon$ approximation for $k$-center for any fixed $\epsilon > 0$.

**Proof.** Let $I$ be an instance of Dominating Set Problem consisting of graph $G = (V, E)$ and integer $k$. We create an instance $I'$ of $k$-center while keeping graph $G$ and $k$ the same. If $I$ has a dominating set of size $k$ then $\text{OPT}(I') = 1$, since every vertex can be reachable from the Dominating Set by at most one hop. Otherwise, we claim that $\text{OPT}(I') \geq 2$. This is because if $\text{OPT}(I') < 2$, then every vertex must be within distance 1, which implies the $k$-center that witnesses $\text{OPT}(I')$ is a dominating set of $I$. Therefore, the $(2 - \epsilon)$ approximation for $k$-center can be used to solve the Dominating Set Problem. This is impossible, unless $P = NP$. 

$\blacksquare$
1.2 Designing Approximation Algorithms

How does one design and more importantly analyze the performance of approximation algorithms? This is a non-trivial task and the main goal of the course is to expose you to basic and advanced techniques as well as central problems. The purpose of this section is to give some high-level insights. We start with how we design polynomial-time algorithms. Note that approximation makes sense mainly in the setting where one can find a feasible solution relatively easily but finding an optimum solution is hard. In some cases finding a feasible solution itself may involve some non-trivial algorithm, in which case it is useful to properly understand the structural properties that guarantee feasibility, and then build upon it.

Some of the standard techniques we learn in basic and advanced undergraduate algorithms courses are recursion based methods such as divide and conquer, dynamic programming, greedy, local search, combinatorial optimization via duality, and reductions to existing problems. How do we adapt these to the approximation setting? Note that intractability implies that there are no efficient characterizations of the optimum solution value.

Greedy and related techniques are often fairly natural for many problems and simple heuristic algorithms often suggest themselves for many problems. (Note that the algorithms may depend on being able to solve some existing problem efficiently. Thus, knowing a good collection of general poly-time solvable problems is often important.) The main difficulty is in analyzing their performance. The key challenge here is to identify appropriate lower bounds on the optimal value (assuming that the problem is a minimization problem) or upper bounds on the optimal value (assuming that the problem is a maximization problem). These bounds allow one to compare the output of the algorithm and prove an approximation bound. In designing poly-time algorithms we often prove that greedy algorithms do not work. We typically do this via examples. This skill is also useful in proving that some candidate algorithm does not give a good approximation. Often the bad examples lead one to a new algorithm.

How does one come up with lower or upper bounds on the optimum value? This depends on the problem at hand and knowing some background and related problems. However, one would like to find some automatic ways of obtaining bounds. This is often provided via linear programming relaxations and more advanced convex programming methods including semi-definite programming, lift-and-project hierarchies etc. The basic idea is quite simple. Since integer linear programming is NP-Complete one can formulate most discrete optimization problems easily and “naturally” as an integer program. Note that there may be many different ways of expressing a given problem as an integer program. Of course we cannot solve the integer program but we
can solve the linear-programming relaxation which is obtained by removing the integrality constraints on the variables. Thus, for each instance \( I \) of a given problem we can obtain an LP relaxation \( LP(I) \) which we typically can be solve in polynomial-time. This automatically gives a bound on the optimum value since it is a relaxation. How good is this bound? It depends on the problem, of course, and also the specific LP relaxation. How do we obtain a feasible solution that is close to the bound given by the LP relaxation. The main technique here is to round the fractional solution \( x \) to an integer feasible solution \( x' \) such that \( x'' \)'s value is close to that of \( x \). There are several non-trivial rounding techniques that have been developed over the years that we will explore in the course. We should note that in several cases one can analyze combinatorial algorithms via LP relaxations even though the LP relaxation does not play any direct role in the algorithm itself. Finally, there is the question of which LP relaxation to use. Often it is required to “strengthen” an LP relaxation via addition of constraints to provide better bounds. There are some automatic ways to strengthen any LP and often one also needs problem specific ideas.

Local search is another powerful technique and the analysis here is not obvious. One needs to relate the value of a local optimum to the value of a global optimum via various exchange properties which define the local search heuristic. For a formal analysis it is necessary to have a good understanding of the problem structure.

Finally, dynamic programming plays a key role in the following way. Its main use is in solving to optimality a restricted version of the given problem or a subroutine that is useful as a building block. How does one obtain a restricted version? This is often done by some clever pre-processing of a given instance.

Reductions play a very important role in both designing approximation algorithms and in proving inapproximability results. Often reductions serve as a starting point in developing a simple and crude heuristic that allows one to understand the structure of a problem which then can lead to further improvements.

Discrete optimization problems are brittle — changing the problem a little can lead to substantial changes in the complexity and approximability. Nevertheless it is useful to understand problems and their structure in broad categories so that existing results can be leveraged quickly and robustly. Thus, some of the emphasis in the course will be on classifying problems and how various parameters influence the approximability.
Chapter 2

Covering Problems

Part of these notes were scribed by Abul Hassan Samee and Lewis Tseng.

Packing and Covering problems together capture many important problems in combinatorial optimization. We will discuss several covering problems in this chapter. Two canonical one problems are Minimum Vertex Cover and its generalization Minimum Set Cover. (Typically we will omit the use of the qualifiers minimum and maximum since this is often clear from the definition of the problem and the context.) They play an important role in the study of approximation algorithms.

A vertex cover in an undirected graph \( G = (V, E) \) is a set \( S \subseteq V \) of vertices such that for each edge \( e \in E \), at least one of its end points is in \( S \). It is also called a node cover. In the Vertex Cover problem, our goal is to find a smallest vertex cover of \( G \). In the weighted version of the problem, a weight function \( w : V \rightarrow \mathbb{R}^+ \) is given, and our goal is to find a minimum weight vertex cover of \( G \). The unweighted version of the problem is also known as Cardinality Vertex Cover. Note that we are picking vertices to cover the edges. Vertex Cover is NP-Hard and is on the list of problems in Karp’s list.

In the Set Cover problem the input is a set \( U \) of \( n \) elements, and a collection \( S = \{S_1, S_2, \ldots, S_m\} \) of \( m \) subsets of \( U \) such that \( \bigcup_i S_i = U \). Our goal in the Set Cover problem is to select as few subsets as possible from \( S \) such that their union covers \( U \). In the weighted version each set \( S_i \) has a non-negative weight \( w_i \) the goal is to find a set cover of minimum weight. Closely related to the Set Cover problem is the Maximum Coverage problem. In this problem the input is again \( U \) and \( S \) but we are also given an integer \( k \leq m \). The goal is to select \( k \) subsets from \( S \) such that their union has the maximum cardinality. Note that Set Cover is a minimization problem while Maximum Coverage is a maximization problem. Set Cover is essentially equivalent to the Hitting Set problem. In Hitting Set the input is \( U \) and \( S \) but the goal is to pick the smallest number of
elements of \( \mathcal{U} \) that cover the given sets in \( \mathcal{S} \). In other words we are seeking a set cover in the dual set system. It is easy to see \textsc{Vertex Cover} is a special case of \textsc{Set Cover}.

\textsc{Set Cover} is an important problem because in discrete optimization. In the standard definition the set system is given \textit{explicitly}. In many applications the set system is \textit{implicit}, and often exponential in the explicit part of the input; nevertheless such set systems are ubiquitous and one can often obtain exact or approximation algorithms. As an example consider the well known \textsc{MST} problem in graphs. One way to phrase \textsc{MST} is the following: given an edge-weighted graph \( G = (V, E) \) find a minimum cost subset of the edges that \textit{cover} all the cuts of \( G \); by cover a cut \( S \subseteq V \) we mean that at least one of the edges in \( \delta(S) \) must be chosen. This may appear to be a strange way of looking at the \textsc{MST} problem but this view is useful as we will see later. Another implicit example is the following. Suppose we are given \( n \) rectangles in the plane and the goal is to choose a minimum number of points in the plane such that each input rectangle contains one of the chosen points. This is perhaps more natural to view as a special case of the \textsc{Hitting Set} problem. In principle the set of points that we can choose from is infinite but it can be seen that we can confine our attention to vertices in the arrangement of the given rectangles and it is easy to see that there are only \( O(n^2) \) vertices — however, explicitly computing them may be expensive and one may want to treat the problem as an implicit one for the sake of efficiency. want to think of

Covering problems have the feature that a superset of a feasible solution is also a feasible solution. More abstractly one can cast covering problems as the following. We are given a finite ground set \( V \) (vertices in a graph or sets in a set system) and a family of feasible solutions \( \mathcal{I} \subseteq 2^V \) where \( \mathcal{I} \) is upward closed; by this we mean that if \( A \in \mathcal{I} \) and \( A \subseteq B \) then \( B \in \mathcal{I} \). The goal is to find the smallest cardinality set \( A \) in \( \mathcal{I} \). In the weighted case \( V \) has weights and the goal is to find a minimum weight set in \( \mathcal{I} \). In some case one can also consider more complex non-additive objectives that assign a cost \( c(S) \) for each \( S \in \mathcal{I} \).

## 2.1 Greedy for \textsc{Set Cover} and \textsc{Maximum Coverage}

In this section we consider the unweighted version of \textsc{Set Cover}.

### 2.1.1 Greedy Algorithm

A natural greedy approximation algorithm for these problems is easy to come up with.
CHAPTER 2. COVERING PROBLEMS

<table>
<thead>
<tr>
<th>Greedy Cover($\mathcal{U}, S$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. repeat</td>
</tr>
<tr>
<td>A. pick the set that covers the maximum number of uncovered elements</td>
</tr>
<tr>
<td>B. mark elements in the chosen set as covered</td>
</tr>
<tr>
<td>2. until done</td>
</tr>
</tbody>
</table>

In case of Set Cover, the algorithm Greedy Cover is done when all the elements in set $\mathcal{U}$ have been covered. And in case of Maximum Coverage, the algorithm is done when exactly $k$ subsets have been selected from $S$.

We will prove the following theorem.

**Theorem 2.1.** Greedy Cover is a $1 - (1 - 1/k)^k \geq (1 - \frac{1}{e}) \approx 0.632$ approximation for Maximum Coverage, and a $(\ln n + 1)$ approximation for Set Cover.

The following theorem due to Feige [19] implies that Greedy Cover is essentially the best possible in terms of the approximation ratio that it guarantees.

**Theorem 2.2.** Unless $NP \subseteq DTIME(n^{O(\log \log n)})$, there is no $(1 - o(1)) \ln n$ approximation for Set Cover. Unless $P=NP$, for any fixed $\epsilon > 0$, there is no $(1 - \frac{1}{e} - \epsilon)$ approximation for Maximum Coverage.

Recently the preceding theorem has been strengthened so that the hardness holds under the assumption that $NP \neq P$ [54].

2.1.2 Analysis of Greedy Cover

We proceed towards the proof of Theorem 2.1 by providing analysis of Greedy Cover separately for Set Cover and Maximum Coverage.

**Analysis for Maximum Coverage**

Let OPT denote the value of an optimal solution to the Maximum Coverage problem; this is the maximum number of elements that are covered by $k$ sets in the given set system. Let $x_i$ denote the number of new elements covered by the $i$-th set chosen by Greedy Cover. Also, let $y_i = \sum_{j=1}^{k} x_i$ be the total number of elements covered after $i$ iterations, and $z_i = OPT - y_i$. Note that, according to our notation, $y_0 = 0$ and $y_k$ is the number of elements chosen by Greedy Cover at the end of the algorithm, and $z_0 = OPT$. The key to the analysis is the following simple claim.
Claim 2.1.1. For \(0 \leq i < k\), \(x_{i+1} \geq \frac{z_i}{k}\).

Proof. Let \(F^* \subseteq \mathcal{U}\) be the elements covered by some fixed optimum solution; we have \(|F^*| = \text{OPT}\). Consider iteration \(i + 1\). Greedy Cover selects the subset \(S_j\) whose inclusion covers the maximum number of uncovered elements. Since \(z_i\) is the total number of elements covered up to iteration \(i\), at least \(\text{OPT} - z_i\) elements from \(F^*\) are uncovered. Let the set of uncovered elements from \(F^*\) be \(x_i\). Since \(x_{i+1}\) is the set that covers the maximum number of uncovered elements, the chosen set in iteration \(i + 1\) covers at least \(|x_i|/k\) uncovered elements. Hence, \(x_{i+1} \geq \frac{z_i}{k}\).

Remark 2.1. It is tempting to make a stronger claim that \(x_{i+1} \geq \frac{z_i}{k}\). This is however false, and it is worthwhile to come up with an example.

By definition we have \(y_k = x_1 + x_2 + \ldots + x_k\) is the total number of elements covered by Greedy Cover. To analyze the worst-case we want to make this sum as small as possible given the preceding claim. Heuristically (which one can formalize), one can argue that choosing \(x_{i+1} = \frac{z_i}{k}\) minimizes the sum. Using this one can argue that the sum is at least \((1 - (1 - 1/k)^k))\text{OPT}\). We give a formal argument now.

Claim 2.1.2. For \(i \geq 0\), \(z_i \leq (1 - \frac{1}{k})^i \cdot \text{OPT}\).

Proof. By induction on \(i\). The claim is trivially true for \(i = 0\) since \(z_0 = \text{OPT}\). We assume inductively that \(z_i \leq (1 - \frac{1}{k})^i \cdot \text{OPT}\). Then

\[
\begin{align*}
z_{i+1} &= z_i - x_{i+1} \\
&\leq z_i (1 - \frac{1}{k}) \quad \text{[using Claim 2.1.1]} \\
&\leq (1 - \frac{1}{k})^{i+1} \cdot \text{OPT}.
\end{align*}
\]

The preceding claims yield the following lemma for algorithm Greedy Cover when applied on Maximum Coverage.

Lemma 2.1. Greedy Cover is a \(1 - (1 - 1/k)^k \geq 1 - \frac{1}{e}\) approximation for Maximum Coverage.

Proof. It follows from Claim 2.1.2 that \(z_k \leq (1 - \frac{1}{k})^k \cdot \text{OPT} \leq \frac{\text{OPT}}{e}\). Hence, \(y_k = \text{OPT} - z_k \geq (1 - \frac{1}{e}) \cdot \text{OPT}\).

We note that \((1 - 1/e) \approx 0.632\).
CHAPTER 2. COVERING PROBLEMS

Analysis for Set Cover

Let $k^*$ denote the value of an optimal solution to the Set Cover problem. Then an optimal solution value to the Maximum Coverage problem with the same system and $k = k^*$ would by $n = |U|$ since it is possible to cover all the $n$ elements in set $U$ with $k^*$ sets. From our previous analysis, $z_{k^*} \leq \frac{n}{e}$. Therefore, at most $\frac{n}{e}$ elements would be uncovered after the first $k^*$ steps of Greedy Cover. Similarly, after $2 \cdot k^*$ steps of Greedy Cover, at most $\frac{n}{e} \cdot 2$ elements would remain uncovered. This easy intuition convinces us that Greedy Cover is a $(\ln \frac{n}{e} + 1)$ approximation for the Set Cover problem. A more formal proof is given below.

**Lemma 2.2.** Greedy Cover is a $(\ln \frac{n}{e} + 1)$ approximation for Set Cover.

**Proof.** Since $z_i \leq (1 - \frac{1}{e})i \cdot n$, after $t = \lceil k^* \ln \frac{n}{e} \rceil$ steps,

$$z_t \leq n(1 - 1/e)^{k^* \ln \frac{n}{e}} \leq ne^{-\ln \frac{n}{e}} \leq k^*.$$

Thus, after $t$ steps, at most $k^*$ elements are left to be covered. Since Greedy Cover picks at least one element in each step, it covers all the elements after picking at most $\lceil k^* \ln \frac{n}{e} \rceil + k^* \leq k^*(\ln n + 1)$ sets. ■

A useful special case of Set Cover is when all sets are “small”. Does the approximation bound for Greedy improve? We can prove the following corollary via Lemma 2.2.

**Corollary 2.3.** If each set in the set system has at most $d$ elements, then Greedy Cover is a $(\ln d + 1)$ approximation for Set Cover.

**Proof.** If each set has at most $d$ elements then we have that $k^* \geq \frac{n}{d}$ and hence $\ln \frac{n}{e} \leq \ln d$. Then the claim follows from Lemma 2.2. ■

Theorem 2.1 follows directly from Lemma 2.1 and 2.2.

A near-tight example for Greedy Cover when applied on Set Cover: Let us consider a set $U$ of $n$ elements along with a collection $S$ of $k + 2$ subsets \{R_1, R_2, C_1, C_2, \ldots, C_k\} of U. Let us also assume that $|C_i| = 2^i$ and $|R_1 \cap C_i| = |R_2 \cap C_i| = 2^{i-1}$ $(1 \leq i \leq k)$, as illustrated in Fig. 2.1.

Clearly, the optimal solution consists of only two sets, i.e., $R_1$ and $R_2$. Hence, $OPT = 2$. However, Greedy Cover will pick each of the remaining $k$ sets, namely $C_k, C_{k-1}, \ldots, C_1$. Since $n = 2 \cdot \sum_{i=0}^{k-1} 2^i = 2 \cdot (2^k - 1)$, we get $k = \Omega(\log n)$. One can construct tighter examples with more involved analysis.
Exercise 2.1. Consider the weighted version of the Set Cover problem where a weight function $w: S \rightarrow \mathbb{R}^+$ is given, and we want to select a collection $S'$ of subsets from $S$ such that $\bigcup_{X \in S'} X = U$, and $\sum_{X \in S'} w(X)$ is the minimum. One can generalize the greedy heuristic in the natural fashion where in each iteration the algorithm picks the set that maximizes the ratio of the number of elements to its weight. Adapt the unweighted analysis to prove that the greedy algorithm yields an $O(\ln n)$ approximation for the weighted version (you can be sloppy with the constant in front of $\ln n$).

2.1.3 Dominating Set

A dominating set in a graph $G = (V, E)$ is a set $S \subseteq V$ such that for each $u \in V$, either $u \in S$, or some neighbor $v$ of $u$ is in $S$. In other words $S$ covers/dominates all the nodes in $V$. In the Dominating Set problem, the input is a graph $G$ and the goal is to find a smallest sized dominating set in $G$.

Exercise 2.2. 1. Show that Dominating Set is a special case of Set Cover.

2. What is the greedy heuristic when applied to Dominating Set. Prove that it yields an $(\ln (\Delta + 1) + 1)$ approximation where $\Delta$ is the maximum degree in the graph.

3. Show that Set Cover can be reduced in an approximation preserving fashion to Dominating Set. More formally, show that if Dominating Set has an $\alpha(n)$-approximation where $n$ is the number of vertices in the given instance then Set Cover has an $(1 - o(1))\alpha(n)$-approximation.
2.2 Vertex Cover

We have already seen that the Vertex Cover problem is a special case of the Set Cover problem. The Greedy algorithm when specialized to Vertex Cover picks a highest degree vertex, removes it and the covered edges from the graph, and recurses in the remaining graph. It follows that the Greedy algorithm gives an $O((\ln \Delta + 1)$ approximation for the unweighted versions of the Vertex Cover problem. One can wonder whether the Greedy algorithm has a better worst-case for Vertex Cover than the analysis suggests. Unfortunately the answer is negative and there are examples where the algorithm outputs a solution with $\Omega(\log n \cdot OPT)$ vertices.

We sketch the construction. Consider a bipartite graph $G = (U, V, E)$ where $U = \{u_1, u_2, \ldots, u_h\}$. $V$ is partitioned into $S_1, S_2, \ldots, S_h$ where $S_i$ has $\lfloor h/i \rfloor$ vertices. Each vertex $v$ in $S_i$ is connected to exactly $i$ distinct vertices of $U$; thus, any vertex $u_i$ is incident to at most one edge from $S_i$. It can be seen that the degree of each vertex $u_i \in U$ is roughly $h$. Clearly $U$ is a vertex cover of $G$ since the graph is bipartite. Convince yourself that the Greedy algorithm will pick all of $V$ starting with the lone vertex in $S_h$ (one may need to break ties to make this happen but the example can be easily perturbed to make this unnecessary). We have $n = \Theta(h \log h)$ and $OPT \leq h$ and Greedy outputs a solution of size $\Omega(h \log h)$.

2.2.1 A 2-approximation for Vertex Cover

There is a very simple 2-approximation algorithm for the Cardinality Vertex Cover problem.

<table>
<thead>
<tr>
<th>Matching-VC($G$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $S \leftarrow \emptyset$</td>
</tr>
<tr>
<td>2. Compute a maximal matching $M$ in $G$</td>
</tr>
<tr>
<td>3. for each edge $(u, v) \in M$ do</td>
</tr>
<tr>
<td>&gt; A. add both $u$ and $v$ to $S$</td>
</tr>
<tr>
<td>4. Output $S$</td>
</tr>
</tbody>
</table>

**Theorem 2.4.** Matching-VC is a 2-approximation algorithm.

The proof of Theorem 2.4 follows from two simple claims.
Claim 2.2.1. Let $\text{OPT}$ be the size of the vertex cover in an optimal solution. Then $\text{OPT} \geq |M|$.

Proof. Any vertex cover must contain at least one end point of each edge in $M$ since no two edges in $M$ intersect. Hence $\text{OPT} \geq |M|$.

Claim 2.2.2. Let $S(M) = \{u, v | (u, v) \in M\}$. Then $S(M)$ is a vertex cover.

Proof. If $S(M)$ is not a vertex cover, then there must be an edge $e \in E$ such that neither of its endpoints are in $M$. But then $e$ can be included in $M$, which contradicts the maximality of $M$.

We now finish the proof of Theorem 2.4. Since $S(M)$ is a vertex cover, Claim 2.2.1 implies that $|S(M)| = 2 \cdot |M| \leq 2 \cdot \text{OPT}$.

**Weighted Vertex Cover:** The matching based heuristic does not generalize in a straightforward fashion to the weighted case but 2-approximation algorithms for the Weighted Vertex Cover problem can be designed based on LP rounding.

2.2.2 Set Cover with small frequencies

*Set Cover* is an instance of *Set Cover* where each element in $\mathcal{U}$ is in at most two sets (in fact, each element was in exactly two sets). This special case of the *Set Cover* problem admits a 2-approximation algorithm. What would be the case if every element is contained in at most three sets? More generally, given an instance of *Set Cover*, for each $e \in \mathcal{U}$, let $f(e)$ denote the number of sets containing $e$. Let $f = \max_e f(e)$, which we call the maximum frequency.

Exercise 2.3. Give an $f$-approximation for *Set Cover*, where $f$ is the maximum frequency of an element. *Hint:* Follow the approach used for *Vertex Cover*.

2.3 Vertex Cover via LP

Let $G = (V, E)$ be an undirected graph with arc weights $w : V \rightarrow R^+$. We can formulate Vertex Cover as an integer linear programming problem as follows. For each vertex $v$ we have a variable $x_v$. We interpret the variable as follows: if $x_v = 1$ if $v$ is chosen to be included in a vertex cover, otherwise $x_v = 0$. With this interpretation we can easily see that the minimum weight vertex cover can be formulated as the following integer linear program.
CHAPTER 2. COVERING PROBLEMS

\[
\min \sum_{v \in V} w_v x_v \\
\text{subject to} \\
x_u + x_v \geq 1 \quad \forall e = (u, v) \in E \\
x_v \in \{0, 1\} \quad \forall v \in V 
\]

However, solving the preceding integer linear program is NP-Hard since it would solve Vertex Cover exactly. Therefore we use Linear Programming (LP) to approximate the optimal solution, OPT(I), for the integer program. First, we can relax the constraint \( x_v \in \{0, 1\} \) to \( x_v \in [0, 1] \). It can be further simplified to \( x_v \geq 0, \forall v \in V \).

Thus, a linear programming formulation for Vertex Cover is:

\[
\min \sum_{v \in V} w_v x_v \\
\text{subject to} \\
x_u + x_v \geq 1 \quad \forall e = (u, v) \in E \\
x_v \geq 0 
\]

We now use the following algorithm:

\[
\text{**Vertex Cover via LP**}
\]

1. Solve LP to obtain an optimal fractional solution \( x^* \)
2. Let \( S = \{v \mid x^*_v \geq \frac{1}{2}\} \)
3. Output \( S \)

**Claim 2.3.1.** \( S \) is a vertex cover.

**Proof.** Consider any edge, \( e = (u, v) \). By feasibility of \( x^* \), \( x^*_u + x^*_v \geq 1 \), and thus \( x^*_u \geq \frac{1}{2} \) or \( x^*_v \geq \frac{1}{2} \). Therefore, at least one of \( u \) and \( v \) will be in \( S \). \( \blacksquare \)

**Claim 2.3.2.** \( w(S) \leq 2 \text{OPT}_{LP}(I) \).

**Proof.** \( \text{OPT}_{LP}(I) = \sum_v w_v x^*_v \geq \frac{1}{2} \sum_{v \in S} w_v = \frac{1}{2} w(S) \). \( \blacksquare \)

Therefore, \( \text{OPT}_{LP}(I) \geq \frac{\text{OPT}(I)}{2} \) for all instances \( I \).

**Remark 2.2.** For minimization problems: \( \text{OPT}_{LP}(I) \leq \text{OPT}(I) \), where \( \text{OPT}_{LP}(I) \) is the optimal solution found by LP; for maximization problems, \( \text{OPT}_{LP}(I) \geq \text{OPT}(I) \).
Integrality Gap: We introduce the notion of integrality gap to show the best approximation guarantee we can obtain if we only use the LP values as a lower bound.

Definition 2.5. For a minimization problem \( \Pi \), the integrality gap for a linear programming relaxation/formulation \( LP \) for \( \Pi \) is \( \sup_{I \in \mathcal{I}} \frac{\text{OPT}(I)}{\text{OPT}_{LP}(I)} \).

That is, the integrality gap is the worst case ratio, over all instances \( I \) of \( \Pi \), of the integral optimal value and the fractional optimal value. Note that different linear programming formulations for the same problem may have different integrality gaps.

Claims 2.3.1 and 2.3.2 show that the integrality gap of the Vertex Cover LP formulation above is at most 2.

Question: Is this bound tight for the Vertex Cover LP?

Consider the following example: Take a complete graph, \( K_n \), with \( n \) vertices, and each vertex has \( w_v = 1 \). It is clear that we have to choose \( n - 1 \) vertices to cover all the edges. Thus, \( \text{OPT}(K_n) = n - 1 \). However, \( x_v = \frac{1}{2} \) for each \( v \) is a feasible solution to the LP, which has a total weight of \( \frac{n}{2} \). So gap is \( 2 - \frac{1}{n} \), which tends to 2 as \( n \to \infty \). One can also prove that the integrality gap is essentially 2 even in a class of sparse graphs.

Exercise 2.4. The vertex cover problem can be solved optimally in polynomial time in bipartite graphs. In fact the LP is integral. Prove this via the maxflow-mincut theorem and the integrality of flows when capacities are integral.

Other Results on Vertex Cover

1. The current best approximation ratio for Vertex Cover is \( 2 - \Theta\left(\frac{1}{\sqrt{\log n}}\right) \) [41].

2. It is known that unless \( P = NP \) there is \( \alpha \)-approximation for Vertex Cover for \( \alpha < 1.36 \) [18]. Under a stronger hypothesis called the Unique Games Conjecture it is known that there is no \( 2 - \epsilon \) approximation for any fixed \( \epsilon > 0 \) [44].

3. There is a polynomial time approximation scheme (PTAS), that is a \( (1 + \epsilon) \)-approximation for any fixed \( \epsilon > 0 \), for planar graphs. This follows from a general approach due to Baker [5]. The theorem extends to more general classes of graphs.
2.4 Set Cover via LP

The input to the Set Cover problem consists of a finite set $U = \{1, 2, ..., n\}$, and $m$ subsets of $U$, $S_1, S_2, ..., S_m$. Each set $S_j$ has a non-negative weight $w_j$ and the goal is to find the minimum weight collection of sets which cover all elements in $U$ (in other words their union is $U$).

A linear programming relaxation for Set Cover is:

$$\begin{align*}
\text{min} & \quad \sum_j w_j x_j \\
\text{subject to} & \quad \sum_{j : i \in S_j} x_j \geq 1 \quad \forall i \in \{1, 2, ..., n\} \\
& \quad x_j \geq 0 \quad 1 \leq j \leq m
\end{align*}$$

And its dual is:

$$\begin{align*}
\text{max} & \quad \sum_{i=1}^{n} y_i \\
\text{subject to} & \quad \sum_{i \in S_j} y_i \leq w_j \quad \forall j \in \{1, 2, ..., m\} \\
& \quad y_i \geq 0 \quad \forall i \in 1, 2, ..., n
\end{align*}$$

We give several algorithms for Set Cover based on this primal/dual pair LPs.

2.4.1 Deterministic Rounding

<table>
<thead>
<tr>
<th>Set Cover via LP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Solve LP to obtain an optimal solution $x^*$, which contains fractional numbers.</td>
</tr>
<tr>
<td>2. Let $P = {i \mid x_i^* &gt; 0}$</td>
</tr>
<tr>
<td>3. Output ${S_j \mid j \in P}$</td>
</tr>
</tbody>
</table>

Note that the above algorithm, even when specialized to Vertex Cover is different from the one we saw earlier. It includes all sets which have a strictly positive value in an optimum solution to the LP.
Let \( x^\ast \) be an optimal solution to the primal LP, \( y^\ast \) be an optimum solution to the dual, and let \( P = \{ j \mid x_j^\ast > 0 \} \). First, note that by strong duality, \( \sum_j w_j x_j^\ast = \sum_i y_i^\ast \). Second, by complementary slackness if \( x_j^\ast > 0 \) then the corresponding dual constraint is tight, that is \( \sum_{i \in S_j} y_i^\ast = w_j \).

**Claim 2.4.1.** The output of the algorithm is a feasible set cover for the given instance.

*Proof.* Exercise. ■

**Claim 2.4.2.** \( \sum_{j \in P} w_j \leq f \sum_j w_j x_j^\ast = \text{OPT}_{LP} \).

*Proof.*

\[
\sum_{j \in P} w_j = \sum_{j : x_j^\ast > 0} w_j = \sum_{j : x_j^\ast > 0} \left( \sum_{i \in S_j} y_i^\ast \right) = \sum_i y_i^\ast \left( \sum_{j : x_j^\ast > 0} 1 \right) \leq f \sum_i y_i^\ast = f \text{OPT}_{LP}(I). 
\]

Notice that the the second equality is due to complementary slackness conditions (if \( x_j^\ast > 0 \), the corresponding dual constraint is tight), the penultimate inequality uses the definition of \( f \), and the last inequality follows from weak duality (a feasible solution for the dual problem is a lower bound on the optimal primal solution).

Therefore we have that the algorithm outputs a cover of weight at most \( f \text{OPT}_{LP} \). We note that \( f \) can be as large as \( n \) in which case the bound given by the algorithm is quite weak. In fact, it is not hard to construct examples that demonstrate the tightness of the analysis.

**Remark 2.3.** The analysis crucially uses the fact that \( x^\ast \) is an optimal solution. On the other hand the algorithm for Vertex Cover is more robust and works with any feasible solution \( x \). It is easy to generalize the earlier rounding for Vertex Cover to obtain an \( f \)-approximation. The point of the above rounding is to illustrate the utility of complementary slackness.

### 2.4.2 Randomized Rounding

Now we describe a different rounding that yields an approximation bound that does not depend on \( f \).
CHAPTER 2. COVERING PROBLEMS

<table>
<thead>
<tr>
<th>Set Cover via Randomized Rounding</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $A = \emptyset$, and let $x^*$ be an optimal solution to the LP</td>
</tr>
<tr>
<td>2. for $k = 1$ to $2\ln n$ do</td>
</tr>
<tr>
<td>A. pick each $S_j$ independently with probability $x_j^*$</td>
</tr>
<tr>
<td>B. if $S_j$ is picked, $A = A \cup {j}$</td>
</tr>
<tr>
<td>3. Output the sets with indices in $A$</td>
</tr>
</tbody>
</table>

Claim 2.4.3. $P[i$ is not covered in an iteration$] = \prod_{j : i \in S_j} (1 - x_j^*) \leq \frac{1}{e}$.

Intuition: We know that $\sum_{j : i \in S_j} x_j^* \geq 1$. Subject to this constraint, if we want to minimize the probability that element $i$ is covered, one can see that the minimum is achieved with $x_j^* = 1/\ell$ for each set $S_j$ that covers $i$; here $\ell$ is the number of sets that cover $i$. Then the probability is $(1 - 1/\ell)^\ell$.

Proof. We use the inequality $(1 - x) \leq e^{-x}$ for all $x \in [0, 1]$.

$$P[i$ is not covered in an iteration$] = \prod_{j : i \in S_j} (1 - x_j^*) \leq \prod_{j : i \in S_j} e^{-x_j^*} \leq e^{-\sum_{j : i \in S_j} x_j^*} \leq \frac{1}{e}. $$

$$\blacksquare$$

We then obtain the following corollaries:

Corollary 2.6. $P[i$ is not covered at the end of the algorithm$] \leq e^{-2\log n} \leq \frac{1}{n^2}$.

Corollary 2.7. $P[all elements are covered, after the algorithm stops] \geq 1 - \frac{1}{n}$.

Proof. Via the union bound. The probability that $i$ is not covered is at most $1/n^2$, hence the probability that there is some $i$ that is not covered is at most $n \cdot 1/n^2 \leq 1/n$.

Now we bound the expected cost of the algorithm. Let $C_t =$ cost of sets picked in iteration $t$, then $E[C_t] = \sum_{j=1}^{n} w_j x_j^*$, where $E[X]$ denotes the expectation of a random variable $X$. Then, let $C = \sum_{t=1}^{2\ln n} C_t$; we have $E[C] = \sum_{t=1}^{2\ln n} E[C_t] \leq 2\ln n \OPT_{LP}$. By Markov’s inequality, $P[C > 2 E[C]] \leq \frac{1}{2}$, hence $P[C \leq 4 \ln n \OPT_{LP}] \geq \frac{1}{2}$. Therefore, $P[C \leq 4 \ln n \OPT_{LP}$ and all items are covered$] \geq \frac{1}{2} - \frac{1}{n}$. Thus, the randomized rounding algorithm, with probability close to $1/2$
succeeds in giving a feasible solution of cost $O(\log n) \OPT_{LP}$. Note that we can check whether the solution satisfies the desired properties (feasibility and cost) and repeat the algorithm if it does not.

1. We can check if solution after rounding satisfies the desired properties, such as all elements are covered, or cost at most $2e \log n \OPT_{LP}$. If not, repeat rounding. Expected number of iterations to succeed is a constant.

2. We can also use Chernoff bounds (large deviation bounds) to show that a single rounding succeeds with high probability (probability at least $1 - \frac{1}{\text{poly}(n)}$).

3. The algorithm can be derandomized. Derandomization is a technique of removing randomness or using as little randomness as possible. There are many derandomization techniques, such as the method of conditional expectation, discrepancy theory, and expander graphs.

4. After a few rounds, select the cheapest set that covers each uncovered element. This has low expected cost. This algorithm ensures feasibility but guarantees cost only in the expected sense. We will see a variant on the homework.

**Randomized Rounding with Alteration:** In the preceding analysis we had to worry about the probability of covering all the elements and the expected cost of the solution. Here we illustrate a simple yet powerful technique of alteration in randomized algorithms and analysis. Let $d$ be the maximum set size.

**Set Cover: Randomized Rounding with Alteration**

1. $A = \emptyset$, and let $x^*$ be an optimal solution to the LP
2. Add to $A$ each $S_j$ independently with probability $\min\{1,\ln d \cdot x_j^*\}$
3. Let $U'$ be the elements uncovered by the chosen sets in $A$
4. For each uncovered element $i \in U'$ do
   A. Add to $A$ the cheapest set that covers $i$
5. Output the sets with indices in $A$

The algorithm has two phases. A randomized phase and a fixing/altering phase. In the second phase we apply a naive algorithm that may have a high cost in the worst case but we will bound its expected cost appropriately. The
algorithm deterministically guarantees that all elements will be covered, and hence we only need to focus on the expected cost of the chosen sets. Let $C_1$ be the random cost of the sets chosen in the first phase and let $C_2$ be the random cost of the sets chosen in the second phase. It is easy to see that $E[C_1] = \ln d \sum_j w_j x_j^* = \ln d \OPT_{LP}$. Let $\mathcal{E}_i$ be the event that element $i$ is not covered after the first randomized phase.

**Exercise 2.5.** $P[\mathcal{E}_i] \leq e^{-\ln d} \leq 1/d$.

The worst case second phase cost can be upper bounded via the next lemma.

**Lemma 2.3.** Let $\beta_i$ be the cost of the cheapest set covering $i$. Then $\sum_i \beta_i \leq d \OPT_{LP}$.

*Proof.* Consider an element $i$. We have the constraint that $\sum_{j:i \in S_j} x_j^* \geq 1$. Since each set covering $i$ has cost at least $\beta_i$, we have $\sum_{j:i \in S_j} c_j x_j^* \geq \beta_i \sum_{j:i \in S_j} x_j^* \geq \beta_i$. Thus,

$$\sum_i \beta_i \leq \sum_i \sum_{j:i \in S_j} c_j x_j^* \leq \sum_j c_j x_j^* |S_j| \leq d \sum_j c_j x_j^* = d \OPT_{LP}.$$

Proof.

Now we bound the expected second phase cost.

**Lemma 2.4.** $E[C_2] \leq \OPT_{LP}$.

*Proof.* We pay for a set to cover element $i$ in the second phase only if it is not covered in the first phase. Hence $C_2 = \sum_i \mathcal{E}_i \beta_i$. Note that the events $\mathcal{E}_i$ for different elements $i$ are not necessarily independent, however, we can apply linearity of expectation.

$$E[C_2] = \sum_i E[\mathcal{E}_i] \beta_i = \sum_i P[\mathcal{E}_i] \beta_i \leq 1/d \sum_i \beta_i \leq \OPT_{LP}.$$

Proof.

Combining the expected costs of the two phases we obtain the following theorem.

**Theorem 2.8.** Randomized rounding with alteration outputs a feasible solution of expected cost $(1 + \ln d) \OPT_{LP}$.

Note that the simplicity of the algorithm and tightness of the bound.

**Remark 2.4.** If $d = 2$ the Set Cover problem becomes the Edge Cover problem in a graph which is the following. Given an edge-weighted graph $G = (V, E)$, find the minimum weight subset of edges such that each vertex is covered. Edge Cover admits a polynomial-time algorithm via a reduction to the minimum-cost matching problem in a general graph. However $d = 3$ for Set Cover is NP-Hard via a reduction from 3-D Matching.
2.4.3 Dual-fitting

In this section, we introduce the technique of dual-fitting for the analysis of approximation algorithms. At a high-level the approach is the following:

1. Consider an algorithm that one wants to analyze.
2. Construct a feasible solution to the dual LP based on the structure of the algorithm.
3. Show that the cost of the solution returned by the algorithm can be bounded in terms of the value of the dual solution.

Note that the algorithm itself need not be LP based. Here, we use Set Cover as an example. See the previous section for the primal and dual LP formulations for Set Cover.

We can interpret the dual as follows: Think of $y_i$ as how much element $i$ is willing to pay to be covered; the dual maximizes the total payment, subject to the constraint that for each set, the total payment of elements in that set is at most the cost of the set.

We rewrite the Greedy algorithm for Weighted Set Cover.

```
Greedy Set Cover
1. Covered = Ø
2. A = Ø;
3. While Covered ≠ U do
   A. j ← arg min$_k$ ($\frac{w_k}{|S_k \cap Uncovered|}$);
   B. Covered = Covered ∪ $S_j$;
   C. A = A ∪ {j}.
4. end while;
5. Output sets in A as cover
```

Let $H_k = 1 + 1/2 + \ldots + 1/k$ be the $k$th Harmonic number. It is well known that $H_k \leq 1 + \ln k$.

**Theorem 2.9.** Greedy Set Cover picks a solution of cost $\leq H_d \cdot \text{OPT}_{LP}$, where $d$ is the maximum set size, i.e., $d = \max_j |S_j|$. 

To prove this, we augment the algorithm to keep track of some additional information.

**Augmented Greedy Algorithm of Weighted Set Cover**

1. \( \text{Covered} = \emptyset \)

2. While \( \text{Covered} \neq U \) do
   
   \( j \leftarrow \arg \min_k \left( \frac{w_k}{|S_k \cap \text{Uncovered}|} \right) \)
   
   \( \text{if } i \text{ is uncovered and } i \in S_j, \text{ set } p_i = \frac{w_j}{|S_j \cap \text{Uncovered}|}; \)
   
   \( \text{Covered} = \text{Covered} \cup S_j \)
   
   \( A = A \cup \{ j \} \)

3. Output sets in \( A \) as cover

It is easy to see that the algorithm outputs a feasible cover.

**Claim 2.4.4.** \( \sum_{j \in A} w_j = \sum_i p_i \).

**Proof.** Consider when \( j \) is added to \( A \). Let \( S'_j \subseteq S_j \) be the elements that are uncovered before \( j \) is added. For each \( i \in S'_j \) the algorithm sets \( p_i = w_j/|S'_j| \).

Hence, \( \sum_{i \in S'_j} p_i = w_j \). Moreover, it is easy to see that the sets \( S'_j, j \in A \) are disjoint and together partition \( U \). Therefore,

\[
\sum_{j \in A} w_j = \sum_{j \in A} \sum_{i \in S'_j} p_i = \sum_{i \in U} p_i.
\]

For each \( i \), let \( y'_i = \frac{1}{p_i} p_i \).

**Claim 2.4.5.** \( y' \) is a feasible solution for the dual LP.

Suppose the claim is true, then the cost of Greedy Set Cover’s solution = \( \sum_i p_i = H_d \sum_i y'_i \leq H_d \text{OPT}_{LP} \). The last step is because any feasible solution for the dual problem is a lower bound on the value of the primal LP (weak duality).

Now, we prove the claim. Let \( S_j \) be an arbitrary set, and let \( |S_j| = t \leq d \). Let \( S_j = \{ i_1, i_2, \ldots, i_t \} \), where we the elements are ordered such that \( i_1 \) is covered by Greedy no-later than \( i_2 \), and \( i_2 \) is covered no later than \( i_3 \) and so on.

**Claim 2.4.6.** For \( 1 \leq h \leq t \), \( p_{i_h} \leq \frac{w_j}{t-h+1} \).
Proof. Let $S_j'$ be the set that covers $i_h$ in Greedy. When Greedy picked $S_j'$ the elements $i_h, i_{h+1}, \ldots, i_t$ from $S_j$ were uncovered and hence Greedy could have picked $S_j$ as well. This implies that the density of $S_j'$ when it was picked was no more than $\frac{w_j}{t-h+1}$. Therefore $p_{ih}$ which is set to the density of $S_j'$ is at most $\frac{w_j}{t-h+1}$. Therefore $\frac{w_j}{t-h+1}$. Therefore

\[\text{From the above claim, we have} \]
\[\sum_{1 \leq h \leq t} p_{ih} \leq \sum_{1 \leq h \leq t} \frac{w_j}{t-h+1} = w_j H_t \leq w_j H_d.\]

Thus, the setting of $y'_i$ to be $p_i$ scaled down by a factor of $H_d$ gives a feasible solution.

2.4.4 Greedy for implicit instances of Set Cover

Set Cover and the Greedy heuristic are quite useful in applications because many instances are implicit, nevertheless, the algorithm and the analysis applies. That is, the universe $U$ of elements and the collection $S$ of subsets of $U$ need not be restricted to be finite or explicitly enumerated in the Set Cover problem. For instance, a problem could require covering a finite set of points in the plane using disks of unit radius. There is an infinite set of such disks, but the Greedy approximation algorithm can still be applied. For such implicit instances, the Greedy algorithm can be used if we have access to an oracle, which, at each iteration, selects a set having the optimal density. However, an oracle may not always be capable of selecting an optimal set. In some cases it may have to make the selections approximately. We call an oracle an $\alpha$-approximate oracle for some $\alpha \geq 1$ if, at each iteration, it selects a set $S$ such that $\frac{w(S)}{S} \leq \alpha \min_A \frac{A}{w(A)}$.

Exercise 2.6. Prove that the approximation guarantee of Greedy with an $\alpha$-approximate oracle would be $\alpha (\ln n + 1)$ for Set Cover, and $(1 - \frac{1}{e\alpha})$ for Maximum Coverage.

We will see several examples of implicit use of the greedy analysis in the course.

2.5 Submodularity

Set Cover turns out to be a special case of a more general problem called Submodular Set Cover. The Greedy algorithm and analysis applies in this more generality. Submodularity is a fundamental notion with many applications in
CHAPTER 2. COVERING PROBLEMS

combinatorial optimization and elsewhere. Here we take the opportunity to provide some definitions and a few results.

Definition 2.10. Given a finite set $E$, a real-valued set function $f : 2^E \to \mathbb{R}$ is submodular iff

$$f(A) + f(B) \geq f(A \cup B) + f(A \cap B) \quad \forall A, B \subseteq E.$$ 

Alternatively, $f$ is a submodular function iff

$$f(A \cup \{i\}) - f(A) \geq f(B \cup \{i\}) - f(B) \quad \forall A \subset B, i \in E \setminus B.$$ 

The second characterization shows that submodularity is based on decreasing marginal utility property in the discrete setting. Adding element $i$ to a set $A$ will help at least as much as adding it to a (larger) set $B \supset A$. It is common to use $A + i$ to denote $A \cup \{i\}$ and $A - i$ to denote $A \setminus \{i\}$.

Exercise 2.7. Prove that the two characterizations of submodular functions are equivalent.

Many application of submodular functions are when $f$ is a non-negative function though there are several important applications when $f$ can be negative. A submodular function $f(\cdot)$ is monotone if $f(A + i) \geq f(A)$ for all $i \in E$ and $A \subseteq E$. Typically one assumes that $f$ is normalized by which we mean that $f(\emptyset) = 0$; this can always be done by shifting the function by $f(\emptyset)$. $f$ is symmetric if $f(A) = f(E \setminus A)$ for all $A \subseteq E$. Submodular set functions arise in a large number of fields including combinatorial optimization, probability, and geometry. Examples include rank function of a matroid, the sizes of cutsets in a directed or undirected graph, the probability that a subset of events do not occur simultaneously, entropy of random variables, etc. In the following we show that the Set Cover and Maximum Coverage problems can be easily formulated in terms of submodular set functions.

Exercise 2.8. Let $\mathcal{U}$ be a set and let $S = \{S_1, S_2, \ldots, S_m\}$ be a finite collection of subsets of $\mathcal{U}$. Let $N = \{1, 2, \ldots, m\}$, and define $f : 2^N \to \mathbb{R}$ as: $f(A) = |\cup_{i \in A} S_i|$ for $A \subseteq E$. Show that $f$ is a monotone non-negative submodular set function.

Exercise 2.9. Let $G = (V, E)$ be a directed graph and let $f : 2^V \to \mathbb{R}$ where $f(S) = |\delta^+(S)|$ is the number of arcs leaving $S$. Prove that $f$ is submodular. Is the function monotone?

2.5.1 Submodular Set Cover

When formulated in terms of submodular set functions, the Set Cover problem is the following. Given a monotone submodular function $f$ (whose value would
be computed by an oracle) on \( N = \{1, 2, \ldots, m\} \), find the smallest set \( S \subseteq N \) such that \( f(S) = f(N) \). Our previous greedy approximation can be applied to this formulation as follows.

<table>
<thead>
<tr>
<th>Greedy Submodular ((f, N))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( S \leftarrow \emptyset )</td>
</tr>
<tr>
<td>2. <strong>While</strong> ( f(S) \neq f(N) ) <strong>do</strong></td>
</tr>
<tr>
<td>A. find ( i ) to maximize ( f(S + i) - f(S) )</td>
</tr>
<tr>
<td>B. ( S \leftarrow S \cup {i} )</td>
</tr>
<tr>
<td>3. Output ( S )</td>
</tr>
</tbody>
</table>

Not so easy exercise.

**Exercise 2.10.**
1. Prove that the greedy algorithm is a \( 1 + \ln(f(N)) \) approximation for **Submodular Set Cover**.

2. Prove that the greedy algorithm is a \( 1 + \ln(\max_i f(i)) \) approximation for **Submodular Set Cover**.

The above results were first obtained by Wolsey [67].

### 2.5.2 Submodular Maximum Coverage

By formulating the **Maximum Coverage** problem in terms of submodular functions, we seek to maximize \( f(S) \) such that \( |S| \leq k \). We can apply algorithm **Greedy Submodular** for this problem by changing the condition in line 2 to be: **while** \( |S| \leq k \).

**Exercise 2.11.** Prove that greedy gives a \((1 - 1/e)\)-approximation for **Submodular Maximum Coverage** problem when \( f \) is monotone and non-negative. *Hint:* Generalize the main claim that we used for **Maximum Coverage**.

The above and many related results were shown in the influential papers of Fisher, Nemhauser and Wolsey [22, 56].

### 2.6 Covering Integer Programs (CIPs)

There are several extensions of **Set Cover** that are interesting and useful. **Submodular Set Cover** is a very general problem while there are intermediate
problems of interest such as Set Mmulticover. We refer to the reader to the relevant chapters in the two reference books. Here we refer to a general problem called Covering Integer Programs (CIPs for short). The goal is to solve the following integer program where $A \in \mathbb{R}^{n \times m}$ is a non-negative matrix. We can assume without loss of generality that $w$ and $b$ are also non-negative.

$$\min \sum_{j=1}^{n} w_j x_j$$

subject to

$$Ax \geq b$$

$$x_j \leq d_j \quad 1 \leq j \leq m$$

$$x_j \geq 0 \quad 1 \leq j \leq m$$

$$x_j \in \mathbb{Z} \quad 1 \leq j \leq m$$

$Ax \geq b$ model covering constraints and $x_j \leq d_j$ models multiplicity constraints. Note that Set Cover is a special case where $A$ is simply the incidence matrix of the sets and elements (the columns correspond to sets and the rows to elements) and $d_j = 1$ for all $j$. What are CIPs modeling? It is a generalization of Set Cover. To see this, assume, without loss of generality, that $A$, $b$ are integer matrices. For each element corresponding to row $i$ the quantity $b_i$ corresponds to the requirement of how many times $i$ needs to be covered. $A_{ij}$ corresponds to the number of times set $S_j$ covers element $i$. $d_j$ is an upper bound on the number of copies of set $S_j$ that are allowed to be picked.

**Exercise 2.12.** Prove that CIPs are a special case of Submodular Set Cover.

One can apply the Greedy algorithm to the above problem and the standard analysis shows that the approximation ratio obtained is $O(\log B)$ where $B = \sum b_i$ (assuming that they are integers). Even though this is reasonable we would prefer a strongly polynomial bound. In fact there are instances where $B$ is exponential in $n$ and the worst-case approximation ratio can be poor. The natural LP relaxation of the above integer program has a large integrality gap in constrast to the case of Set Cover. One needs to strengthen the LP relaxation via what are known as knapsack cover inequalities. We refer the reader to the paper of Kolliopoulos and Young [45] and recent one by Chekuri and Quanrud [13] for more on this problem.
Chapter 3

Knapsack

In this lecture we explore the Knapsack problem. This problem provides a good basis for learning some important procedures used for approximation algorithms that give better solutions at the cost of higher running time.

3.1 The Knapsack Problem

In the Knapsack problem we are given a number (knapsack capacity) \( B \geq 0 \), and a set \( N \) of \( n \) items; each item \( i \) has a given size \( s_i \geq 0 \) and a profit \( p_i \geq 0 \). We will assume that all the input numbers are integers (or more generally rationals). Given a subset of the items \( A \subseteq N \), we define two functions, \( s(A) = \sum_{i \in A} s_i \) and \( p(A) = \sum_{i \in A} p_i \), representing the total size and profit of the group of items respectively. The goal is to choose a subset of the items, \( A \), such that \( s(A) \leq B \) and \( p(A) \) is maximized. We will assume, without loss of generality, that \( s_i \leq B \) for all \( i \); we can discard items that do not satisfy this constraint.

It is not difficult to see that if all the profits are identical (say 1), the natural greedy algorithm that inserts items in the order of non-increasing sizes yields. Assuming the profits and sizes are integral, we can still find an optimal solution to the problem using dynamic programming in either \( O(nB) \) or \( O(nP) \) time, where \( P = \sum_{i=1}^{n} p_i \). These are standard exercises. While these algorithms appear to run in polynomial time, it should be noted that \( B \) and \( P \) can be exponential in the size of the input written in binary. We call such algorithms pseudo-polynomial time algorithms as their running times are polynomial when numbers in the input are given in unary. Knapsack is a classical NP-Hard problem and these results (and the proof of NP-Hardness) show that the hardness manifests itself when the numbers are large (exponential in \( n \) which means that the number of bits in the size or profit are polynomial in \( n \)).
3.1.1 A Greedy Algorithm

Consider the following greedy algorithm for the Knapsack problem which we will refer to as GreedyKnapsack. We sort all the items by the ratio of their profits to their sizes so that \( \frac{p_1}{s_1} \geq \frac{p_2}{s_2} \geq \cdots \geq \frac{p_n}{s_n} \). Afterward, we greedily take items in this order as long as adding an item to our collection does not exceed the capacity of the knapsack. It turns out that this algorithm can be arbitrarily bad. Suppose we only have two items in \( N \). Let \( s_1 = 1, p_1 = 2, s_2 = B, \) and \( p_2 = B \). GreedyKnapsack will take only item 1, but taking only item 2 would be a better solution and the ratio of the profits in the two cases is \( \frac{2}{B} \) which can be made arbitrarily small. As it turns out, we can easily modify this algorithm to provide a 2-approximation by simply taking the best of GreedyKnapsack’s solution or the most profitable item. We will call this new algorithm ModifiedGreedy.

**Theorem 3.1.** ModifiedGreedy has an approximation ratio of 1/2 for the Knapsack problem.

**Proof.** Let \( k \) be the index of the first item that is not accepted by GreedyKnapsack. Consider the following claim:

**Claim 3.1.1.** \( p_1 + p_2 + \cdots + p_k \geq \text{OPT} \). In fact, \( p_1 + p_2 + \cdots + \alpha p_k \geq \text{OPT} \) where \( \alpha = \frac{B - (s_1 + s_2 + \cdots + s_{k-1})}{s_k} \) is the fraction of item \( k \) that can still fit in the knapsack after packing the first \( k - 1 \) items.

The proof of Theorem 3.1 follows immediately from the claim. In particular, either \( p_1 + p_2 + \cdots + p_{k-1} \) or \( p_k \) must be at least \( \text{OPT}/2 \). We now only have to prove Claim 3.1.1. We give an LP relaxation of the Knapsack problem as follows: Here, \( x_i \in [0,1] \) denotes the fraction of item \( i \) packed in the knapsack.

\[
\begin{align*}
\text{maximize} \quad & \sum_{i=1}^{n} p_i x_i \\
\text{subject to} \quad & \sum_{i=1}^{n} s_i x_i \leq B \\
& x_i \leq 1 \text{ for all } i \in \{1 \ldots n\} \\
& x_i \geq 0 \text{ for all } i \in \{1 \ldots n\}
\end{align*}
\]

Let OPT\(_{LP}\) be the optimal value of the objective function in this linear programming instance. Any solution to Knapsack is a feasible solution to the LP and both problems share the same objective function, so OPT\(_{LP} \geq \text{OPT} \). Now set \( x_1 = x_2 = \cdots = x_{k-1} = 1, x_k = \alpha, \) and \( x_i = 0 \) for all \( i > k \). This is a feasible solution to the LP. We leave it as an exercise to the reader to argue that is an optimum solution. Therefore, \( p_1 + p_2 + \cdots + \alpha p_k = \text{OPT}' \geq \text{OPT} \). The first statement of the lemma follows from the second as \( \alpha \leq 1 \). \( \blacksquare \)
3.1.2 A Polynomial Time Approximation Scheme

Using the results from the last section, we make a few simple observations. Some of these lead to a better approximation.

Observation 3.2. If for all $i$, $p_i \leq \epsilon \cdot \text{OPT}$, GreedyKnapsack gives a $(1 - \epsilon)$ approximation.

Proof. Follows easily from Claim 3.1.1.

Observation 3.3. There are at most $\lceil \frac{1}{\epsilon} \rceil$ items with profit at least $\epsilon \cdot \text{OPT}$ in any optimal solution.

The next claim is perhaps more interesting and captures the intuition that the bad case for greedy happens only when there are “big” items.

Claim 3.1.2. If for all $i$, $s_i \leq \epsilon B$, GreedyKnapsack gives a $(1 - \epsilon)$ approximation.

Proof. We give a proof sketch via the LP relaxation. Recall that $k$ is the first item that did not fit in the knapsack. We make the following observation. Recall that $\text{OPT}_{LP}$ is the optimum value of LP relaxation. Suppose we reduce the knapsack capacity to $B' = s_1 + s_2 + \ldots s_{k-1}$ while keeping all the items the same. Let $\text{OPT}'_{LP}$ be the value for the new size. We claim that $\text{OPT}'_{LP} \geq \frac{B'}{B} \cdot \text{OPT}_{LP}$ — this is because we can take any feasible solution to the original LP and scale each variable by $B'/B$ to obtain a feasible solution with the new capacity. What is $\text{OPT}'_{LP}$? We note that Greedy will fill $B'$ to capacity with the first $k-1$ items and hence, $\text{OPT}'_{LP} = p_1 + \ldots + p_{k-1}$. Combining, we obtain that

$$p_1 + \ldots + p_{k-1} \geq \frac{B'}{B} \cdot \text{OPT}_{LP} \geq \frac{B'}{B} \cdot \text{OPT}.$$  

We note that $B' + s_k \geq B$ since item $k$ did not fit, and hence $B' \geq B - s_k \geq B - \epsilon B \geq (1 - \epsilon)B$. Therefore $B'/B \geq (1 - \epsilon)$ and this finishes the proof.

We may now describe the following algorithm. Let $\epsilon \in (0, 1)$ be a fixed constant and let $h = \lceil \frac{1}{\epsilon} \rceil$. We will try to guess the $h$ most profitable items in an optimal solution and pack the rest greedily.
**CHAPTER 3. KNAPSACK**

---

**GUESS H + GREEDY(N, B)**

1. For each \( S \subseteq N \) such that \(|S| \leq h \) and \( s(S) \leq B \) do
   
   A. Pack \( S \) in knapsack of size at most \( B \)
   
   B. Let \( i \) be the least profitable item in \( S \). Remove all items \( j \in N - S \) where \( p_j > p_i \).
   
   C. Run \textsc{GreedyKnapsack} on remaining items with remaining capacity \( B - \sum_{i \in S} s_i \)

2. Output best solution from above

---

**Theorem 3.4.** \textsc{Guess H + Greedy} gives a \((1 - \varepsilon)\) approximation and runs in \( O(n^{1/\varepsilon} + 1) \) time.

**Proof.** For the running time, observe that there are \( O(n^h) \) subsets of \( N \). For each subset, we spend linear time greedily packing the remaining items. The time initially spent sorting the items can be ignored thanks to the rest of the running time.

For the approximation ratio, consider a run of the loop where \( S \) actually is the \( h \) most profitable items in an optimal solution and the algorithm’s greedy stage packs the set of items \( A' \subseteq (N - S) \). Let \( \text{OPT}' \) be the optimal way to pack the smaller items in \( N - S \) so that \( \text{OPT} = p(S) + \text{OPT}' \). Let item \( k \) be the first item rejected by the greedy packing of \( N - S \). We know \( p_k \leq \varepsilon \text{OPT} \) so by Claim 3.1.1 \( p(A') \geq \text{OPT}' - \varepsilon \text{OPT} \). This means the total profit found in that run of the loop is \( p(S) + p(A') \geq (1 - \varepsilon) \text{OPT} \).

Note that for any fixed choice of \( \varepsilon > 0 \), the preceding algorithm runs in polynomial time. This type of algorithm is known as a *polynomial time approximation scheme* or PTAS. The term “scheme” refers to the fact that the algorithm varies with \( \varepsilon \). We say a maximization problem \( \Pi \) has a PTAS if for all \( \varepsilon > 0 \), there exists a polynomial time algorithm that gives a \((1 - \varepsilon)\) approximation \((1 + \varepsilon)\) for minimization problems). In general, one can often find a PTAS for a problem by greedily filling in a solution after first searching for a good basis on which to work. As described below, \textsc{Knapsack} actually has something stronger known as a *fully polynomial time approximation scheme* or FPTAS. A maximization problem \( \Pi \) has a FPTAS if for all \( \varepsilon > 0 \), there exists an algorithm that gives a \((1 - \varepsilon)\) approximation \((1 + \varepsilon)\) for minimization problems) and runs in time polynomial in both the input size and \( 1/\varepsilon \).
3.1.3 Rounding and Scaling

Earlier we mentioned exact algorithms based on dynamic programming that run in $O(nB)$ and $O(nP)$ time but noted that $B$ and $P$ may be prohibitively large. If we could somehow decrease one of those to be polynomial in $n$ without losing too much information, we might be able to find an approximation based on one of these algorithms. Let $p_{\text{max}} = \max_i p_i$ and note the following.

**Observation 3.5.** $p_{\text{max}} \leq \text{OPT} \leq np_{\text{max}}$

Now, fix some $\epsilon \in (0, 1)$. We want to scale the profits and round them to be integers so we may use the $O(nP)$ algorithm efficiently while still keeping enough information in the numbers to allow for an accurate approximation. For each $i$, let $p'_i = \lfloor \frac{n}{\epsilon p_{\text{max}}} p_i \rfloor$. Observe that $p'_i \leq \frac{n}{\epsilon}$ so now the sum of the profits $P'$ is at most $\frac{n^2}{\epsilon}$. Also, note that we lost at most $n$ profit from the scaled optimal solution during the rounding, but the scaled down $\text{OPT}$ is still at least $\frac{n}{\epsilon}$. We have only lost an $\epsilon$ fraction of the solution. This process of rounding and scaling values for use in exact algorithms has use in a large number of other maximization problems. We now formally state the algorithm **ROUND&SCALE** and prove its correctness and running time.

```
ROUND&SCALE(N, B)
1. For each $i$ set $p'_i = \lfloor \frac{n}{\epsilon p_{\text{max}}} p_i \rfloor$
2. Run exact algorithm with run time $O(nP')$ to obtain $A$
3. Output $A$
```

**Theorem 3.6.** **ROUND&SCALE** gives a $(1 - \epsilon)$ approximation and runs in $O\left(\frac{n^2}{\epsilon}\right)$ time.

**Proof.** The rounding can be done in linear time and as $P' = O\left(\frac{n^2}{\epsilon}\right)$, the dynamic programing portion of the algorithm runs in $O\left(\frac{n^3}{\epsilon}\right)$ time. To show the approximation ratio, let $\alpha = \frac{n}{\epsilon p_{\text{max}}}$ and let $A$ be the solution returned by the algorithm and $A'$ be the optimal solution. Observe that for all $X \subseteq N$, $\alpha p(X) - |X| \leq p'(X) \leq \alpha p(X)$ as the rounding lowers each scaled profit by at most 1. The algorithm returns the best choice for $A$ given the scaled and rounded values, so we know $p'(A) \geq p'(A')$.

$$p(A) \geq \frac{1}{\alpha} p'(A) \geq \frac{1}{\alpha} p'(A') \geq p(A') - \frac{n}{\alpha} = \text{OPT} - \epsilon p_{\text{max}} \geq (1 - \epsilon) \text{OPT}$$
It should be noted that this is not the best FPTAS known for Knapsack. In particular, [47] shows a FPTAS that runs in $O(n \log(1/e) + 1/e^4)$ time. There have been several improvements and we refer the reader to Chan’s paper for the latest [10].

### 3.2 Other Problems

There are many variants of Knapsack and it is a fundamental problem of interest in integer programming as well in several other areas. One can find a book length treatment in [43]. We close with an interesting variant.

**Multiple Knapsack:** The input now consists of $m$ knapsacks with capacities $B_1, B_2, \ldots, B_m$ and $n$ items with sizes and profits as in Knapsack. We again wish to pack items to obtain as large a profit as possible, except now we have more than one knapsack with which to do so. An interesting special case is when all the knapsack capacities are the same quantity $B$ which is related to the well known Bin Packing problem.
Chapter 4

Packing Problems

In the previous lecture we discussed the Knapsack problem. In this lecture we discuss other packing and independent set problems. We first discuss an abstract model of packing problems. Let \( N \) be a finite ground set. A collection of \( I \subset 2^N \) of subsets of \( N \) is said to be down closed if the following property is true: \( A \in I \) implies that for all \( B \subset A, B \in I \). A down closed collection is also often called and independence system. The sets in \( I \) are called independent sets. Given an independence family \((N, I)\) and a non-negative weight function \( w : N \rightarrow \mathbb{R}^+ \) the maximum weight independent set problem is to find \( \max_{S \in I} w(S) \). That is, find an independent set in \( I \) of maximum weight. Often we may be interested in the setting where all weights are 1 in which case we wish to find the maximum cardinality independent set. We discuss some canonical examples.

**Example 4.1.** Independent sets in graphs: Given a graph \( G = (V, E) \)
\( I = \{S \subseteq V \mid \text{ there are no edges between nodes in } S\} \). Here the ground set is \( V \). There are many interesting special cases of the graph problem. For instance problems arising from geometric objects such as intervals, rectangles, disks and others.

**Example 4.2.** Matchings in graphs: Given a graph \( G = (V, E) \) let \( I = \{M \subseteq E \mid M \text{ is a matching in } G\} \). Here the ground set is \( E \).

**Example 4.3.** Matroids: A matroid \( M = (N, I) \) is defined as a system where \( I \) is down closed and in addition satisfies the following key property: if \( A, B \in I \) and \( |B| > |A| \) then there is an element \( e \in B \setminus A \) such that \( A \cup \{e\} \in I \). There are many examples of matroids. We will not go into details here.

**Example 4.4.** Intersections of independence systems: given some \( k \) independence systems on the same ground set \((N, I_1), (N, I_2), \ldots, (N, I_k)\) the system defined by \((N, I_1 \cap I_2 \ldots \cap I_k)\) is also an independence system. Well-known examples include intersections of matroids.
4.1 Maximum Independent Set Problem in Graphs

A basic graph optimization problem with many applications is the maximum (weighted) independent set problem (MIS) in graphs.

Definition 4.1. Given an undirected graph $G = (V,E)$ a subset of nodes $S \subseteq V$ is an independent set (stable set) iff there is no edge in $E$ between any two nodes in $S$. A subset of nodes $S$ is a clique if every pair of nodes in $S$ have an edge between them in $G$.

The MIS problem is the following: given a graph $G = (V,E)$ find an independent set in $G$ of maximum cardinality. In the weighted case, each node $v \in V$ has an associated non-negative weight $w(v)$ and the goal is to find a maximum weight independent set. This problem is NP-Hard and it is natural to ask for approximation algorithms. Unfortunately, as the famous theorem below shows, the problem is extremely hard to approximate.

Theorem 4.2 (Håstad [29]). Unless $P = NP$ there is no $\frac{1}{2}$-approximation for MIS for any fixed $\epsilon > 0$ where $n$ is the number of nodes in the given graph.

Remark 4.1. The maximum clique problem is to find the maximum cardinality clique in a given graph. It is approximation-equivalent to the MIS problem; simply complement the graph.

The theorem basically says the following: there are a class of graphs in which the maximum independent set size is either less than $n^b$ or greater than $n^{1-b}$ and it is NP-Complete to decide whether a given graph falls into the former category or the latter.

The lower bound result suggests that one should focus on special cases, and several interesting positive results are known. First, we consider a simple greedy algorithm for the unweighted problem.

```
Greedy(G)
1. S ← ∅
2. While G is not empty do
   A. Let v be a node of minimum degree in G
   B. S ← S ∪ {v}
   C. Remove v and its neighbors from G
3. Output S
```
Theorem 4.3. Greedy outputs an independent set $S$ such that $|S| \geq \frac{n}{\Delta + 1}$ where $\Delta$ is the maximum degree of any node in the graph. Moreover $|S| \geq \frac{\alpha(G)}{\Delta}$ where $\alpha(G)$ is the cardinality of the largest independent set. Thus Greedy is a $1/\Delta$ approximation.

Proof. We upper bound the number of nodes in $V \setminus S$ as follows. A node $u$ is in $V \setminus S$ because it is removed as a neighbor of some node $v \in S$ when Greedy added $v$ to $S$. Charge $u$ to $v$. A node $v \in S$ can be charged at most $\Delta$ times since it has at most $\Delta$ neighbors. Hence we have that $|V \setminus S| \leq \Delta |S|$. Since every node is either in $S$ or $V \setminus S$ we have $|S| + |V \setminus S| = n$ and therefore $(\Delta + 1)|S| \geq n$ which implies that $|S| \geq \frac{n}{\Delta + 1}$.

We now argue that $|S| \geq \frac{\alpha(G)}{\Delta}$. Let $S^*$ be a largest independent set in $G$. As in the above proof we can charge each node $v$ in $S^* \setminus S$ to a node $u \in S \setminus S^*$ which is a neighbor of $v$. The number of nodes charged to a node $u \in S \setminus S^*$ is at most $\Delta$. Thus $|S^* \setminus S| \leq \Delta |S \setminus S^*|$.

\[\blacksquare\]

Exercise 4.1. Show that Greedy outputs an independent set of size at least $\frac{n}{2(d+1)}$ where $d$ is the average degree of $G$.

Remark 4.2. The well-known Turan’s theorem shows via a clever argument that there is always an independent set of size $\frac{n}{d+1}$ where $d$ is the average degree of $G$.

Remark 4.3. For the case of unweighted graphs one can obtain an approximation ratio of $\Omega\left(\frac{\log d}{d \log \log d}\right)$ where $d$ is the average degree. Surprisingly, under a complexity theory conjecture called the Unique-Games conjecture it is known to be NP-Hard to approximate MIS to within a factor of $O\left(\frac{\log^2 \Delta}{\Delta}\right)$ in graphs with maximum degree $\Delta$ when $\Delta$ is sufficiently large.

Exercise 4.2. Consider the weighted MIS problem on graphs of maximum degree $\Delta$. Alter Greedy to sort the nodes in non-increasing order of the weight and show that it gives a $\frac{1}{\Delta}$-approximation. Can one obtain an $\Omega(1/d)$-approximation for the weighted case where $d$ is the average degree?

LP Relaxation: One can formulate a simple linear-programming relaxation for the (weighted) MIS problem where we have a variable $x(v)$ for each node $v \in V$ indicating whether $v$ is chosen in the independent set or not. We have constraints which state that for each edge $(u, v)$ only one of $u$ or $v$ can be chosen.

\[
\text{maximize } \sum_{v \in V} w(v) x(v) \\
\text{subject to } x(u) + x(v) \leq 1 \quad (u, v) \in E \\
x(v) \in [0, 1] \quad v \in V
\]
Although the above is a valid integer programming relaxation of MIS when the variables are constrained to be in \( \{0, 1\} \), it is not a particularly useful formulation for the following simple reason.

**Claim 4.1.1.** For any graph the optimum value of the above LP relaxation is at least \( w(V)/2 \). In particular, for the unweighted case it is at least \( n/2 \).

Simply set each \( x(v) \) to 1/2!

One can obtain a strengthened formulation below by observing that if \( S \) is clique in \( G \) then any independent set can pick at most one node from \( S \).

\[
\begin{align*}
\text{maximize} & \quad \sum_{v \in V} w(v)x(v) \\
\text{subject to} & \quad \sum_{v \in S} x(v) \leq 1 \quad \text{\( S \) is a clique in \( G \)} \\
& \quad x(v) \in [0, 1] \quad v \in V
\end{align*}
\]

The above linear program has an exponential number of constraints, and it cannot be solved in polynomial time in general, but for some special cases of interest the above linear program can indeed be solved (or approximately solved) in polynomial time and leads to either exact algorithms or good approximation bounds.

**Approximability of Vertex Cover and MIS:** The following is a basic fact and is easy to prove.

**Fact 4.1.** In any graph \( G = (V, E) \), \( S \) is a vertex cover in \( G \) if and only if \( V \setminus S \) is an independent set in \( G \). Thus \( \alpha(G) + \beta(G) = |V| \) where \( \alpha(G) \) is the size of a maximum independent set in \( G \) and \( \beta(G) \) is the size of a minimum vertex cover in \( G \).

The above shows that if one of Vertex Cover or MIS is NP-Hard then the other is as well. We have seen that Vertex Cover admits a 2-approximation while MIS admits no constant factor approximation. It is useful to see why a 2-approximation for Vertex Cover does not give any useful information for MIS even though \( \alpha(G) + \beta(G) = |V| \). Suppose \( S^* \) is an optimal vertex cover and has size \( \geq |V|/2 \). Then a 2-approximation algorithm is only guaranteed to give a vertex cover of size \( |V|! \). Hence one does not obtain a non-trivial independent set by complementing the approximate vertex cover.

**Some special cases of MIS:** We mention some special cases of MIS that have been considered in the literature, this is by no means an exhaustive list.

- Interval graphs; these are intersection graphs of intervals on a line. An exact algorithm can be obtained via dynamic programming and one can solve more general versions via linear programming methods.
• Note that a maximum (weight) matching in a graph $G$ can be viewed as a maximum (weight) independent set in the line-graph of $G$ and can be solved exactly in polynomial time. This has been extended to what are known as claw-free graphs.

• Planar graphs and generalizations to bounded-genus graphs, and graphs that exclude a fixed minor. For such graphs one can obtain a PTAS due to ideas originally from Brenda Baker.

• Geometric intersection graphs. For example, given $n$ disks on the plane find a maximum number of disks that do not overlap. One could consider other (convex) shapes such as axis parallel rectangles, line segments, pseudo-disks etc. A number of results are known. For example a PTAS is known for disks in the plane. An $\Omega(\frac{1}{\log n})$-approximation for axis-parallel rectangles in the plane when the rectangles are weighted and an $\Omega(\frac{1}{\log \log n})$-approximation for the unweighted case. For the unweighted case, very recently, Mitchell obtained a constant factor approximation!

4.1.1 Elimination Orders and MIS

We have seen that a simple Greedy algorithm gives a $\Delta$-approximation for MIS in graphs with max degree $\Delta$. One can also get a $\Delta$ approximation for a larger class of $\Delta$-degenerate graphs. To motivate degenerate graphs consider the class of planar graphs. The maximum degree of a planar graph need not be small. Nevertheless, via Euler’s theorem, we know that every planar graph has a vertex of degree at most 5 since the maximum number of edges in a planar graph is at most $3n - 6$. Moreover, every subgraph of a planar graph is planar, and hence the Greedy algorithm will repeatedly find a vertex of degree at most 5 in each iteration. From this one can show that Greedy gives a 1/5-approximation for MIS in planar graphs. Now consider the intersection graph of a collection of intervals on the real line. That is, we are given $n$ intervals $I_1, I_2, \ldots, I_n$ where each $I_i = [a_i, b_i]$ for real numbers $a_i \leq b_i$. The goal is to find a maximum number of the intervals in the given set of intervals which do not overlap. This is the same as finding MIS in the intersection graph of the intervals - the graph is obtained by creating a vertex $v_i$ for each $I_i$, and by adding edges $v_i v_j$ if $I_i$ and $I_j$ overlap. It is well-known that greedily picking intervals in earliest finish time order (ordering them according to $b_i$ values) is optimal; the reader should try to prove this. Can one understand the analysis of all these examples in a unified fashion? Yes. For this purpose we consider the class of inductive $k$-independent graphs considered by by Akcoglu et al. [2] and later again by Ye and Borodin [68].
For a vertex \( v \) in a graph we use \( N(v) \) denote the neighbors of \( v \) (not including \( v \) itself). For a graph \( G = (V, E) \) and \( S \subseteq V \) we use \( G[S] \) to denote the subgraph of \( G \) induced by \( S \).

**Definition 4.4.** An undirected graph \( G = (V, E) \) is inductive \( k \)-independent if there is an ordering of the vertices \( v_1, v_2, \ldots, v_n \) such that for \( 1 \leq i \leq n \), \( a(G[N(v_i)] \cap \{v_{i+1}, \ldots, v_n\}) \leq k \).

Graphs which are inductively 1-independent have a perfect elimination ordering and are called chordal graphs because they have an alternate characterization. A graph is chordal iff each cycle \( C \) in \( G \) has a chord (an edge connecting two nodes of \( C \) which is not an edge of \( C \)), or in other words there is no induced cycle of length more than 3.

**Exercise 4.3.** Prove that the intersection graph of intervals is chordal.

**Exercise 4.4.** Prove that if \( \Delta(G) \leq k \) then \( G \) is inductively \( k \)-independent. Prove that if \( G \) is \( k \)-degenerate then \( G \) is inductively \( k \)-independent.

The preceding shows that planar graphs are inductively 5-independent. In fact one can show something stronger. They are inductively 3-independent. Given a graph \( G \) one can ask whether there is an algorithm that checks whether \( G \) is inductively \( k \)-independent. There is such an algorithm that runs in time \( O(k^2 n^{k+2}) \) [68]. A classical result shows how to recognize chordal graphs \( (k = 1) \) in linear time. However, most of the useful applications arise by showing that a certain class of graphs are inductively \( k \)-independent for some small value of \( k \). See [68] for several examples.

**Exercise 4.5.** Prove that the Greedy algorithm that considers the vertices in the inductive \( k \)-independent order gives a \( \frac{1}{k} \)-approximation for MIS.

Interestingly one can obtain a \( \frac{1}{\sqrt{k}} \)-approximation for the maximum weight independent set problem in inductively \( k \)-independent graphs. The algorithm is simple and runs in linear time but is not obvious. To see this consider the weighted problem for intervals. The standard algorithm to solve this is via dynamic programming. However, one can obtain an optimum solution for all chordal graphs (given the ordering). We refer the reader to [68] for the algorithm and proof (originally from [2]). Showing a \( \Omega(1/k) \)-approximation is easier.

### 4.2 The efficacy of the Greedy algorithm for a class of Independence Families

The Greedy algorithm can be defined easily for an arbitrary independence system. It iteratively adds the best element to the current independent set while
maintaining feasibility. Note that the implementation of the algorithm requires having an oracle to find the best element to add to a current independent set $S$.

<table>
<thead>
<tr>
<th>Greedy$(N, I)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $S \leftarrow \emptyset$</td>
</tr>
<tr>
<td>2. While (TRUE)</td>
</tr>
<tr>
<td>A. Let $A \leftarrow {e \in N \setminus S \mid S + e \in I}$</td>
</tr>
<tr>
<td>B. If $A = \emptyset$ break</td>
</tr>
<tr>
<td>C. $e \leftarrow \arg\max_{e \in A} w(e)$</td>
</tr>
<tr>
<td>D. $S \leftarrow S \cup {e}$</td>
</tr>
<tr>
<td>3. Output $S$</td>
</tr>
</tbody>
</table>

**Exercise 4.6.** Prove that the Greedy algorithm gives a $1/2$-approximation for the maximum weight matching problem in a general graph. Also prove that this bound is tight even in bipartite graphs. Note that max weight matching can be solved exactly in polynomial time.

**Remark 4.4.** It is well-known that the Greedy algorithm gives an optimum solution when $(N, I)$ is a matroid. Kruskal’s algorithm for min/max weight spanning tree is a special case of this fact.

It is easy to see that Greedy does poorly for MIS problem in general graphs. A natural question is what properties of $I$ enable some reasonable performance guarantee for Greedy. A very general result in this context has been established due to Jenkyn’s generalizing several previous results. In order to state the result we set up some notation. Given an independence system $(N, I)$ we say that a set $A \in I$ is a base if it is a maximal independent set. It is well-known that in a matroid $M$ all bases have the same cardinality. However this is not true in general independence system.

**Definition 4.5.** An independence system $(N, I)$ is a $k$-system if for any two bases $A, B \in I$, $|A| \leq k|B|$. That is, the ratio of the cardinality of a maximum base and the cardinality of a minimum base is at most $k$.

The following theorem is not too difficult but not so obvious either.

**Theorem 4.6.** Greedy gives a $1/k$-approximation for the maximum weight independent set problem in a $k$-system.
The above theorem generalizes and unifies several examples that we have seen so far including MIS in bounded degree graphs, matchings, matroids etc. How does one see that a given independence system is indeed a $k$-system for some parameter $k$? For instance matchings in graphs form a 2-system. The following simple lemma gives an easy way to argue that a given system is a $k$-system.

**Lemma 4.1.** Suppose $(N, I)$ is an independence system with the following property: for any $A \in I$ and $e \in N \setminus A$ there is a set $Y \subset A$ such that $|Y| \leq k$ and $(A \setminus Y) \cup \{e\} \in I$. Then $I$ is a $k$-system.

We leave the proof of the above as an exercise.

We refer the reader to [21, 50] for analysis of Greedy in $k$-systems and other special cases.

### 4.3 Randomized Rounding with Alteration for Packing Problems

The purpose of this section to highlight a technique for rounding LP relaxations for packing problems. We will consider a simple example, namely the maximum weight independent set problem in interval graphs. Recall that we are given $n$ intervals $I_1, I_2, \ldots, I_n$ with non-negative weights $w_1, \ldots, w_n$ and the goal is to find a maximum weight subset of them which do not overlap. Let $I_i = [a_i, b_i]$ and let $p_1, p_2, \ldots, p_m$ be the collection of end points of the intervals. We can write a simple LP relaxation for this problem. For each interval $i$ we have a variable $x_i \in [0, 1]$ to indicate whether $I_i$ is chosen or not. For each point $p_j$, among all intervals that contain it, at most one can be chosen. These are clique constraints in the underlying interval graph.

$$\text{maximize} \quad \sum_{i=1}^{n} w_i x_i$$

subject to

$$\sum_{i:p_j \in I_i} x_i \leq 1 \quad 1 \leq j \leq m$$

$$x_i \in [0, 1] \quad 1 \leq i \leq n$$

Note that it is important to retain the constraint that $x_i \leq 1$. Interestingly it is known that the LP relaxation defines an integer polytope and hence one can solve the integer program by solving the LP relaxation! This is because the incidence matrix defining the LP is totally unimodular (TUM). We refer the
reader to books on combinatorial optimization for further background on this topic. Here we assume that we do not know the integer properties of the LP. We will round it via a technique that is powerful and generalizes to NP-Hard variants of the interval scheduling problem among many others.

Suppose we solve the LP and obtain an optimum fraction solution \( x^* \). How do we round to obtain an integer solution whose value is close to that of \( x^* \)? Suppose we randomly choose \( \frac{1}{2} \) with probability \( c \) for some \( c \leq 1 \). Let \( R \) be the random set of chosen intervals. Then the expected weight of \( R \), by linearity of expectation, is \( \frac{1}{2} \sum_i w_i x_i^* \geq c \cdot \text{OPT} \). However, it is highly likely that the random solution \( R \) is not going to be feasible. Some constraint will be violated. The question is how we can fix or alter \( R \) to find a subset \( R' \subseteq R \) such that \( R' \) is a feasible solution and the expected value of \( R' \) is not too much smaller than that of \( R \). This depends on the independence structure.

Here we illustrate this via the interval problem. Without loss of generality we assume that \( I_1, \ldots, I_n \) are sorted by their right endpoint. In other words the order is a perfect elimination order for the underlying interval graph.

**Rounding-with-Alteration**

1. Let \( x \) be an optimum fractional solution
2. Round each \( i \) to 1 independently with probability \( x_i / 2 \). Let \( x' \) be rounded solution.
3. \( R \leftarrow \{ i \mid x'_i = 1 \} \)
4. \( S \leftarrow \emptyset \)
5. For \( i = n \) down to 1 do
   A. If \( (i \in R) \) and \( (S \cup \{i\} \) is feasible) then \( S \leftarrow S \cup \{i\} \)
6. Output feasible solution \( S \)

The algorithm consists of two phases. The first phase is a simple selection phase via independent randomized rounding. The second phase is deterministic and is a greedy pruning step in the reverse elimination order. To analyze the expected value of \( S \) we consider two binary random variables for each \( i \), \( Y_i \) and \( Z_i \). \( Y_i \) is 1 if \( i \in R \) and 0 otherwise. \( Z_i \) is 1 if \( i \in S \) and 0 otherwise.

By linearity of expectation,

**Claim 4.3.1.** \( \mathbb{E}[w(S)] = \sum_i w_i \mathbb{E}[Z_i] = \sum_i w_i \mathbb{P}[Z_i = 1] \).

Via the independent randomized rounding in the algorithm.
Claim 4.3.2. $P[Y_i = 1] = x_i/2$.

How do we analyze $P[Z_i = 1]$? The random variables $Z_1, \ldots, Z_n$ are not independent and could be highly correlated even though $Y_1, \ldots, Y_n$ are independent. For this purpose we try to understand $P[Z_i = 0 \mid Y_i = 1]$ which is the conditional probability that an interval $I_i$ that is chosen in the first step is rejected in the pruning phase. We often would not be able to get an exact estimate of this quantity but we can upper bound it as follows. Here the ordering plays a crucial role. Why would $I_i$ be rejected in the pruning phase? Note that when $I_i$ is considered in the pruning phase, the only intervals that have been considered have their right end points after the right end point of $I_i$. Let $A_i = \{ j \mid j > i \text{ and } I_i \text{ and } I_j \text{ intersect at } b_j \}$ be the potential set of intervals that can cause $i$ to be rejected. Recall that the LP implies the following constraint:

$$x_i + \sum_{j \in A} x_j \leq 1$$

at the point $b_j$. Let $E_1$ be the event that $I_i$ is rejected in the pruning phase. Let $E_2$ be the event that at least one of the intervals in $A$ is selected in the first phase. Note that $E_1$ can happen only if $E_2$ happens. Thus $P[E_1] \leq P[E_2]$. In general we try to upper bound $P[E_2]$. In this simple case we have an exact formula for it.

$$P[E_2] = 1 - \prod_{j \in A} P[Y_j = 0] = 1 - \prod_{j \in A} (1 - x_j/2).$$

We claim that $P[E_2] \leq \sum_{j \in A} x_j/2 \leq 1/2$. One can derive this by showing that $\prod_{j \in A} (1 - x_j/2)$ subject to $\sum_{j \in A} x_j/2 \leq 1/2$ is at least 1/2. Another way of doing this is via Markov’s inequality. Let $T = \sum_{j \in A} Y_j$ be the number of intervals from $A$ selected in the first phase. $E[T] \leq \sum_{j \in A} x_j/2 < 1/2$. By Markov’s inequality $P[T \geq 2E[T]] \leq 1/2$. $E_2$ is the event that $P[T \geq 1]$.

Using the claim,

$$P[Z_i = 1 \mid Y_i = 1] = 1 - P[Z_i = 0 \mid Y_i = 1] \geq 1/2.$$ 

This allows us to lower bound the expected weight of the solution output by the algorithm, and yields a randomized 1/4 approximation.

Claim 4.3.3. $\mathbb{E}[w(S)] \geq \sum_i w_i x_i/4$.

Proof. We have

$$\mathbb{E}[w(S)] = \sum_i w_i P[Z_i = 1] = \sum_i w_i P[Y_i = 1] P[Z_i = 1 \mid Y_i = 1] \geq \sum_i w_i (x_i/2) \geq \sum_i w_i x_i/4.$$ 

This type of rounding has applications to a variety of settings - see [14] for applications and the general framework called contention resolution schemes.
4.4 Packing Integer Programs (PIPs)

We can express the \textsc{Knapsack} problem as the following integer program. We scaled the knapsack capacity to 1 without loss of generality.

\begin{align*}
\text{maximize } & \sum_{i=1}^{n} p_i x_i \\
\text{subject to } & \sum_{i} s_i x_i \leq 1 \\
& x_i \in \{0, 1\} \quad 1 \leq i \leq n
\end{align*}

More generally if have multiple linear constraints on the “items” we obtain the following integer program.

**Definition 4.7.** A packing integer program (PIP) is an integer program of the form

\[
\max \{ w^T x \mid Ax \leq b, x \in \{0, 1\}^n \}
\]

where \( w \) is a \( 1 \times n \) non-negative vector and \( A \) is a \( m \times n \) matrix with entries in \([0, 1]\). We call it a \( \{0, 1\}\)-PIP if all entries are in \([0, 1]\).

In some cases it is useful/natural to define the problem as

\[
\max \{ w^T x \mid Ax \leq b, x \in \{0, 1\}^n \}
\]

where entries in \( A \) and \( b \) are required to rational/integer valued. We can convert it into the above form by dividing each row of \( A \) by \( b_i \).

When \( m \) the number of rows of \( A \) (equivalently the constraints) is small the problem is tractable. It is some times called the \( m \)-dimensional knapsack and one can obtain a PTAS for any fixed constant \( m \). However, when \( m \) is large we observe that \( \text{MIS} \) can be cast as a special case of \( \{ 0, 1 \} \)-PIP. It corresponds exactly to the simple integer/linear program that we saw in the previous section. Therefore the problem is at least as hard to approximate as \( \text{MIS} \). Here we show via a clever LP-rounding idea that one can generalize the notion of bounded-degree to \textit{column-sparsity} in PIPs and obtain a related approximation. We will then introduce the notion of \textit{width} of the constraints and show how it allows for improved bounds.

**Definition 4.8.** A PIP is \( k \)-column-sparse if the number of non-zero entries in each column of \( A \) is at most \( k \). A PIP has width \( W \) if \( \max_{i,j} A_{ij}/b_i \leq 1/W \).

4.4.1 Randomized Rounding with Alteration for PIPs

We saw that randomized rounding gave an \( O(\log n) \) approximation algorithm for the \textsc{Set Cover} problem which is a canonical covering problem. Here we will consider the use of randomized rounding for packing problems. Let \( x \) be an optimum fractional solution to the natural LP relaxation of a PIP where we
replace the constraint $x \in \{0,1\}^n$ by $x \in [0,1]^n$. Suppose we apply independent randomized rounding where we set $x'_i$ to 1 with probability $x_i$. Let $x'$ be the resulting integer solution. The expected weight of this solution is exactly $\sum_i w_i x_i$ which is the LP solution value. However, $x'$ may not satisfy the constraints given by $Ax \leq b$. A natural strategy to try to satisfy the constraints is to set $x'_i$ to 1 with probability $c x_i$ where $c < 1$ is some scaling constant. This may help in satisfying the constraints because the scaling creates some room in the constraints; we now have that the expected solution value is $c \sum_i w_i x_i$, a loss of a factor of $c$.

Scaling by itself does not allow us to claim that all constraints are satisfied with good probability. A very useful technique in this context is the technique of alteration; we judiciously fix/alter the rounded solution $x'$ to force it to satisfy the constraints by setting some of the variables that are 1 in $x'$ to 0. The trick is to do this in such a way as to have a handle on the final probability that a variable is set to 1. We will illustrate this for the Knapsack problem and then generalize the idea to $k$-sparse PIPs. The algorithms we present are from [6]. See [14] for further applications and related problems.

**Rounding for Knapsack:** Consider the Knapsack problem. It is convenient to think of this in the context of PIPs. So we have $ax \leq 1$ where $a_i$ now represents the size of item $i$ and the knapsack capacity is 1; $w_i$ is the weight of item. Suppose $x$ is a fractional solution. Call an item $i$ “big” if $a_i > 1/2$ and otherwise it is “small”. Let $S$ be the indices of small items and $B$ the indices of the big items. Consider the following rounding algorithm.

<table>
<thead>
<tr>
<th>Rounding-with-Alteration for Knapsack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Let $x$ be an optimum fractional solution</td>
</tr>
<tr>
<td>2. Round each $i$ to 1 independently with probability $x_i/4$. Let $x'$ be rounded solution.</td>
</tr>
<tr>
<td>3. $x'' = x'$</td>
</tr>
<tr>
<td>4. If $(x'_i = 1$ for exactly one big item $i)$</td>
</tr>
<tr>
<td>A. For each $j \neq i$ set $x''_j = 0$</td>
</tr>
<tr>
<td>5. Else If $(\sum_{i \in S} s_i x'_i &gt; 1$ or two or more big items are chosen in $x'$)</td>
</tr>
<tr>
<td>A. For each $j$ set $x''_j = 0$</td>
</tr>
<tr>
<td>6. Output feasible solution $x''$</td>
</tr>
</tbody>
</table>

In words, the algorithm alters the rounded solution $x'$ as follows. If exactly
one big item is chosen in \( x' \) then the algorithm retains that item and rejects all the other small items. Otherwise, the algorithm rejects all items if two or more big items are chosen in \( x' \) or if the total size of all small items chosen in \( x' \) exceeds the capacity.

The following claim is easy to verify.

**Claim 4.4.1.** The integer solution \( x'' \) is feasible.

Now let us analyze the probability of an item \( i \) being present in the final solution. Let \( \mathcal{E}_1 \) be the event that \( \sum_{i \in S} a_i \cdot x'_i > 1 \), that is the sum of the sizes of the small items chose in \( x' \) exceeds the capacity. Let \( \mathcal{E}_2 \) be the event that at least one big item is chosen in \( x' \).

**Claim 4.4.2.** \( \Pr[\mathcal{E}_1] \leq 1/4. \)

*Proof.* Let \( X_i = \sum_{i \in S} a_i \cdot x'_i \) be the random variable that measures the sum of the sizes of the small items chosen. We have, by linearity of expectation, that

\[
\mathbb{E}[X_i] = \sum_{i \in S} a_i \cdot \mathbb{E}[x'_i] = \sum_{i \in S} a_i / 4 \leq 1/4.
\]

By Markov’s inequality, \( \Pr[X_i > 1] \leq \mathbb{E}[X_i] / 1 \leq 1/4. \)

**Claim 4.4.3.** \( \Pr[\mathcal{E}_2] \leq 1/2. \)

*Proof.* Since the size of each big item in \( B \) is at least \( 1/2 \), we have \( 1 \geq \sum_{i \in B} a_i \cdot x_i \geq \sum_{i \in B} x_i / 2 \). Therefore \( \sum_{i \in B} x_i / 4 \leq 1/2 \). Event \( \mathcal{E}_2 \) happens if some item \( i \in B \) is chosen in the random selection. Since \( i \) is chosen with probability \( x_i / 4 \), by the union bound, \( \Pr[\mathcal{E}_2] \leq \sum_{i \in B} x_i / 4 \leq 1/2. \)

**Lemma 4.2.** Let \( Z_i \) be the indicator random variable that is 1 if \( x''_i = 1 \) and 0 otherwise. Then \( \mathbb{E}[Z_i] = \Pr[Z_i = 1] \geq x_i / 16. \)

*Proof.* We consider the binary random variable \( X_i \) which is 1 if \( x'_i = 1 \). We have \( \mathbb{E}[X_i] = \Pr[X_i = 1] = x_i / 4. \) We write

\[
\Pr[Z_i = 1] = \Pr[X_i = 1] \cdot \Pr[Z_i = 1 \mid X_i = 1] = x_i / 4 \cdot \Pr[Z_i = 1 \mid X_i = 1].
\]

To lower bound \( \Pr[Z_i = 1 \mid X_i = 1] \) we upper bound the probability \( \Pr[Z_i = 0 \mid X_i = 1] \), that is, the probability that we reject \( i \) conditioned on the fact that it is chosen in the random solution \( x' \).

First consider a big item \( i \) that is chosen in \( x' \). Then \( i \) is rejected iff if another big item is chosen in \( x' \); the probability of this can be upper bounded by \( \Pr[\mathcal{E}_1] \).
If item $i$ is small then it is rejected if and only if $E_2$ happens or if a big item is chosen which happens with $P[E_1]$. In either case

$$P[Z_i = 0|X_i = 1] \leq P[E_1] + P[E_2] \leq 1/4 + 1/2 = 3/4.$$ 

Thus,

$$P[Z_i = 1] = P[X_i = 1] \cdot P[Z_i = 1 | X_i = 1] = \frac{x_i}{4}(1 - P[Z_i = 0 | X_i = 1]) \geq \frac{x_i}{16}. \quad \blacksquare$$

One can improve the above analysis to show that $P[Z_i = 1] \geq x_i/8$.

**Theorem 4.9.** The randomized algorithm outputs a feasible solution of expected weight at least $\sum_{i=1}^{n} w_i x_i/16$.

**Proof.** The expected weight of the output is

$$\mathbb{E}[\sum_{i} w_i x_i”] = \sum_{i} w_i \mathbb{E}[Z_i] \geq \sum_{i} w_i x_i/16$$

where we used the previous lemma to lower bound $\mathbb{E}[Z_i]. \quad \blacksquare$

**Rounding for $k$-sparse PIPs:** We now extend the rounding algorithm and analysis above to $k$-sparse PIPs. Let $x$ be a feasible fractional solution to $\max \{ w x | Ax \leq 1, x \in [0, 1]^n \}$. For a column index $i$ we let $N(i) = \{ j | A_{j,i} > 0 \}$ be the indices of the rows in which $i$ has a non-zero entry. Since $A$ is $k$-column-sparse we have that $|N(i)| \leq k$ for $1 \leq i \leq n$. When we have more than one constraint we cannot classify an item/index $i$ as big or small since it may be big for some constraints and small for others. We say that $i$ is small for constraint $j \in N(i)$ if $A_{j,i} \leq 1/2$ otherwise $i$ is big for constraint $j$. Let $S_j = \{ i | j \in N(i), \text{ and } i \text{ small for } j \}$ be the set of all small columns for $j$ and $B_j = \{ i | j \in N(i), \text{ and } i \text{ small for } j \}$ be the set of all big columns for $j$. Note that $S_j \cap B_j$ is the set of all $i$ with $A_{j,i} > 0$. 


CHAPTER 4. PACKING PROBLEMS

Rounding-with-Alteration for $k$-sparse PIPs

1. Let $x$ be an optimum fractional solution
2. Round each $i$ to 1 independently with probability $x_i/(4k)$. Let $x'$ be rounded solution.
3. $x'' = x'$
4. For $j = 1$ to $m$ do
   A. If ($x'_i = 1$ for exactly one $i \in B_j$)
      1. For each $h \in S_j \cup B_j$ and $h \neq i$ set $x''_h = 0$
   B. Else If ($\sum_{i \in S_j} A_{ji}x'_i > 1$ or two or more items from $B_j$ are chosen in $x'$)
      1. For each $h \in S_j \cup B_j$ set $x''_h = 0$
5. Output feasible solution $x''$

The algorithm, after picking the random solution $x'$, alters it as follows: it applies the previous algorithm’s strategy to each constraint $j$ separately. Thus an element $i$ can be rejected at different constraints $j \in N(i)$. We need to bound the total probability of rejection. As before, the following claim is easy to verify.

Claim 4.4.4. The integer solution $x''$ is feasible.

Now let us analyze the probability of an item $i$ being present in the final solution. Let $\mathcal{E}_1(j)$ be the event that $\sum_{i \in S_j} A_{ji}x'_i > 1$, that is the sum of the sizes of the items that are small for $j$ in $x'$ exceed the capacity. Let $\mathcal{E}_2(j)$ be the event that at least one big item for $j$ is chosen in $x'$. The following claims follow from the same reasoning as the ones before with the only change being the scaling factor.

Claim 4.4.5. $P[\mathcal{E}_1(j)] \leq 1/(4k)$.
Claim 4.4.6. $P[\mathcal{E}_2(j)] \leq 1/(2k)$.

Lemma 4.3. Let $Z_i$ be the indicator random variable that is 1 if $x''_i = 1$ and 0 otherwise. Then $E[Z_i] = P[Z_i = 1] \geq x_i/(16k)$.

Proof. We consider the binary random variable $X_i$ which is 1 if $x'_i = 1$ after the randomized rounding. We have $E[X_i] = P[X_i = 1] = x_i/(4k)$. We write

$$P[Z_i = 1] = P[X_i = 1] \cdot P[Z_i = 1 \mid X_i = 1] = \frac{x_i}{4k} P[Z_i = 1 \mid X_i = 1].$$
We upper bound the probability $P[Z_i = 0 | X_i = 1]$, that is, the probability that we reject $i$ conditioned on the fact that it is chosen in the random solution $x'$. We observe that

$$P[Z_i = 0 | X_i = 1] \leq \sum_{j \in N(i)} (P[\mathcal{E}_1(j)] + P[\mathcal{E}_2(j)]) \leq k(1/(4k) + 1/(2k)) \leq 3/4.$$ 

We used the fact that $N(i) \leq k$ and the claims above. Therefore,

$$P[Z_i = 1] = P[X_i = 1] \cdot P[Z_i = 1 | X_i = 1] = \frac{x_i}{4k}(1 - P[Z_i = 0 | X_i = 1]) \geq \frac{x_i}{16k}.$$ 

The theorem below follows by using the above lemma and linearity of expectation to compare the expected weight of the output of the randomized algorithm with that of the fractional solution.

**Theorem 4.10.** The randomized algorithm outputs a feasible solution of expected weight at least $\sum_{i=1}^n w_i x_i / (16k)$. There is $1/(16k)$-approximation for $k$-sparse PIPs.

**Larger width helps:** We saw during the discussion on the knapsack problem that if all items are small with respect to the capacity constraint then one can obtain better approximations. For PIPs we defined the width of a given instance as $W$ if $\max_{i,j} A_{ij}/b_i \leq 1/W$; in other words no single item is more than $1/W$ times the capacity of any constraint. One can show using a very similar algorithm and analysis as above that the approximation bound improves to $\Omega(1/k^{[W]})$ for instance with width $W$. Thus if $W = 2$ we get a $\Omega(1/\sqrt{k})$ approximation instead of $\Omega(1/k)$-approximation. More generally when $W \geq c \log k/\epsilon$ for some sufficiently large constant $c$ we can get a $(1 - \epsilon)$-approximation. Thus, in the setting with multiple knapsack constraints, the notion of small with respect to capacities is that in each constraint the size of the item is $\leq \frac{c \epsilon}{\log k}$ times the capacity of that constraint.
Chapter 5

Load Balancing and Bin Packing

This chapter is based on notes first scribed by Rachit Agarwal.

In the last lecture, we studied the K/n and a problem. The problem is an NP-hard problem but does admit a pseudo-polynomial time algorithm and can be solved efficiently if the input size is small. We used this pseudo-polynomial time algorithm to obtain an FPTAS for Knapsack. In this lecture, we study another class of problems, known as strongly NP-hard problems.

Definition 5.1 (Strongly NP-hard Problems). An NPO problem \( \pi \) is said to be strongly NP-hard if it is NP-hard even if the inputs are polynomially bounded in combinatorial size of the problem \(^1\).

Many NP-hard problems are in fact strongly NP-hard. If a problem \( \Pi \) is strongly NP-hard, then \( \Pi \) does not admit a pseudo-polynomial time algorithm. We study two such problems in this lecture, MULTIPROCESSOR SCHEDULING and Bin Packing.

5.1 Load Balancing / MultiProcessor Scheduling

A central problem in scheduling theory is to design a schedule such that the finishing time of the last jobs (also called makespan) is minimized. This problem is often referred to as the Load Balancing, the Minimum Makespan Scheduling or MultiProcessor Scheduling problem.

\(^1\)An alternative definition: A problem \( \pi \) is strongly NP-hard if every problem in NP can be polynomially reduced to \( \pi \) in such a way that numbers in the reduced instance are always written in unary

56
5.1.1 Problem Description

In the MultiProcessor Scheduling problem, we are given \( m \) identical machines \( M_1, \ldots, M_m \) and \( n \) jobs \( J_1, J_2, \ldots, J_n \). Job \( J_i \) has a processing time \( p_i \geq 0 \) and the goal is to assign jobs to the machines so as to minimize the maximum load\(^2\).

5.1.2 Greedy Algorithm

Consider the following greedy algorithm for the MultiProcessor Scheduling problem which we will call Greedy MultiProcessor Scheduling.

<table>
<thead>
<tr>
<th>Greedy MultiProcessor Scheduling:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order (list) the jobs arbitrarily</td>
</tr>
<tr>
<td>For ( i = 1 ) to ( n ) do</td>
</tr>
<tr>
<td>Assign Job ( J_i ) to the machine with least current load</td>
</tr>
<tr>
<td>Update load of the machine that receives job ( J_i )</td>
</tr>
</tbody>
</table>

This algorithm is also called a list scheduling algorithm following Graham's terminology from his paper from 1966 [24]. The list is the order in which the jobs are processed and changing it creates different schedules. We prove that the Greedy MultiProcessor Scheduling algorithm gives a \((2 - 1/m)\)-approximation.

**Theorem 5.2.** Greedy MultiProcessor Scheduling algorithm gives a \((2 - 1/m)\)-approximation for any list.

To prove the theorem, we will make use of two lower bounds on the length of the optimal schedule which we denote by OPT.

**Observation 5.3.** \( \text{OPT} \geq \frac{\sum_i p_i}{m} \).

**Observation 5.4.** \( \text{OPT} \geq \max_i p_i \).

We leave the proofs of the observations as easy exercises.

**Proof of Theorem 5.2.** Fix the list and let \( L \) denote the makespan of the Greedy MultiProcessor Scheduling algorithm. Let \( L_i \) denote the load of machine \( M_i \) and let \( M_{\ell} \) be the most heavily loaded machine in the schedule by Greedy MultiProcessor Scheduling algorithm.

Let \( J_k \) be the last job assigned to \( M_{\ell} \). Since Greedy MultiProcessor Scheduling algorithm assigns a job to the machine that is least loaded, all machines must be loaded at least \( L - p_k \) at the time of assigning \( J_k \). Hence, we have:

\[
\left( \sum_{i=1}^{n} p_i \right) - p_k \geq m (L - p_k)
\]  

\[(5.1)\]

\(^2\)The load of a machine is defined as the sum of the processing times of jobs that are assigned to that machine.
which implies:

\[
L - p_k \leq \frac{(\sum_{i=1}^{n} p_i) - p_k}{m}
\]

hence

\[
L \leq \frac{(\sum_{i=1}^{n} p_i)}{m} + p_k \left(1 - \frac{1}{m}\right)
\]

\[
\leq \text{OPT} + \text{OPT} \left(1 - \frac{1}{m}\right)
\]

\[
= \text{OPT} \left(2 - \frac{1}{m}\right)
\]

where the third step follows from the two lower bounds on OPT.

The above analysis is tight, i.e., there exist instances where the greedy algorithm produces a schedule which has a makespan \((2 - 1/m)\) times the optimal. Consider the following instance: \(m(m - 1)\) jobs with unit processing time and a single job with processing time \(m\). Suppose the greedy algorithm schedules all the short jobs before the long job, then the makespan of the schedule obtained is \((2m - 1)\) while the optimal makespan is \(m\). Hence the algorithm gives a schedule which has makespan \(2 - 1/m\) times the optimal.

It may seem from the tight example above that an approximation ratio \(\alpha < (2 - 1/m)\) could be achieved if the jobs are sorted before processing, which indeed is the case. The following algorithm, due to [25], sorts the jobs in decreasing order of processing time prior to running Greedy Multiprocessor Scheduling algorithm.

```
Modified Greedy Multiprocessor Scheduling:
Sort the jobs in decreasing order of processing times
For i = 1 to n do
    Assign Job J_i to the machine with least current load
    Update load of the machine that receives job J_i
```

Graham [25] proved the following tight bound.

**Theorem 5.5.** Modified Greedy Multiprocessor Scheduling algorithm gives a \((4/3 - 1/3m)\)-approximation for the Multiprocessor Scheduling problem.

We will not prove the preceding theorem which requires some careful case analysis. Instead we will show how one can obtain an easier bound of 3/2 via the following claim.

**Claim 5.1.1.** Suppose \(p_i \geq p_j\) for all \(i > j\) and \(n > m\). Then, \(\text{OPT} \geq p_m + p_{m+1}\).
Proof. Since \( n > m \) and the processing times are sorted in decreasing order, some two of the \((m + 1)\) largest jobs must be scheduled on the same machine. Notice that the load of this machine is at least \( p_m + p_{m+1} \).

**Exercise 5.1.** Prove that Modified Greedy Multiprocessor Scheduling gives a \((3/2 - 1/2m)\)-approximation using the preceding claim and the other two lower bounds on OPT that we have seen already.

Before going to the description of a PTAS for Multiprocessor Scheduling problem, we discuss the case when the processing times of the jobs are bounded from above.

**Claim 5.1.2.** If \( p_i \leq \varepsilon \cdot \text{OPT}, \forall i \), then Modified Greedy Multiprocessor Scheduling gives a \((1 + \varepsilon)\)-approximation.

5.1.3 A PTAS for Multi-Processor Scheduling

We will now give a PTAS for the problem of scheduling jobs on identical machines. We would like to use the same set of ideas that were used for the Knapsack problem (see Lecture 4): that is, given an explicit time \( T \) we would like to round the job lengths and use dynamic programming to see if they will fit within time \( T \). Then the unrounded job lengths should fit within time \( T(1 + \varepsilon) \).

**Big Jobs, Small Jobs and Rounding Big Jobs:** For the discussion that follows, we assume that all the processing times have been scaled so that \( \text{OPT} = 1 \) and hence, \( p_{max} \leq 1 \).

Given all the jobs, we partition the jobs into two sets: Big jobs and Small jobs. We call a job \( J_i \) “big” if \( p_i \geq \varepsilon \). Let \( B \) and \( S \) denote the set of big jobs and small jobs respectively, i.e., \( B = \{ J_i : p_i \geq \varepsilon \} \) and \( S = \{ J_i : p_i < \varepsilon \} \). The significance of such a partition is that once we pack the jobs in set \( B \), the jobs in set \( S \) can be greedily packed using list scheduling.

**Claim 5.1.3.** If there is an assignment of jobs in \( B \) to the machines with load \( L \), then greedily scheduling jobs of \( S \) on top gives a schedule of value no greater than \( \max \{ L, (1 + \varepsilon) \cdot \text{OPT} \} \).

**Proof.** Consider scheduling the jobs in \( S \) after all the jobs in \( B \) have been scheduled (with load \( L \)). If all of these jobs in \( S \) finish processing by time \( L \), the total load is clearly no greater than \( L \).

If the jobs in \( S \) cannot be scheduled within time \( L \), consider the last job to finish (after scheduling the small jobs). Suppose this job starts at time \( T' \). All the machines must have been fully loaded up to \( T' \), which gives \( \text{OPT} \geq T' \). Since, for all jobs in \( S \), we have \( p_i \leq \varepsilon \cdot \text{OPT} \), this job finishes at \( T' + \varepsilon \cdot \text{OPT} \). Hence, the schedule can be no more than \( T' + \varepsilon \cdot \text{OPT} \leq (1 + \varepsilon) \cdot \text{OPT} \), settling the claim. ■
Scheduling Big Jobs: We concentrate on scheduling the jobs in $\mathcal{B}$. We round the sizes of all jobs in $\mathcal{B}$ using geometrically increasing interval sizes using the following procedure:

<table>
<thead>
<tr>
<th>Rounding Jobs:</th>
</tr>
</thead>
<tbody>
<tr>
<td>For each big job $i$ do</td>
</tr>
<tr>
<td>If $p_i \in (e(1+\epsilon)^i, e(1+\epsilon)^{i+1}]$</td>
</tr>
<tr>
<td>Set $p_i = e(1+\epsilon)^{i+1}$</td>
</tr>
</tbody>
</table>

Let $\mathcal{B}'$ be the set of new jobs.

Claim 5.1.4. If jobs in $\mathcal{B}$ can be scheduled with load 1, then the rounded jobs in $\mathcal{B}'$ can be scheduled with load $(1+\epsilon)$.

Claim 5.1.5. The number of distinct big job sizes after rounding is $O\left(\frac{1}{\epsilon}\right)$.

Proof. Notice that due to scaling, we have $p_i \leq 1$ for all jobs $J_i$. Since the job sizes are between $\epsilon$ and 1 the number of geometric powers of $(1+\epsilon)$ required is $k$ where

\[ e(1+\epsilon)^k \leq 1 \]

\[ \Rightarrow k \leq \ln(\frac{1}{\epsilon}) = O\left(\frac{1}{\epsilon}\right). \]

Lemma 5.1. If the number of distinct job sizes is $k$, then there is an exact algorithm that returns the schedule (if there is one) and runs in time $O(n^k)$.

Proof. Use Dynamic Programming.

Corollary 5.6. Big jobs can be scheduled (if possible) with load $(1+\epsilon)$ in time $n^{O\left(\frac{\ln(1/\epsilon)}{\epsilon}\right)}$.

Once we have scheduled the jobs in $\mathcal{B}$, using Claim 5.1.3, we can pack small items using greedy list scheduling on top of them. The overall algorithm is then given as:

<table>
<thead>
<tr>
<th>PTAS Multiprocessor Scheduling:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Guess OPT</td>
</tr>
<tr>
<td>2. Define $\mathcal{B}$ and $S$</td>
</tr>
<tr>
<td>3. Round $\mathcal{B}$ to $\mathcal{B}'$</td>
</tr>
<tr>
<td>4. If jobs in $\mathcal{B}'$ can be scheduled in $(1+\epsilon)$ OPT</td>
</tr>
<tr>
<td>Greedily pack $S$ on top</td>
</tr>
<tr>
<td>Else</td>
</tr>
<tr>
<td>Modify the guess and Repeat.</td>
</tr>
</tbody>
</table>

In the following subsection, we comment on the guessing process.
Guessing: We define a \((1 + \varepsilon)\)-relaxed decision procedure:

**Definition 5.7.** Given \(\varepsilon > 0\) and a time \(T\), a \((1 + \varepsilon)\)-relaxed decision procedure returns:

- Output a schedule (if there is one) with makespan load \((1 + \varepsilon) \cdot T\) or
- Output correctly that there is no schedule with makespan \(T\).

Define

\[
L = \max \left\{ \max_j p_j, \frac{1}{m} \sum_j p_j \right\}
\]

\(L\) is a lower bound on \(OPT\) as we saw earlier. Furthermore, an upper bound on \(OPT\) is given by the Greedy Multiprocessor Scheduling algorithm, which is \(2L\). Consider running the decision procedure with guess \(L + i\varepsilon L\) for each integer \(i \in \lceil \frac{2}{\varepsilon} \rceil\). We will choose the schedule with the best makespan among all the successful runs. If \(L^*\) is the optimum load then the algorithm will try the decision procedure with \(L^* + \varepsilon L \leq (1 + \varepsilon)L^*\). For this guess we are guaranteed a solution and the decision procedure will succeed in outputting a schedule with load \((1 + \varepsilon)(1 + \varepsilon)L^* \leq (1 + 3\varepsilon)L^*\) for sufficiently small \(\varepsilon\). We run the decision procedure \(O(1/\varepsilon)\) times. This gives us the desired PTAS.

**Remark 5.1.** A PTAS indicates that the problem can approximated arbitrarily well in polynomial time. However, a running time of the form \(n^{f(\varepsilon)}\) is typically not very interesting. We have seen that an FPTAS is ruled out for the makespan minimization problem. However, it does admit what is now called an Efficient PTAS (EPTAS) whose running time is \(2^{O(1/\varepsilon^2 \log(1/\varepsilon)^3)} + \text{poly}(n)\). See [36].

### 5.1.4 Section Notes

Multiprocessor Scheduling is NP-hard as we can reduce 2-Partition to Multiprocessor Scheduling on two machines. Note that this reduction only proves that Multiprocessor Scheduling is weakly NP-hard. When \(m\) is a fixed constant Horowitz and Sahni [33] give an FPTAS. However Multiprocessor Scheduling problem is strongly NP-hard when \(m\) is part of the input (by a reduction from 3-Partition [23]). Thus, there can not exist an FPTAS for the Multiprocessor Scheduling problem in general, unless \(P = NP\). However, Hochbaum and Shmoys [31] gave a PTAS which is the one we described. EPTASes have been developed for several problems and a key technique is the use of integer linear programming solvers with a small number of variables.
5.2 Bin Packing

5.2.1 Problem Description

In the Bin Packing problem, we are given a set of \( n \) items \( \{1, 2, \ldots, n\} \). Item \( i \) has size \( s_i \in (0, 1] \). The goal is to find a minimum number of bins of capacity 1 into which all the items can be packed.

One could also formulate the problem as partitioning \( \{1, 2, \ldots, n\} \) into \( k \) sets \( B_1, B_2, \ldots, B_k \) such that \( \sum_{i \in B_j} s_i \leq 1 \) and \( k \) is minimum.

5.2.2 Greedy Approaches

Consider the following greedy algorithm for bin packing:

<table>
<thead>
<tr>
<th>Greedy Bin Packing:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order items in some way</td>
</tr>
<tr>
<td>For ( i = 1 ) to ( n )</td>
</tr>
<tr>
<td>If item ( i ) can be packed in some open bin</td>
</tr>
<tr>
<td>Pack it</td>
</tr>
<tr>
<td>Else</td>
</tr>
<tr>
<td>Open a new bin and pack ( i ) in the new bin</td>
</tr>
</tbody>
</table>

In Greedy Bin Packing algorithm, a new bin is opened only if the item cannot be packed in any of the already opened bins. However, there might be several opened bins in which the item \( i \) could be packed. Several rules could be formulated in such a scenario:

- **First Fit**: Pack item in the earliest opened bin
- **Last Fit**: Pack item in the last opened bin
- **Best Fit**: Pack item in the bin that would have least amount of space left after packing the item
- **Worst Fit**: Pack item in the bin that would have most amount of space left after packing the item

Irrespective of what strategy is chosen to pack an item in the opened bins, one could get the following result:

**Theorem 5.8.** Any greedy rule yields a 2-approximation.

**Observation 5.9.** \( \text{OPT} \geq \sum_i s_i \).

We call a bin \( \alpha \)-full if items occupy space at most \( \alpha \).
**Claim 5.2.1.** Greedy has at most 1 bin that is $\frac{1}{2}$-full.

*Proof.* For the sake of contradiction, assume that there are two bins $B_i$ and $B_j$ that are $\frac{1}{2}$-full. WLOG, assume that Greedy Bin Packing algorithm opened bin $B_i$ before $B_j$. Then, the first item that the algorithm packed into $B_j$ must be of size at most $\frac{1}{2}$. However, this item could have been packed into $B_i$ since $B_i$ is $\frac{1}{2}$-full. This is a contradiction to the fact that Greedy Bin Packing algorithm opens a new bin if and only if the item can not be packed in any of the opened bins. ■

*Proof of Theorem 5.8.* Let $m$ be the number of bins opened by Greedy Bin Packing algorithm. From Claim 5.2.1, we have:

$$\sum_i s_i > \frac{m - 1}{2}$$

Using the observation that $\text{OPT} \geq \sum_i s_i$, we get:

$$\text{OPT} > \frac{m - 1}{2}$$

which gives us:

$$m < 2 \cdot \text{OPT} + 1$$

$$\Rightarrow m \leq 2 \cdot \text{OPT}$$

■

**5.2.3 (Asymptotic) PTAS for Bin Packing**

A natural question follows the discussion above: Can Bin Packing have a PTAS? In this subsection, we settle this question in negative. In particular, we give a reduction from an NP-complete problem to the Bin Packing problem and show that a PTAS for the Bin Packing problem will give us an exact solution for the NP-complete problem in polynomial time. We consider the Partition problem:

In the Partition problem, we are given a set of items $\{1, 2, \ldots, n\}$. Item $i$ has a size $s_i$. The goal is to partition the $\{1, 2, \ldots, n\}$ into two sets $\mathcal{A}$ and $\mathcal{B}$ such that $\sum_{i \in \mathcal{A}} s_i = \sum_{j \in \mathcal{B}} s_j$.

**Claim 5.2.2.** If Bin Packing has a $(\frac{3}{2} - \epsilon)$-approximation for any $\epsilon > 0$, the Partition problem can be solved exactly in polynomial time.
Proof. Given an instance \( I \) of the Partition problem, we construct an instance of the Bin Packing problem as follows: Scale the size of the items such that \( \sum_i s_i = 2 \). Consider the scaled sizes of the items as an instance \( I' \) of the Bin Packing problem. If all items of \( I' \) can be packed in 2 bins, then we have an “yes” answer to \( I \). Otherwise, the items of \( I' \) need 3 bins and the answer to \( I \) is “no”.

\( \text{OPT} \) for \( I' \) is 2 or 3. Hence, if there is a \((\frac{3}{2} - \varepsilon)\)-approximation algorithm for the Bin Packing problem, we can determine the value of \( \text{OPT} \) which in turn implies that we can solve \( I \). Thus, there can not exist a \((\frac{3}{2} - \varepsilon)\)-approximation algorithm for the Bin Packing problem, unless \( P = NP \). ■

Recall the scaling property where we discussed why many optimization problems do not admit additive approximations. We notice that the Bin Packing problem does not have the scaling property. Hence it may be possible to find an additive approximation algorithms. We state some of the results in this context:

**Theorem 5.10** (Johnson ’74 [39]). There exists a polynomial time algorithm such \( A_I \) such that:

\[
A_I(I) \leq \frac{11}{9} \text{OPT}(I) + 4
\]

for all instances \( I \) of the Bin Packing problem.

**Theorem 5.11** (de la Vega, Lueker ’81 [16]). For any fixed \( \varepsilon > 0 \) there exists a polynomial time algorithm such \( A_{FL} \) such that:

\[
A_{FL}(I) \leq (1 + \varepsilon) \text{OPT}(I) + 1
\]

for all instances \( I \) of the Bin Packing problem.

**Theorem 5.12** (Karmarkar, Karp ’82 [42]). There exists a polynomial time algorithm such \( A_{KK} \) such that:

\[
A_{KK}(I) \leq \text{OPT}(I) + O(\log^2(\text{OPT}(I)))
\]

for all instances \( I \) of the Bin Packing problem.

This has been improved recently.

**Theorem 5.13** (Hoberg and Rothvoss 2017 [30]). There exists a polynomial time algorithm such \( A_{HT} \) such that:

\[
A_{KK}(I) \leq \text{OPT}(I) + O(\log(\text{OPT}(I)))
\]

for all instances \( I \) of the Bin Packing problem.

A major open problem is the following.

**Open Question 5.14.** Is there a polynomial-time algorithm \( A \) such that \( A(I) \leq \text{OPT}(I) + c \), for some fixed constant \( c \)? In particular is \( c = 1 \)?

**Exercise 5.2.** Show that First Fit greedy rule yields a \( \frac{3}{2} \) \( \text{OPT} \) +1-approximation.
5.2.4 Asymptotic PTAS for Bin Packing

A recurring theme in last two lectures has been the rounding of jobs/tasks/items. To construct an asymptotic PTAS for Bin Packing problem, we use the same set of ideas with simple in retrospect but non-obvious modifications. In particular, we divide the set of items into big and small items and concentrate on packing the big items first. We show that such a technique results in an asymptotic PTAS for the Bin Packing problem.

Consider the set of items, \( s_1, s_2, \ldots, s_n \). We divide the items into two sets, \( \mathcal{B} = \{ i : s_i \geq \epsilon \} \) and \( \mathcal{S} = \{ j : s_j < \epsilon \} \). Similar to the MultiProcessor Scheduling problem, where we rounded up the processing times of the jobs, we round up the sizes of the items in the Bin Packing problem. Again, we concentrate only on the items in \( \mathcal{B} \). Let \( n' = |\mathcal{B}| \) be the number of big items.

We observed earlier that \( \text{OPT} \geq \sum_i s_i \) and hence \( \text{OPT} \geq \epsilon \cdot n' \) by just considering the big items.

**Claim 5.2.3.** Suppose \( n' > 4/\epsilon^2 \). Then \( \text{OPT} \geq 4/\epsilon \).

If there are very few big items one can solve the problem by brute force.

**Claim 5.2.4.** Suppose \( n' < 4/\epsilon^2 \). An optimal solution for the Bin Packing problem can be computed in \( 2^{O(1/\epsilon^4)} \) time.

**Proof Sketch.** If the number of big items is small, one can find the optimal solution using brute force search.

The following gives a procedure to round up the items in \( \mathcal{B} \):

<table>
<thead>
<tr>
<th>Rounding Item Sizes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort the items such that ( s_1 \geq s_2 \geq \cdots \geq s_n' ).</td>
</tr>
<tr>
<td>Group items in ( k = 2/\epsilon^2 ) groups ( \mathcal{B}_1, \ldots, \mathcal{B}_k ) such that each group has ( \lfloor n'/k \rfloor ) items.</td>
</tr>
<tr>
<td>Round the size of all the items in group ( \mathcal{B}<em>i ) to the size of the smallest item in ( \mathcal{B}</em>{i-1} ).</td>
</tr>
</tbody>
</table>

**Lemma 5.2.** Consider the restriction of the bin packing problem to instances in which the number of distinct item sizes is \( k \). There is an \( n^{O(k)} \)-time algorithm that outputs the optimum solution.

**Proof Sketch.** Use Dynamic Programming.

**Claim 5.2.5.** The items in \( \mathcal{B} \) can be packed in \( \text{OPT} + |\mathcal{B}_1| \) bins in time \( n^{O(1/\epsilon^2)} \).

**Proof.** Using Rounding Item Sizes, we have restricted all items but those in \( \mathcal{B}_1 \) to have one of the \( k - 1 \) distinct sizes. Using lemma 5.2, these items can be packed efficiently in OPT. Furthermore, the items in \( \mathcal{B}_1 \) can always be packed in \( |\mathcal{B}_1| \) bins (one per bin). Hence, the total number of bins is \( \text{OPT} + |\mathcal{B}_1| \).

The running time of the algorithm follows since \( k = O(1/\epsilon^2) \).
Lemma 5.3. Let $\epsilon > 0$ be fixed. Consider the restriction of the bin packing problem to instances in which each item is of size at least $\epsilon$. There is a polynomial time algorithm that solves this restricted problem within a factor of $(1 + \epsilon)$.

Proof. Using Claim 5.2.5, we can pack $\mathcal{B}$ in $\text{OPT} + |\mathcal{B}_1|$ bins. Recall that $|\mathcal{B}_1| = \lceil n'/k \rceil \leq e^2 \cdot n'/2 \leq \epsilon \cdot \text{OPT}/8$ where, we have used Claim 5.2.3 to reach the final expression.

Theorem 5.15. For any $\epsilon$, $0 < \epsilon < 1/2$, there is an algorithm $\mathcal{A}_\epsilon$ that runs in time polynomial in $n$ and finds a packing using at most $(1 + 2\epsilon)\text{OPT} + 1$ bins.

Proof. Assume that the number of bins used to pack items in $\mathcal{B}$ is $m$ and the total number of bins used after packing items in $\mathcal{S}$ is $m'$. Clearly

$$m' \leq \max \left\{ m, \left\lceil \frac{\sum_i s_i}{(1 - \epsilon)} \right\rceil \right\}$$

since at most one bin must be $(1 - \epsilon)$ full using an argument in Greedy Bin Packing. Furthermore,

$$\left\lceil \frac{\sum_i s_i}{(1 - \epsilon)} \right\rceil \leq \left( \sum_i s_i \right) (1 + 2\epsilon) + 1$$

for $\epsilon < 1/2$. This gives the required expression.

The algorithm is summarized below:

<table>
<thead>
<tr>
<th>Asymptotic PTAS Bin Packing:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split the items in $\mathcal{B}$ (big items) and $\mathcal{S}$ (small items)</td>
</tr>
<tr>
<td>Round the sizes of the items in $\mathcal{B}$ to obtain constant number of item sizes</td>
</tr>
<tr>
<td>Find optimal packing for items with rounded sizes</td>
</tr>
<tr>
<td>Use this packing for original items in $\mathcal{B}$</td>
</tr>
<tr>
<td>Pack items in $\mathcal{S}$ using Greedy Bin Packing.</td>
</tr>
</tbody>
</table>

5.2.5 Section Notes

An excellent but perhaps somewhat dated survey on approximation algorithms for Bin Packing problem is [40]. See [30] for some pointers to more recent work. There has also been substantial recent work on various generalizations to multiple dimensions.
Chapter 6

Unrelated Machine Scheduling and Generalized Assignment

This chapter is based on notes first scried by Alina Ene.

6.1 Scheduling on Unrelated Parallel Machines

We have a set $J$ of $n$ jobs, and a set $M$ of $m$ machines. The processing time of job $i$ is $p_{ij}$ on machine $j$. Let $f : J \rightarrow M$ be a function that assigns each job to exactly one machine. The makespan of $f$ is $\max_{1 \leq j \leq m} \sum_{i : f(i) = j} p_{ij}$, where $\sum_{i : f(i) = j} p_{ij}$ is the total processing time of the jobs that are assigned to machine $j$. In the Scheduling on Unrelated Parallel Machines problem, the goal is to find an assignment of jobs to machines of minimum makespan.

We can write an LP for the problem that is very similar to the routing LP from the previous lecture. For each job $i$ and each machine $j$, we have a variable $x_{ij}$ that denotes whether job $i$ is assigned to machine $j$. We also have a variable $\lambda$ for the makespan. We have a constraint for each job that ensures that the job is assigned to some machine, and we have a constraint for each machine that ensures that the total processing time of jobs assigned to the machines is at most the makespan $\lambda$. 
The above LP is very natural, but unfortunately it has unbounded integrality gap. Suppose that we have a single job that has processing time $T$ on each of the machines. Clearly, the optimal schedule has makespan $T$. However, the LP can schedule the job to the extend of $1/m$ on each of the machines, i.e., it can set $x_{1j} = 1/m$ for all $j$, and the makespan of the resulting fractional schedule is only $T/m$.

To overcome this difficulty, we modify the LP slightly. Suppose we knew that the makespan of the optimal solution is equal to $\lambda$, where $\lambda$ is some fixed number. If the processing time $p_{ij}$ of job $i$ on machine $j$ is greater than $\lambda$, job $i$ is not scheduled on machine $j$, and we can strengthen the LP by setting $x_{ij}$ to 0 or equivalently, by removing the variable. More precisely, let $S_\lambda = \{(i, j) \mid i \in J, j \in M, p_{ij} \leq \lambda\}$. Given a value $\lambda$, we can write the following LP for the problem.

$$\text{LP}(\lambda)$$

\[
\begin{align*}
\sum_{j: (i,j) \in S_\lambda} x_{ij} &= 1 & \forall i \in J \\
\sum_{i: (i,j) \in S_\lambda} x_{ij}p_{ij} &\leq \lambda & \forall j \in M \\
x_{ij} &\geq 0 & \forall (i, j) \in S_\lambda
\end{align*}
\]

Note that the LP above does not have an objective function. In the following, we are only interested in whether the LP is feasible, i.e, whether there is an assignment that satisfies all the constraints. Also, we can think of $\lambda$ as a parameter and $\text{LP}(\lambda)$ as a family of LPs, one for each value of the parameter. A useful observation is that, if $\lambda$ is a lower bound on the makespan of the optimal schedule, $\text{LP}(\lambda)$ is feasible and it is a valid relaxation for the Scheduling on Unrelated Parallel Machines problem.

**Lemma 6.1.** Let $\lambda^*$ be the minimum value of the parameter $\lambda$ such that $\text{LP}(\lambda)$ is feasible. We can find $\lambda^*$ in polynomial time.
Proof. For any fixed value of $\lambda$, we can check whether $\text{LP}(\lambda)$ is feasible using a polynomial-time algorithm for solving LPs. Thus we can find $\lambda^*$ using binary search starting with the interval $[0, \sum_{i,j} p_{ij}]$.

In the following, we will show how to round a solution to $\text{LP}(\lambda^*)$ in order to get a schedule with makespan at most $2\lambda^*$. As we will see shortly, it will help to round a solution to $\text{LP}(\lambda^*)$ that is a vertex solution.

Let $x$ be a vertex solution to $\text{LP}(\lambda^*)$. Let $G$ be a bipartite graph on the vertex set $J \cup M$ that has an edge $ij$ for each variable $x_{ij} \neq 0$. We say that job $i$ is fractionally set if $x_{ij} \in (0,1)$ for some $j$. Let $F$ be the set of all jobs that are fractionally set, and let $H$ be a bipartite graph on the vertex set $F \cup M$ that has an edge $ij$ for each variable $x_{ij} \in (0,1)$; note that $H$ is the induced subgraph of $G$ on $F \cup M$. As shown in Lemma 6.2, the graph $H$ has a matching that matches every job in $F$ to a machine, and we will use such a matching in the rounding algorithm.

**Lemma 6.2.** The graph $G$ has a matching that matches every job in $F$ to a machine.

We are now ready to give the rounding algorithm.

<table>
<thead>
<tr>
<th>SUPM-Rounding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find $\lambda^*$</td>
</tr>
<tr>
<td>Find a vertex solution $x$ to $\text{LP}(\lambda^*)$</td>
</tr>
<tr>
<td>For each $i$ and $j$ such that $x_{ij} = 1$, assign job $i$ to machine $j$</td>
</tr>
<tr>
<td>Construct the graph $H$</td>
</tr>
<tr>
<td>Find a maximum matching $M$ in $H$</td>
</tr>
<tr>
<td>Assign the fractionally set jobs according to the matching $M$</td>
</tr>
</tbody>
</table>

**Theorem 6.1.** Consider the assignment constructed by SUPM-Rounding. Each job is assigned to a machine, and the makespan of the schedule is at most $2\lambda^*$.

Proof. By Lemma 6.2, the matching $M$ matches every fractionally set job to a machine and therefore all of the jobs are assigned. After assigning all of the integrally set jobs, the makespan (of the partial schedule) is at most $\lambda^*$. Since $M$ is a matching, each machine receives at most one additional job. Let $i$ be a fractionally set job, and suppose that $i$ is matched (in $M$) to machine $j$. Since the pair $(i,j)$ is in $S^\lambda$, the processing time $p_{ij}$ is at most $\lambda^*$, and therefore the total processing time of machine $j$ increases by at most $\lambda$ after assigning the fractionally set jobs. Therefore the makespan of the final schedule is at most $2\lambda^*$.

**Exercise 6.1.** Give an example that shows that Theorem 6.1 is tight. That is, give an instance and a vertex solution such that the makespan of the schedule SUPM-Rounding is at least $(2 - o(1))\lambda^*$. 

Since \( \lambda^* \) is a lower bound on the makespan of the optimal schedule, we get the following corollary.

**Corollary 6.2.** SLPM-Rounding achieves a 2-approximation.

Now we turn our attention to Lemma 6.2 and some other properties of vertex solutions to \( \text{LP}(\lambda) \). The following can be derived from the rank lemma which is described in Chapter A. Here we give a self-contained proof.

**Lemma 6.3.** If \( \text{LP}(\lambda) \) is feasible, any vertex solution has at most \( m + n \) non-zero variables and it sets at least \( n - m \) of the jobs integrally.

**Proof.** Let \( x \) be a vertex solution to \( \text{LP}(\lambda) \). Let \( r \) denote the number of pairs in \( S_\lambda \). Note that \( \text{LP}(\lambda) \) has \( r \) variables, one for each pair \( (i, j) \in S_\lambda \). If \( x \) is a vertex solution, it satisfies \( r \) of the constraints of \( \text{LP}(\lambda) \) with equality. The first set of constraints consists of \( m \) constraints, and the second set of constraints consists of \( n \) constraints. Therefore at least \( r - (m + n) \) of the tight constraints are from the third set of constraints, i.e., at least \( r - (m + n) \) of the variables are set to zero.

We say that job \( i \) is set fractionally if \( x_{ij} \in (0, 1) \) for some \( j \); job \( i \) is set integrally if \( x_{ij} \in \{0, 1\} \) for all \( j \). Let \( I \) and \( F \) be the set of jobs that are set integrally and fractionally (respectively). Clearly, \( |I| + |F| = n \). Any job \( i \) that is fractionally set is assigned (fractionally) to at least two machines, i.e., there exist \( j \neq \ell \) such that \( x_{ij} \in (0, 1) \) and \( x_{i\ell} \in (0, 1) \). Therefore there are at least \( 2|F| \) distinct non-zero variables corresponding to jobs that are fractionally set. Additionally, for each job \( i \) that is integrally set, there is a variable \( x_{ij} \) that is non-zero. Thus the number of non-zero variables is at least \( |I| + 2|F| \). Hence \( |I| + |F| = n \) and \( |I| + 2|F| \leq m + n \), which give us that \( |I| \) is at least \( n - m \). ■

**Definition 6.3.** A connected graph is a **pseudo-tree** if the number of edges it most the number of vertices plus one. A graph is a **pseudo-forest** if each of its connected components is a pseudo-tree.

**Lemma 6.4.** The graph \( G \) is a pseudo-forest.

**Proof.** Let \( C \) be a connected component of \( G \). We restrict \( \text{LP}(\lambda) \) and \( x \) to the jobs and machines in \( C \) to get \( \text{LP}'(\lambda) \) and \( x' \). Note that \( x' \) is a feasible solution to \( \text{LP}'(\lambda) \). Additionally, \( x' \) is a vertex solution to \( \text{LP}'(\lambda) \). If not, \( x' \) is a convex combination of two feasible solutions \( x_1' \) and \( x_2' \) to \( \text{LP}'(\lambda) \). We can extend \( x_1' \) and \( x_2' \) to two solutions \( x_1 \) and \( x_2 \) to \( \text{LP}(\lambda) \) using the entries of \( x \) that are not in \( x' \). By construction, \( x_1 \) and \( x_2 \) are feasible solutions to \( \text{LP}(\lambda) \). Additionally, \( x \) is a convex combination of \( x_1 \) and \( x_2 \), which contradicts the fact that \( x \) is a vertex solution. Thus \( x' \) is a vertex solution to \( \text{LP}'(\lambda) \) and, by Lemma 6.3, \( x' \) has at most \( n' + m' \) non-zero variables, where \( n' \) and \( m' \) are the number of jobs and machines in \( C \). Thus \( C \) has \( n' + m' \) vertices and at most \( n' + m' \) edges, and therefore it is a pseudo-tree. ■
Proof. of Lemma 6.2 Note that each job that is integrally set has degree one in \( G \). We remove each integrally set job from \( G \); note that the resulting graph is \( H \). Since we removed an equal number of vertices and edges from \( G \), it follows that \( H \) is a pseudo-forest as well. Now we construct a matching \( M \) as follows.

Note that every job vertex has degree at least two, since the job is fractionally assigned to at least two machines. Thus all of the leaves (degree-one vertices) of \( H \) are machines. While \( H \) has at least one leaf, we add the edge incident to the leaf to the matching and we remove both of its endpoints from the graph. If \( H \) does not have any leaves, \( H \) is a collection of vertex-disjoint cycles, since it is a pseudo-forest. Moreover, each cycle has even length, since \( H \) is bipartite. We construct a perfect matching for each cycle (by taking alternate edges), and we add it to our matching.

\[ \square \]

Exercise 6.2. (Exercise 17.1 in [65]) Give a proof of Lemma 6.2 using Hall’s theorem.

\[ \text{Remark 6.1.} \] We could allow each machine \( M_j \) to have a different capacity \( c_j \) which is more natural in certain settings. However, since \( p_{ij} \) values are allowed to depend on \( j \) we can scale them to ensure that \( c_j = \lambda \) for every \( j \) without loss of generality.

In the following, we will show that, if there is an assignment of cost \( C \) and makespan at most \( \lambda \), then we can construct a schedule of cost at most \( C \) and makespan at most \( 2\lambda \). In fact the assignment will have a stronger property that the load on a a machine exceeds \( \lambda \) due to at most one job.

As before, we let \( S_\lambda \) denote the set of all pairs \((i, j)\) such that \( p_{ij} \leq \lambda \). We can generalize the relaxation \( \text{LP}(\lambda) \) from the previous section to the following LP.

6.2 Generalized Assignment Problem

The Generalized Assignment problem is a generalization of the Scheduling on Unrelated Parallel Machines problem in which there are costs associated with each job-machine pair, in addition to a processing time. More precisely, we have a set \( J \) of \( n \) jobs, a set \( M \) of \( m \) machines, and a target \( \lambda \). The processing time of job \( i \) is \( p_{ij} \) on machine \( j \), and the cost of assigning job \( i \) to machine \( j \) is \( c_{ij} \). Let \( f : J \rightarrow M \) be a function that assigns each job to exactly one machine. The assignment \( f \) is feasible if its makespan is at most \( \lambda \) (recall that \( \lambda \) is part of the input), and its cost is \( \sum_i c_{i,f(i)} \). In the Generalized Assignment problem, the goal is to construct a minimum cost assignment \( f \) that is feasible, provided that there is a feasible assignment.

Remark 6.1. We could allow each machine \( M_j \) to have a different capacity \( c_j \) which is more natural in certain settings. However, since \( p_{ij} \) values are allowed to depend on \( j \) we can scale them to ensure that \( c_j = \lambda \) for every \( j \) without loss of generality.

In the following, we will show that, if there is an assignment of cost \( C \) and makespan at most \( \lambda \), then we can construct a schedule of cost at most \( C \) and makespan at most \( 2\lambda \). In fact the assignment will have a stronger property that the load on a a machine exceeds \( \lambda \) due to at most one job.

As before, we let \( S_\lambda \) denote the set of all pairs \((i, j)\) such that \( p_{ij} \leq \lambda \). We can generalize the relaxation \( \text{LP}(\lambda) \) from the previous section to the following LP.
Since we also need to preserve the costs, we can no longer use the previous rounding; in fact, it is easy to see that the previous rounding is arbitrarily bad for the Generalized Assignment problem. However, we will still look for a matching, but in a slightly different graph.

But before we give the rounding algorithm for the Generalized Assignment problem, we take a small detour into the problem of finding a minimum-cost matching in a bipartite graph. In the Minimum Cost Bipartite Matching problem, we are given a bipartite graph \( B = (V_1 \cup V_2, E) \) with costs \( c_e \) on the edges, and we want to construct a minimum cost matching \( M \) that matches every vertex in \( V_1 \), if there is such a matching. For each vertex \( v \), let \( \delta(v) \) be the set of all edges incident to \( v \). We can write the following LP for the problem.

\[
\text{BipartiteMatching}(B) \quad \min \sum_{e \in E(B)} c_e y_e
\]

subject to
\[
\begin{align*}
  \sum_{e \in \delta(v)} y_e &= 1 & \forall v & \in V_1 \\
  \sum_{e \in \delta(v)} y_e &\leq 1 & \forall v & \in V_2 \\
  y_e &\geq 0 & \forall e & \in E(B)
\end{align*}
\]

The following is well-known in combinatorial optimization [58].

**Theorem 6.4.** For any bipartite graph \( B \), any vertex solution to \text{BipartiteMatching}(B)\( is an integer solution. Moreover, given a feasible fractional solution, we can find in polynomial time a feasible solution \( z \) such that \( z \) is integral and

\[
\sum_{e \in E(B)} c_e z_e \leq \sum_{e \in E(B)} c_e y_e.
\]
In the rest of the section we give two different proofs that establish our claimed result. One is based on the first work that gave this result [59], and the other is based on iterative rounding [46].

6.2.1 Shmoys-Tardos Rounding

Let $x$ be an optimal vertex solution to GAP-LP. As before, we want to construct a graph $G$ that has a matching $M$ that matches all jobs. The graph $G$ will now have costs on its edges and we want a matching of cost at most $C$. Recall that for Scheduling on Unrelated Parallel Machines we defined a bipartite graph on the vertex set $J \cup M$ that has an edge $ij$ for every variable $x_{ij}$ that is non-zero. We can construct the same graph for Generalized Assignment, and we can assign a cost $c_{ij}$ to each edge $ij$. If the solution $x$ was actually a fractional matching — that is, if $x$ was a feasible solution to BipartiteMatching($G$) — Theorem ?? would give us the desired matching. The solution $x$ satisfies the constraints corresponding to vertices $v \in J$, but it does not necessarily satisfy the constraints corresponding vertices $v \in M$, since a machine can be assigned more than one job. To get around this difficulty, we will introduce several nodes representing the same machine, and we will use $x$ to construct a fractional matching for the resulting graph.

The fractional solution $x$ assigns $\sum_{i \in J} x_{ij}$ jobs to machine $j$; let $k_j = \lceil \sum_{i \in J} x_{ij} \rceil$. We construct a bipartite graph $G$ as follows. For each job $i$, we have a node $i$. For each machine $j$, we have $k_j$ nodes $(j, 1), \cdots, (j, k_j)$. We can think of the nodes $(j, 1), \cdots, (j, k_j)$ as slots on machine $j$. Since now we have multiple slots on each of the machines, we need a fractional assignment $y$ that assigns a job to slots on the machines. More precisely, $y$ has an entry $y_{i,(j,s)}$ for each job $i$ and each slot $(j, s)$ that represents the fraction of job $i$ that is assigned to the slot. We give the algorithm that constructs $y$ from $x$ below. Once we have the solution $y$, we add an edge between any job $i$ and any machine slot $(j, s)$ such that $y_{i,(j,s)}$ is non-zero. Additionally, we assign a cost $c_{i,(j,s)}$ to each edge $(i, (j, s))$ of $G$ that is equal to $c_{ij}$.
CHAPTER 6. UNRELATED MACHINE SCHEDULING AND GENERALIZED ASSIGNMENT

**GreedyPacking**(x)

\[
\begin{align*}
y &= 0 \quad \text{\langle initialize } y \text{ to 0\rangle} \\
s &= 1 \quad \text{\langle s is the current bin\rangle} \\
R &= 1 \quad \text{\langle R is the space available on bin } s\rangle \\
& \text{for } i = 1 \text{ to } h \\
& \langle \text{pack } x_{ij} \text{ into the bins}\rangle \\
& \text{if } x_{ij} \leq R \\
& \quad y_i,(j,s) = x_{ij} \\
& \quad R = R - x_{ij} \\
& \quad \text{if } R = 0 \\
& \quad \quad s = s + 1 \\
& \quad \quad R = 1 \\
& \text{else} \\
& \quad y_i,(j,s) = R \\
& \quad y_i,(j,s+1) = x_{ij} - R \quad \text{\langle pack } x_{ij} - R \text{ in the next bin\rangle} \\
& \quad R = 1 - y_i,(j,s+1) \\
& \quad s = s + 1 \\
& \text{return } y
\end{align*}
\]

![Figure 6.1: Constructing y from x.](image)

When we construct y, we consider each machine in turn. Let j be the current machine. Recall that we want to ensure that y assigns at most one job to each slot; as such, we will think of each slot on machine j as a bin with capacity 1. We
“pack” jobs into the bins greedily. We only consider jobs \( i \) such that \( p_{ij} \) is at most \( \lambda \); let \( h \) denote the number of such jobs. We assume without loss of generality that these are labeled as \( 1, 2, \ldots, h \), and \( p_{1j} \geq p_{2j} \geq \cdots \geq p_{hj} \). Informally, when we construct \( y \), we consider the jobs \( 1, 2, \ldots, h \) in this order. Additionally, we keep track of the bin that has not been filled and the amount of space \( B \) available on that bin. When we consider job \( i \), we try to pack \( x_{ij} \) into the current bin: if there is at least \( x_{ij} \) space available, i.e., \( x_{ij} \leq s \), we pack the entire amount into the current bin; otherwise, we pack as much as we can into the current bin, and we pack the rest into the next bin. (See Figure 1 for an example.)

**Lemma 6.5.** The solution \( y \) constructed by GreedyPack is a feasible solution to BipartiteMatching\((G)\). Moreover,

\[
\sum_{(i, (j, s)) \in E(G)} y_{i, (j, s)} c_{i, (j, s)} = \sum_{(i, j) \in S_1} x_{ij} c_{ij}.
\]

**Proof.** Note that, by construction, \( x_{ij} = \sum_{s=1}^{k_j} y_{i, (j, s)} \). Therefore, for any job \( i \), we have

\[
\sum_{(i, (j, s)) \in \delta(i)} y_{i, (j, s)} = \sum_{j: (i, j) \in S_1} \sum_{s=1}^{k_j} y_{i, (j, s)} = \sum_{j: (i, j) \in S_1} x_{ij} = 1
\]

Additionally, since we imposed a capacity of 1 on the bins associated with each slot, it follows that, for any slot \((j, s)\),

\[
\sum_{(i, (j, s)) \in \delta((j, s))} y_{i, (j, s)} \leq 1
\]

Therefore \( y \) is a feasible solution to BipartiteMatching\((G)\). Finally,

\[
\sum_{(i, (j, s)) \in E(G)} = \sum_{i=1}^{n} \sum_{j: (i, j) \in S_1} \sum_{s=1}^{k_j} y_{i, (j, s)} c_{ij} = \sum_{(i, j) \in S_1} x_{ij} c_{ij}
\]

Theorem 6.4 gives us the following corollary.

**Corollary 6.5.** The graph \( G \) has a matching \( M \) that matches every job and it has cost at most \( \sum_{(i, j) \in S_1} x_{ij} c_{ij} \). Moreover, we can find such a matching in polynomial time.
### GAP-Rounding

Let $x$ be an optimal solution to GAP-LP

\[ y = \text{GreedyPacking}(x) \]

construct the graph $G$

construct a matching $M$ in $G$ such that $M$ matches every job and the cost of $M$ is at most $\sum_{(i,j) \in S} x_{ij} c_{ij}$

for each edge $(i, (j,s)) \in M$

assign job $i$ to machine $j$

---

**Theorem 6.6.** Let $C = \sum_{(i,j) \in S} x_{ij} c_{ij}$. The schedule returned by GAP-Rounding has cost at most $C$ and makespan at most $2\lambda$.

**Proof.** By Corollary 6.5, the cost of the schedule is at most $C$. Therefore we only need to upper bound the makespan of the schedule.

Consider a machine $j$. For any slot $(j,s)$ on machine $j$, let

\[ q_{js} = \max_{i : y_{ij,(j,s)} > 0} p_{ij} \]

That is, $q_{js}$ is the maximum processing time of any pair $ij$ such that job $i$ is assigned (in $y$) to the slot $(j,s)$. It follows that the total processing time of the jobs that $M$ assigns to machine $j$ is at most $\sum_{s=1}^{k_f} q_{js}$.

Since GAP-LP has a variable $x_{ij}$ only for pairs $(i, j)$ such that $p_{ij}$ is at most $\lambda$, it follows that $q_{ij}$ is at most $\lambda$. Therefore we only need to show that $\sum_{s=2}^{k_f} q_{js}$ is at most $\lambda$ as well. Consider a slot $s$ on machine $j$ such that $s > 1$. Recall that we labeled the jobs that are relevant to machine $j$ — that is, jobs $i$ such that $p_{ij}$ is at most $\lambda$ — as $1, 2, \ldots, h$ such that $p_{1j} \geq p_{2j} \geq \cdots \geq p_{hj}$. Consider a job $\ell$ that is assigned to slot $s$. Since GreedyPacking considers jobs in non-increasing order according to their processing times, the processing time $p_{\ell j}$ of job $\ell$ is at most the processing time of any job assigned to the slot $s-1$. Therefore $p_{\ell j}$ is upper bounded by any convex combination of the processing times of the jobs that are assigned to the slot $s-1$. Since the slot $s-1$ is full, $\sum_{i} y_{i,(j,s-1)} = 1$ and thus $p_{\ell j}$ is at most $\sum_{i} y_{i,(j,s-1)} p_{ij}$. It follows that

\[ \sum_{s=2}^{k_f} q_{js} \leq \sum_{s=2}^{k_f} \sum_{i=1}^{k_i} y_{i,(j,s-1)} p_{ij} \leq \sum_{s=1}^{k_f} \sum_{i=1}^{k_i} y_{i,(j,s)} p_{ij} \]

By construction, $\sum_{s} y_{i,(j,s)} = x_{ij}$, and therefore

\[ \sum_{s=1}^{k_f} \sum_{i=1}^{k_i} y_{i,(j,s)} p_{ij} = \sum_{i=1}^{k_i} p_{ij} \sum_{s=1}^{s} y_{i,(j,s)} = \sum_{i=1}^{k_i} p_{ij} x_{ij} \]
Since $x$ is a feasible solution to the GAP-LP,

$$
\sum_{s=2}^{k_i} q_{js} \leq \sum_{i} p_{ij} x_{ij} \leq \lambda
$$

which completes the proof.

6.2.2 Iterative Rounding

Here we describe an alternative proof/algorithm that illustrates the powerful iterative rounding framework that initially came out of Jain’s seminal work [35] and has since then become a powerful technique in exact and approximation algorithms. We need some additional formalism to properly describe the algorithm. We will consider the input instance as being specified by a graph $G = (J \cup M, E)$ where an edge $ij \in E$ implies that $i$ is allowed to be schedule on $j$ and has size $p_{ij}$. It is also important to consider non-uniform capacities on the machines to allow for a recursive (or iterative) algorithm. Thus, we will assume that each machine $M_j$ has a capacity $b_j$. We will assume that $p_{ij} \leq b_j$ for all $ij \in E$ since we have already argued that this is a necessary constraint. In fact this assumption will not be needed until the very end. It is easy to generalize the LP relaxation for GAP-LP to handle non-uniform capacities and to handle the constraints specified by $G$. We will use the notation $\delta(i)$ and $\delta(j)$ to denote the edges incident to job $i$ and machine $j$ respectively.

**GAP-LP**

$$
\begin{align*}
\text{min} \quad & \sum_{(i,j) \in E} c_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{j: (i,j) \in \delta(i)} x_{ij} = 1 & \forall i \in J \\
& \sum_{i: (i,j) \in E} p_{ij} x_{ij} \leq b_j & \forall j \in M \\
& x_{ij} \geq 0 & \forall (i,j) \in E
\end{align*}
$$

To explain the underlying intuition for iterated rounding approach in the specific context of GAP, consider the situation where each machine $j$ has infinite capacity. In this case it is easy to find the minimum cost assignment. We simply assign each job $i$ to the machine $j = \arg \min_{j' \in \delta(i)} c_{ij'}$ which is the cheapest one that it is allowed to be assigned to. We also observe that if we drop all the capacity constraints from GAP-LP, and only leave the assignment constraints
\( \sum_{j} x_{ij} = 1 \) for each \( i \), then the optimum solution of this LP is the same as the one obtained by assigning each job to its cheapest allowed machine. Now consider another scenario. Suppose each machine \( j \) has in-degree at most \( k \) in \( G \) — that is, there are only \( k \) jobs that can ever be assigned to any machine \( j \). Now suppose we assign each job to its cheapest allowed machine. Clearly the cost is optimum. But what about the load? Since each machine had in-degree at most \( k \) we will load a machine \( j \) to at most \( \frac{1}{k} \). Thus, if \( k = 2 \) we will only violate the machine’s load by a factor of 2. However, this seems to be very restrictive assumption. Now consider a less restrictive scenario where there is one machine \( j \) such that its in-degree is at most 2. Then, in the LP relaxation, we can omit the constraint that limits its load since we are guaranteed that at most 2 jobs can be assigned to it (note that we still have the job assignment constraints which only allow a job to be assigned to machines according to the edges of \( G \)). Omitting constraints in an iterative fashion by taking advantage of sparsity in the basic feasible solution is the key idea.

To allow dropping of constraints we need some notation. Given an instance of GAP specified by \( G = (J \cup M, E) \) and \( M' \subseteq M \), we let \( \text{GAPLP}(G, M') \) denote the LP relaxation for GAP where we only impose the load constraints for machines in \( M' \). In other words we have dropped the load constraints for \( M \setminus M' \). Note that jobs are still allowed to machines in \( M \setminus M' \).

The key structural lemma that allows for iterated rounding is the following.

**Lemma 6.6.** Let \( y \) be a basic feasible solution to \( \text{GAPLP}(G, M') \). Then one of the following properties holds:

1. There is some \( ij \in E \) such \( y_{ij} = 0 \) or \( y_{ij} = 1 \).
2. There is a machine \( j \in M' \) such that \( d(j) \leq 1 \).
3. There is a machine \( j \in M' \) such that \( \deg(j) = 2 \) and \( \sum_{ij} y_{ij} \geq 1 \).

**Proof.** Let \( y \) be a basic feasible solution. If \( y_{ij} = 0 \) or \( y_{ij} = 1 \) for some edge \( ij \in E \) we are done. Similarly if there is a machine \( j \in M' \) with \( d(j) \leq 1 \) we are done. Thus we restrict our attention to the case when \( y_{ij} \) is strictly fractional for every edge \( ij \in E \) and \( d(j) > 1 \) for each machine \( j \in M' \). Note that \( \deg(i) \geq 2 \) for each job \( i \); if it is one then the job will be assigned fully on that single edge. We will prove that the third property holds.

\( \text{GAPLP}(G, M') \) has \( n + m' \) non-trivial constraints where \( n = |J| \) and \( m' = |M'| \). Since \( y \) is a basic feasible solution, via the rank lemma, this implies that \( |E| = n + m' \). But degree of every \( i \) and every \( j \) is at least 2. This implies that in fact every node’s degree is exactly two in \( E \) and no machine in \( M \setminus M' \) has any edges incident to it. Thus \( G \) consists of a collection of disjoint even cycles. Let \( S \) be any such cycle. For every job \( i \in S \) we have sum of \( \sum_{j} y_{ij} = 1 \); hence
\[ \sum_{i \in S} y_{ij} = |S|/2. \] Hence there is some machine \( j \in S \) such that \( \sum_i y_{ij} \geq 1 \) and moreover its degree is exactly two as we argued. 

\begin{tabular}{|l|}
\hline
\textbf{GAP-Iter-Rounding}(G) \\
\hline
1. \( F = \emptyset, \ M' = M \) \\
2. While \(|F| < n \) do \\
\hspace{0.5cm} A. Obtain an optimum basic feasible solution \( y \) to GAPLP(G, M') \\
\hspace{0.5cm} B. If there is \( ij \in E \) such that \( y_{ij} = 0 \) then \( G = G - ij \) break \\
\hspace{0.5cm} C. If there is \( ij \in E \) such that \( y_{ij} = 1 \) then \\
\hspace{1cm} \( F = F \cup \{(ij)\}, \ G = G - i, \ b_i = b_j - p_{ij}, \ break \) \\
\hspace{0.5cm} D. If there \( j \in M' \) such that \( d(j) = 1 \) or \( d(j) = 2 \) and \( \sum_i y_{ij} \geq 1 \) then \\
\hspace{1cm} \( M' = M' - j \). \\
3. Output assignment \( F \) \\
\hline
\end{tabular}

\textbf{Theorem 6.7.} Given an instance of GAP that is feasible and has optimum cost \( C \), the algorithm GAP-Iter-Rounding outputs an assignment whose cost is at most \( C \) and such that each machine \( j \) has load at most \( 2b_j \).

The proof is by induction on the number of iterations. Alternatively, it is useful to view the algorithm recursively. We will sketch the proof and leave some of the formal details to the reader (who can also consult [46]). We observe that the algorithm makes progress in each iteration via Lemma 6.6. The analysis will consider the four cases that can happen in each iteration: (i) \( y_{ij} = 0 \) for some \( ij \in E \) (ii) \( y_{ij} = 1 \) for some \( ij \in E \) (iii) \( d(j) \leq 1 \) for some \( j \in M' \) and (iv) \( d(j) = 2 \) and \( \sum_i y_{ij} \geq 1 \) for some \( j \in M' \).

Thus the algorithm terminates in polynomial number of iterations. It is also not hard to see that \( F \) corresponds to an assignment of jobs to machines.

\textbf{Observation 6.8.} The algorithm terminates and outputs an assignment of jobs to machines, and and job \( i \) is assigned to \( j \) implies \( ij \in E \).

Now we prove that the assignment has good properties in terms of the cost and loads.

\textbf{Lemma 6.7.} The cost of the LP solution at the start of each iteration is at most \( C - \sum_{ij \in F} c_{ij} \). Hence, at the end of the algorithm the cost of the assignment \( F \) is at most \( C \).
Proof. This is true in the first iteration since \( F = \emptyset \) and the LP cost is less than that of an optimum integer feasible solution. Now consider an iteration assuming that the precondition holds.

If \( y_{ij} = 0 \) we remove \( ij \) from \( E \) and we note that the cost of the LP for the next iteration does not increase since \( y \) itself is feasible for the residual instance.

If \( y_{ij} = 1 \) and we add \( ij \) to \( F \), we can charge the cost of \( ij \) to what the LP has already paid on the edge \( ij \), and the solution \( y \) with \( ij \) removed is feasible to the residual instance obtained by removing job \( i \) and reducing the capacity of \( j \) to \( b_j - p_{ij} \).

In the other cases we do not change \( F \) but drop constraints so the LP cost can only decrease in the subsequent iteration.

Now we upper bound the load on each machine \( j \).

**Lemma 6.8.** For each machine \( j \), \( \sum_{ij \in E} p_{ij} \leq 2b_j \). In fact a stronger property holds: for each \( j \), its load at the end of the algorithm is at most \( b_j \) or there is a single job assigned to \( j \) such that removing it reduces the load of \( j \) to at most \( b_j \).

**Proof.** The proof is by induction on iterations. We will sketch it. Consider a machine \( j \). If \( y_{ij} = 0 \) in some iteration we remove \( ij \) and the load on \( j \) does not change. If \( y_{ij} = 1 \) we add \( ij \) to \( F \) but for subsequent iterations we reduce \( b_j \) by \( p_{ij} \) hence we account for the increase in load.

Thus, the only reason why the load of \( j \) may exceed \( b_j \) is because we drop the load constraint for \( j \) in some iteration. If we drop it when \( d(j) = 1 \) then at most one more job can be assigned to \( j \) and hence its final load can be at most \( b_j + p_{ij} \) for some \( ij \in E \). Thus, if \( p_{ij} \leq b_j \) for all \( i \) the load is at most \( 2b_j \). We can also drop the constraint for \( j \) when \( d(j) = 2 \). However, in this case we have the property that \( y_{ia,j} + y_{ib,j} \geq 1 \) for some two jobs \( i_a \) and \( i_b \) which are the only edges incident to \( j \) at that iteration. Since \( y \) was feasible we also had the constraint that \( p_{ia,j}y_{ia,j} + p_{ib,j}y_{ib,j} \leq b'_j \) where \( b'_j \) was the residual capacity of \( j \) in that iteration. Assume without loss of generality that \( p_{ia,j} \leq p_{ib,j} \); then it follows that \( b'_j \geq p_{ia,j} \). Thus the final load of \( j \) is at most \( b_j - b'_j + p_{ia,j} + p_{ib,j} \) since both \( i_a \) and \( i_b \) can be assigned to \( j \). But this load is at most \( b_j + p_{ib,j} \leq 2b_j \). ■

**Running time:** The algorithm runs in polynomial number of iterations and in each iteration it requires an optimum basic feasible solution to the \( GAPLP(G, M') \). This can be done in polynomial time. We remark that the algorithm is deliberately described in a simple iterative fashion to make the proof easier. One can speed up the algorithm by considering all the cases together in each iteration. Although iterated rounding is a powerful technique the running time is typically expensive. Finding faster algorithms that achieves similar guarantees is an open area of research.
6.3 Maximization version of GAP

We consider the maximization version which we refer to Max-GAP. We have $n$ items and $m$ bins (instead of jobs and machines) where $p_{ij}$ is the size of item $i$ in bin $j$. Each bin $j$ has a capacity $c_j$ and assigning item $i$ to bin $j$ yields a profit/weight $w_{ij}$. The goal is to assign items to bins to maximize the weight while not violating any capacities. When $m = 1$ we obtain the Knapsack problem.

**Multiple Knapsack Problem (MKP):** MKP is a special case of Max-GAP in which $w_{ij} = w_i$ for all $i, j$ and $p_{ij} = p_i$ for all $i, j$. In other words the item characteristics do not depend on where it is assigned to.

**Exercise 6.3.** Prove that MKP does not admit an FPTAS even for $m = 2$.

MKP admits a PTAS [12] and even an EPTAS [37]. Simply greedy algorithms that pack bins one by one using an algorithm for Knapsack as a black box yield a $(1 - 1/e - \epsilon)$ and $1/2 - \epsilon$ approximation for MKP when the bins are indentical and when the bins are arbitrary [12].

In contrast to MKP, Max-GAP does not admit a PTAS. There is an absolute constant $c > 1$ such that a $c - \epsilon$ approximation implies $P = NP$ [12]. However, the following is known.

**Theorem 6.9.** For every fixed $m$ there is a PTAS for Max-GAP.

The preceding theorem can be shown by generalizing the ideas behind the PTAS for Knapsack we discussed in an earlier chapter. An interested reader may try to prove this by considering the case of $m = 2$.

**A $\frac{1}{2}$-approximation:** There is a simple yet clever way to achieve a $\frac{1}{2}$-approximation for Max-GAP via the 2-approximation for the min-cost version that we already saw. Recall that for the min-cost version the algorithm output a solution with cost no more than the optimum cost while violating the capacity of each bin by at most one item. We outline how one can use this to obtain a 2-approximation for Max-GAP and leave the details are an exercise to the reader.

- Reduce the exact maximization version to the exact min-cost version in which all items have to be assigned by adding an extra dummy bin.
- Use the result for min-cost version to obtain an assignment with weight at least that of the optimum while violating each bin’s capacity by at most one item.
- Use the preceding assignment to find a feasible packing of items that has profit at least $OPT/2$. 
For \textbf{Max-GAP} one can use a stronger LP relaxation and obtain a \((1 - 1/e + \delta)\)-approximation. We refer the reader to [20] for this result, and also to [7] for connections to submodular function maximization. The latter connection allows one to obtain an extremely simple \(1/2 - \epsilon\) greedy approximation algorithm that is not obvious to discover.

6.4 Bibliographic Notes

The 2-approximation for unrelated machine scheduling is by Lenstra, Shmoys and Tardos [49]. The same paper showed that unless \(P = NP\) there is no \(3/2 - \epsilon\)-approximation for unrelated machine scheduling. Bridging this gap has been a major open problem in scheduling. A special case called restricted assignment problem has been extensively studied in this context; in such instances \(p_{ij} \in \{p_i, \infty\}\) which means that a job specifies the machines it can be assigned to, but its processing time does not vary among the machines it can be assigned to. The \(3/2\) hardness from [49] applies to restricted assignment problem as well. Svensson [63] showed that a strengthened form of LP relaxation (called the configuration LP) has an integrality gap better than 2 for restricted assignment problem but so far there is no polynomial time algorithm to actually output an assignment! The best algorithm that beats 2 for this case runs in quasi-polynomial time [38].

As we mentioned Shmoys and Tardos obtained the 2-approximation for \textbf{GAP}. The iterated rounding proof is from [46].

The approximability of \textbf{Max-GAP} when \(m\) is not a fixed constant was studied in [12] although a PTAS for fixed \(m\) was known quite early [11]. The current best approximation ratio is via a configuration LP [20]. The precise integrality gap of the configuration LP is an interesting open problem.
Chapter 7

Congestion Minimization in Networks

In Chapter ?? we saw the Scheduling on Unrelated Parallel Machines problem. Here we consider two problems that also consider allocation with the objective of minimizing load/congestion. We will first consider the Congestion Minimization problem in graphs and then abstract it as the Min-Max-Integer-Programs.

7.1 Congestion Minimization and VLSI Routing

A classical routing problem that was partly inspired by VLSI (very large scale integrated) circuit design is the following. Let $G = (V, E)$ be a directed graph and let $(s_1, t_1), \ldots, (s_k, t_k)$ be $k$ source-sink pairs. We want to connect each pair $(s_i, t_i)$ by a path $P_i$ such that the paths do not share edges (or nodes). Alternatively we would like to minimize the maximum number of paths that use any edge — this is called the Congestion Minimization problem. A special case is EDP which is to decide if there are paths for the pairs that are edge-disjoint (NDP is the version where the paths are required to be node-disjoint). EDP and NDP are classical NP-Complete problems and have many important connections to multicommodity flows, routing, cuts, and graph theory. Thus Congestion Minimization is an NP-Hard optimization problem. Here we will consider two variants.

Choosing one path from a given collection: We consider a conceptually simpler variant where we are given a finite path collection $\mathcal{P}_i$ for each pair $(s_i, t_i)$ where each path $P \in \mathcal{P}_i$ connects $s_i$ to $t_i$. The goal is to choose, for each pair $(s_i, t_i)$, one path from $\mathcal{P}_i$ so as to minimize the maximum congestion on any edge. We can develop a simple integer program for this as follows. For each $i$ and each $P \in \mathcal{P}_i$ we have a variable $x_{i,P}$ which indicates whether we choose $P$
to route pair $i$. We need to choose exactly one path for each pair $i$. We want to minimize the maximum number of paths using any edge.

$$\text{minimize } \lambda$$

subject to

$$\sum_{p \in P_i} x_{i,p} = 1 \quad 1 \leq i \leq k$$

$$\sum_{i=1}^{k} \sum_{p \in P_i, p \ni e} x_{i,p} \leq \lambda \quad \forall e \in E$$

$$x_{i,p} \in \{0, 1\} \quad 1 \leq i \leq k, P \in P_i$$

As usual we relax the integer constraints to obtain an LP relaxation where we replace $x_{i,p} \in \{0, 1\}$ with $x_{i,p} \in [0, 1]$ (in this particular case we can simply use $x_{i,p} \geq 0$ due to the other constraints). Note the similarities with the IP/LP for Scheduling on Unrelated Parallel Machines. The LP relaxation is of size polynomial in the input size since the path collection $P_i$ is explicitly given for each $i$ as part of the input.

Let $\lambda^*$ be the optimum LP solution value. There is a technicality that arises just as we saw with Scheduling on Unrelated Parallel Machines. It may happen that $\lambda^* < 1$ while we know that the optimum congestion is at least 1. Technically we should find the smallest integer $\lambda^*$ such that the preceding LP is feasible. We will assume henceforth that $\lambda^*$ is an integer.

How do we round? A simple strategy is randomized rounding. In fact the technique of randomized rounding and analysis via Chernoff bounds was developed in the influential paper of Raghavan and Thompson [RaghavanT] precisely for this problem!

**Randomized-Rounding**

1. Solve LP relaxation and find optimum solution $x^*, \lambda^*$.
2. For $i = 1$ to $k$ do
   A. Pick exactly one path $Q_i \in P_i$ randomly where the probability of picking $P$ is exactly $x^*_{i,p}$.
3. Output $Q_1, Q_2, \ldots, Q_k$.

Note that the choices for the pairs done with independent randomness. The analysis requires the use of Chernoff-Hoeffding bounds. See Chapter B.
Theorem 7.1. Randomized rounding outputs one path per pair and with probability at least $(1 - 1/m^2)$ no edge is contained in more than $c \frac{\log m}{\log \log m} \lambda^*$ paths where $c$ is an absolute constant. Here $m$ are the number of edges in the graph $G$. One can also show that for any fixed $\epsilon > 0$ the congestion is at most $(1 + \epsilon)\lambda^* + c \frac{\log m}{\epsilon^2}$ with high probability.

Proof. The proof is a simple application of Chernoff-bounds and the union bound. Fix an edge $e \in E$. Let $Y_e$ be the random variable which is the total number of paths in $P_1, P_2, \ldots, P_k$ that use $e$. Let $Y_{e,i}$ be the binary random variable that indicates whether $e \in Q_i$. Note that $Y_e = \sum_{i=1}^k Y_{e,i}$.

The first observation is that the variables $Y_{e,1}, Y_{e,2}, \ldots, Y_{e,k}$ are independent since we used independent randomness for the pairs. Second we claim that $E[Y_e] = 1 = \sum_{e \in P_{i,e} \in P} x_{i,p}^*$. Do you see why? Thus, by linearity of expectation,

$$E[Y_e] = \sum_i E[Y_{e,i}] = \sum_i \sum_{p \in P_{i,e} \in P} x_{i,p}^* \leq \lambda^*$$

where the last inequality follows from the LP constraint.

Since $Y_e$ is the sum of independent binary valued random variables and $E[Y_e] \leq \lambda^*$ we can apply Chernoff-bounds to estimate $P[Y_e \geq c \frac{\log m}{\log \log m} \lambda^*]$. Applying Corollary B.5 we conclude that we can choose $c$ such that this probability is at most $1/m^2$. Now we apply the union bound over all edges and conclude that

$$P[\exists e \in E, Y_e \geq c \frac{\log m}{\log \log m} \lambda^*] \leq m/m^2 \leq 1/m^2.$$ 

Thus, with probability $\geq 1 - 1/m^2$ no edge is loaded to more that $c \frac{\log m}{\log \log m} \lambda^*$.

The second bound can be derived in the same way by using the second Chernoff-bound in Corollary B.5. \[\blacksquare\]

Remark 7.1. The bound $(1 + \epsilon)\lambda^* + c \log m/\epsilon^2$ implies that when $\lambda^* \geq c \log m$ we obtain a constant factor approximation.

Remark 7.2. In a graph $m = O(n^2)$ and hence one often sees the bounds expressed in terms of $n$ rather than $m$. We chose to write in terms of $m$ rather than $n$ to highlight the fact that the bound depends on the number of constraints via the union bound. We will discuss later how column sparsity based bounds can give refined results that avoid the union bound.

Implicitly given path collections: In the traditional version of Congestion Minimization we are only given $G$ and the pairs and the goal is to choose one path $P$ for each $(s_i, t_i)$ pair from the set of all paths between $s_i$ and $t_i$. In other words $P_i = \{ P \mid P \text{ is an } s_i-t_i \text{ path in } G \}$. In other words $P_i$ is implicitly defined
and its size can be exponential in the graph size. It is then not obvious that we can solve the LP relaxation that we saw above. However, one can indeed solve it in polynomial-time via the Ellipsoid method. First, we observe that the LP could have an exponential number of variables but only a polynomial number of non-trivial constraints: \( k \) for the pairs and \( m \) for the edges. Thus, one is guaranteed, by the rank lemma that there is an optimum solution that has only \( k + m \) non-zero variables. Thus, one is guaranteed the existence of a polynomial-sized solution. To see that one can indeed find it efficiently, we need to look at the dual and notice that the separation oracle for the dual is the shortest path problem.

Another way to see that the LP can solved is by writing a compact formulation via the well-known multicommodity flow. We want to send one unit of flow from \( s_i \) to \( t_i \) so that the total flow on any edge is at most \( \lambda \). We use variables \( f(e, i) \) to denote a flow for pair \( i \) on edge \( e \).

\[
\begin{align*}
\text{minimize} & \quad \lambda \\
\text{subject to} & \quad \sum_{e \in \delta^+(s_i)} f(e, i) - \sum_{e \in \delta^-(s_i)} f(e, i) = 1 & 1 \leq i \leq k \\
& \quad \sum_{e \in \delta^+(v)} f(e, i) - \sum_{e \in \delta^-(v)} f(e, i) = 0 & 1 \leq i \leq k, v \in V - \{s_i, t_i\} \\
& \quad \sum_{i=1}^{k} f(e, i) \leq \lambda & \forall e \in E \\
& \quad f(e, i) \geq 0 & 1 \leq i \leq k, e \in E
\end{align*}
\]

The preceding multicommodity flow LP has a polynomial number of variables and can be solved in polynomial-time. Given a flow, for each commodity/pair \( i \) we can take the one unit of \( s_i-t_i \) flow and use standard flow-decomposition to obtain a path-flow with at most \( m \) paths in the collection. We can then apply the rounding that we saw above with an given explicit path collection in exactly the same way.

Remark 7.3. The Ellipsoid based algorithm may seem impractical. However, one can approximately solve the implicit path based LP via multiplicative weight update methods efficiently. The implicit formulation and Ellipsoid method is also useful when one may want to restrict \( \mathcal{P}_i \) in some fashion. For instance we can set \( \mathcal{P}_i \) to be the set of all \( s_i-t_i \) paths in \( G \) with at most \( d \) edges for some given parameter \( d \). This will ensure that we choose only “short” paths for each pair. It is not hard to see that the separation oracle for the dual is another shortest path type problem that can be solved efficiently (via Bellman-Ford type algorithm). This is not easy to capture/see via the compact flow based formulation.
Derandomization: Is there a deterministic algorithm with the roughly the same approximation guarantee? The algorithm can be derandomized via the notion of pessimistic estimators. Congestion Minimization was one of the first instances with a sophisticated use of this technique [57].

Integrality gap and Hardness of Approximation: There is simple yet clever example demonstrating that the integrality gap of the flow relaxation in directed graphs is $\Omega(\log m / \log \log m)$ [48]. In a remarkable result, [15] showed that $O(\log m / \log \log m)$ is the hardness factor. The complexity of Congestion Minimization is less clear in undirected graphs. It is known that the LP integrality gap and hardness of approximation are $\Omega(\log \log n / \log \log \log n)$ [3]. Closing the gap between the upper and lower bounds is a major open problem.

Here we outline the integrality gap example for directed graphs from [48]. The graph $G$ and the pairs are constructed in a recursive fashion. Let $h$ be a parameter that we will fix later. We start with a directed path $v_0, v_1, \ldots, v_n$. We add a demand pair $(B_1, C_1)$ which connects to the path as follows. We partition the path into $n/h$ paths of equal length: add an arc to $B$ to the start of each sub-path and an arc from the end of each sub-path to $C$. See figure.

One can see from the figure that the pair $(s, t)$ can split its flow along $h$ paths. Now we consider each of the $h$ sub-paths and recursively create an instance on the path with length $n/h - 1$ (while keeping parameter $h$ the same). Note that in the second level of the recursion we add $h$ new source-sink pairs, one for each sub-path. We stop the recursion when the size of the sub-path is $\Theta(h)$. Let $d$ be the depth of the recursion.

We claim that there is a fractional routing of all demand pairs where the congestion is at most $d/h$. This follows by splitting the flow of the pairs $h$ ways. The next claim is that some edge has congestion $d$ in any integral routing. This can be seen inductively. The top level pair $(s, t)$ has to choose one amongst the $h$ sub-paths — all edges in that sub-path will be used by the route for $(s, t)$. Inductively there is some edge in that sub-path with congestion $d - 1$ and hence
the congestion of that edge will $d$ when we add the path for $(s, t)$.

It now remains to set the parameters. If we choose $h = \log^2 n$ say then $d = \Theta(\log n / \log \log n)$. The fractional congestion is $\leq 1$ and integrally congestion is $\Theta(\log n / \log \log n)$.

**Short paths and improved congestion via Lovász-Local-Lemma:** We consider the congestion minimization problem when the path for each pair is required to be “short”. By this we mean that we are required to route on a path with at most $d$ edges where $d$ is some given parameter. One can imagine that in many applications $d$ is small and is a fixed constant, say 10. The question is whether the approximation ratio can be improved. Indeed one can show that the LP integrality gap is $O(\log d / \log \log d)$. Thus, when $d \ll n$ we get a substantial improvement. However, proving this and obtaining a polynomial time algorithm are quite non-trivial. One requires the use of the subtle Lovász-Local-Lemma (LLL), a powerful tool in probabilistic combinatorics. Typically LLL only gives a proof of existence and there was substantial work in making LLL constructive/efficient. Srinivasan obtained an algorithm via derandomization of LLL in this context with a lot of technical work [61]. There was a breakthrough work of Moser and Tardos [53] that gave an extremely simple way to make LLL constructive and this has been refined and developed over the last decade. For the congestion minimization problem we refer the reader to [27] which builds upon [53] and describes an efficient randomized algorithm that outputs a solution with congestion $O(\log d / \log \log d)$. In fact the application is given in the context of a more abstract problem that we discuss in the next section.

**Integer flows and Unsplittable flows:** We worked with the simple setting where each pair $(s_i, t_i)$ wishes to send one unit of flow. One can imagine a situation where one wants to send $d_i$ units of flow for pair $i$ where $d_i$ is some (integer) demand value. There are two interesting variants. The first one requires integer valued flow for each pair which means that we want to find $d_i$ paths for $(s_i, t_i)$ that each carry one unit of flow (the paths can overlap). This variant can be essentially reduced to the unit demand flow by creating $d_i$ copies of $(s_i, t_i)$ — we leave this as a simple exercise for the reader. The second variant is that we want each pair’s flow of $d_i$ units to be sent along a single path — this is called unsplittable flow. When discussing unsplittable flow it is also natural to consider capacities on the edges. Thus, each edge has a capacity $u_e$ and one wants to minimize congestion relative to $u_e$. The techniques we discussed can be generalized relatively easily to this version as well to obtain the same kind of bounds. The unsplittable flow problem is interesting even in the setting where there is a single source/sink or when the graph is a simple ring or a path. Interesting results are known here and we refer the reader to [1, 9, 17, 26, 52, 60] for further pointers.
7.2 Min-max Integer Programs

If one looks at the rounding and analysis for Congestion Minimization we notice that the algorithm uses very little about the structure of the graph. This can be thought about in two ways. One is that perhaps we can do better by exploiting graph structure. Two, we can abstract the problem into a more general class where the same technique applies. As we mentioned, in directed graphs the bound of $O(\log n / \log \log n)$ is tight but the bound may not be tight in undirected graphs which admit more structure.

Here we consider the second point and develop a resource allocation view point while making an analogy to Congestion Minimization so that the abstract problem can be more easily understood. Suppose we have $m$ resources $e_1, e_2, \ldots, e_m$. We have $k$ requests $r_1, r_2, \ldots, r_k$. Each request $i$ can be satisfied in $\ell_i$ ways — let $P_i$ denote a collection of $\ell_i$ vectors $v_{i,1}, v_{i,2}, \ldots, v_{i,\ell_i}$. Each vector $v_{i,j} \in P_i$ is an $m$-dimensions: for each $k \in [m]$, $v_{i,j,k}$ is a scalar that represents the load it induces on resource $e_k$. The goal is to choose, for each $i$, exactly one $j \in [\ell_i]$ so as to minimize the maximum load on any resource. One can write this conveniently as the following integer program where we have variables $x_{i,j}$ for $1 \leq i \leq k$ and $1 \leq j \leq \ell_i$, which indicates whether $i$ chooses $j$.

\[
\begin{align*}
\text{minimize} & \quad \lambda \\
\text{subject to} & \quad \sum_{1 \leq j \leq \ell_i} x_{i,j} = 1 & 1 \leq i \leq k \\
& \quad \sum_{i=1}^{k} \sum_{1 \leq j \leq \ell_i} v_{i,j,k} x_{i,j} \leq \lambda & \forall e_k \\
& \quad x_{i,j} \in \{0,1\} & 1 \leq i \leq k, 1 \leq j \leq \ell_i
\end{align*}
\]

One can view the above integer program compactly as

\[
\begin{align*}
\text{minimize} & \quad \lambda \\
\text{subject to} & \quad \sum_{1 \leq j \leq \ell_i} x_{i,j} = 1 & 1 \leq i \leq k \\
& \quad Ax \leq \lambda 1 \\
& \quad x_{i,j} \in \{0,1\} & 1 \leq i \leq k, 1 \leq j \leq \ell_i
\end{align*}
\]

where $A$ is a non-negative matrix with $m$ rows. As with Scheduling on Unrelated Parallel Machines we need to be careful when relaxing the IP to an
LP since the optimum solution to the LP can be a poor lower bound unless we ensure that \( x_{i,j} = 0 \) if \( v_{i,j,k} > \lambda \). We will assume that we have indeed done this.

One can do randomized rounding exactly as we did for Congestion Minimization and obtain an \( O(\frac{\log m}{\log \log m}) \) approximation. We say that \( A \) is \( d \)-column-sparse if the maximum number of non-zeroes in any column of \( A \) is at most \( d \). This corresponds to paths in Congestion Minimization being allowed to have only \( d \) edges. One can obtain an \( O(\frac{\log d}{\log \log d}) \)-approximation in this more general setting as well [27].
Bibliography


Appendix A

Basic Feasible Solutions to LPs and the Rank Lemma

We discuss the rank lemma about vertex solutions for linear programs. Recall that a polyhedron in $\mathbb{R}^n$ is defined as the intersection of finite collection of half spaces. Without loss of generality we can assume that it is defined by a system of inequalities of the form $Ax \leq b$ where $A$ is a $m \times n$ matrix and $b$ is a $m \times 1$ vector. A polyhedron $P$ is bounded if $P$ is contained in finite radius ball around the origin. A polytope in $\mathbb{R}^n$ is defined as the convex hull of a finite collection of points. A fundamental theorem about linear programming states that any bounded polyhedron is a polytope. If the polyhedron is not bounded then it can be expressed as the Minkowski sum of a polytope and a cone.

A bounded polyhedron $P$ in $\mathbb{R}^n$ defined by a system $Ax \leq b$ must necessarily have $m \geq n$. A point $p \in P$ is a basic feasible solution or a vertex solution of the system if it is the unique solution to a system $H'x = 1'$ where $H'$ is a sub-matrix of $A$ with $n$ inequalities and the rank of $H'$ is equal to $n$. The inequalities in $H'$ are said to be tight for $y$. Note that there may be many other inequalities in $Ax \leq b$ that are tight at $y$ and in general there many be many different rank $n$ sub-matrices that give rise to the same basic feasible solution $y$.

Lemma A.1. Suppose $y$ is a basic feasible solution of a system $Ax \leq b$, $\ell \leq x \leq u$ where $A$ is a $m \times n$ matrix and $\ell$ and $u$ are vectors defining lower and upper bounds on the variables $x \in \mathbb{R}^n$. Let $S = \{i : \ell_i < y_i < u_i\}$ be the set of indices of “fractional” variables in $y$. Then $|S| \leq \text{rank}(A) \leq m$. In particular the number of fractional variables in $y$ is at most the number of “non-trivial” constraints (those that are defined by $A$).

An extension of the previous lemma is often useful when the system defining the polyhedron has equality constraints.
Corollary A.1. Suppose $y$ is a basic feasible solution of a system $Ax \leq b, Cx = d, \ell \leq x \leq u$ where $A$ is an $m \times n$ matrix, $C$ is an $m' \times n$ matrix, and $\ell$ and $u$ are vectors defining lower and upper bounds on the variables $x \in \mathbb{R}^n$. Let $S = \{i : \ell_i < y_i < u_i\}$ be the set of indices of “fractional” variables in $y$. Then $|S| \leq \text{rank}(A, C) \leq m + m'$.

A special case of the preceding corollary is called the rank lemma in [46].

The lemmas are a simple consequence of the definition of basic feasible solution. We will focus on the proof of Lemma A.1. It is interesting only when $\text{rank}(A)$ or $m$ is smaller than $n$, otherwise the claim is trivial. Before we prove it formally we observe some simple corollaries. Suppose we have a system $Ax \leq b, x \geq 0$ where $\ell$ is a $< \times =$ matrix, $\ell$ is a $< \times =$ matrix, and $\ell$ and $D$ are vectors defining lower and upper bounds on the variables $x \in \mathbb{R}^n$. Let $(8 = \{8 : \ell_8 < H_8 < D_8\})$ be the set of indices of “fractional” variables in $H$. Then $|8| \leq \text{rank}(A, C) \leq m + m'$.

A.0.1 Some Examples

We give some examples to illustrate the utility of the rank lemma in the context of LP relaxations that arise in approximation algorithms.

Knapsack: The natural LP relaxation for this of the form $\max \sum_{i=1}^n w_i x_i$ subject to $x \in [0, 1]^n, \sum_{i=1}^n s_i x_i \leq 1$ consisting of a single non-trivial constraint. A basic feasible solution has at most 1 fractional variable. See Chapter 3.

Packing Integer Programs (PIPs): The LP relaxation is of the form $\max \sum_{i=1}^n w x$ subject to $Ax \leq b, x \in [0, 1]^n$ where $A$ is an $m \times n$ non-negative matrix. See Chapter 4. When $m = 1$ we have the Knapsack problem and hence the general problem is sometimes referred to as the $m$-dimensional Knapsack problem, especially when $m$ is a fixed constant. A basic feasible solution for the LP has
at most \( m \) fractional variables. When \( m \) is a fixed constant one can exploit this after guessing the big items to obtain a PTAS.

**Generalized Assignment:** See Chapter 6.

### A.0.2 Connection to Caratheodory’s Theorem

Suppose we have \( n \) points \( P = \{p_1, p_2, \ldots, p_n\} \) in \( d \)-dimensional Euclidean space \( \mathbb{R}^d \). A point \( p \in \mathbb{R}^d \) is in the convex hull of \( P \) iff \( p \) is a convex combination of points in \( P \). Formally, this means that exist scalars \( \lambda_1, \ldots, \lambda_n \geq 0 \) such that \( \sum_i \lambda_i = 1 \) and \( p = \sum_i \lambda_i p_i \) (note that this is a vector sum). Caratheodory’s theorem states that if \( p \) is in the convex hull of \( P \) then there is subset \( P' \subseteq P \) such that \( p \) is in the convex hull of \( P' \) and \( |P'| \leq d + 1 \). One can prove Caratheodory’s theorem directly but it is helpful to see it also as a consequence of the rank lemma.

Consider the system of inequalities \( \sum_i \lambda_i \geq 0, 1 \leq i \leq n, \sum_i \lambda_i = 1, \sum_i \lambda_i p_i = p \) where the system \( \sum_i \lambda_i p_i = p \) consists of \( d \) equalities. This system of inequalities in the variables \( \lambda_1, \ldots, \lambda_n \) is feasible by assumption (since \( p \) is in the convex hull of \( P \)). If we take any basic feasible solution of this system of inequalities we see that at most \( d + 1 \) of them are non-zero by the rank lemma.

One implication of Caratheodory’s theorem in the context of combinatorial optimization is the following. Suppose we have a polytope \( P \) which is an LP relaxation of some combinatorial problem and let \( x \in P \) be any feasible point. Then \( x \) can be written as a convex combination of at most \( n + 1 \) vertices of \( P \) where \( n \) is number of variables. Moreover, via the Ellipsoid method, one can find such a convex combination efficiently as long as one can optimize over \( P \) efficiently. As an example suppose \( G = (V, E) \) is a graph and \( P \) is the spanning tree polytope of \( G \) (the vertices are the characteristic vectors of spanning trees) which is \( \mathbb{R}^m \) \( (m = |E|) \). Then any fractional spanning tree \( x \in P \) can be decomposed into at most \( m + 1 \) spanning trees.

In the context of approximation \( P \) is typically a relaxation of some hard combinatorial optimization problem. In such a case the vertices of \( P \) do not correspond to structures we are interested in. For example we can consider the minimum Steiner tree problem in a graph \( G = (V, E) \) with terminal set \( S \subseteq V \). There are several LP relaxations but perhaps the simplest one is the cut relaxation which has an integrality gap of \( 2 \). In such a case a feasible point \( x \in P \) cannot be decomposed into convex combination of Steiner trees. However it can be shown that \( 2x \) dominates a convex combination of Steiner trees and such a convex combination can be found efficiently. It requires more technical work to precisely formalize this and we refer the reader to the work of Carr and Vempala [8] — you can also find a few applications of such decompositions in the same paper and it is a simple yet powerful tool to keep in mind.
Appendix B

Probabilistic Inequalities

The course will rely heavily on probabilistic methods. We will mostly rely on discrete probability spaces. We will keep the discussion high-level where possible and use certain results in a black-box fashion.

Let $\Omega$ be a finite set. A probability measure $p$ assigns a non-negative number $p(\omega)$ for each $\omega \in \Omega$ such that $\sum_{\omega \in \Omega} p(\omega) = 1$. The tuple $(\Omega, p)$ defines a discrete probability space; an event in this space is any subset $A \subseteq \Omega$ and the probability of an event is simply $p(A) = \sum_{\omega \in A} p(\omega)$. When $\Omega$ is a continuous space such as the interval $[0, 1]$ things get trickier and we need to talk about a measure spaces $\sigma$-algebras over $\Omega$; we can only assign probability to certain subsets of $\Omega$. We will not go into details since we will not need any formal machinery for what we do in this course.

An important definition is that of a random variable. We will focus only on real-valued random variables in this course. A random variable $X$ in a probability space is a function $\Omega \to \mathbb{R}$. In the discrete setting the expectation of $X$, denoted by $E[X]$, is defined as $\sum_{\omega \in \Omega} p(\omega)X(\omega)$. For continuous spaces $E[X] = \int X(\omega)d\mu(\omega)$ with appropriate definition of the integral. The variance of $X$, denoted by $\text{Var}[X]$ or as $\sigma_X^2$, is defined as $E[(X - E[X])^2]$. The standard deviation is $\sigma_X$, the square root of the variance.

Theorem B.1 (Markov’s Inequality). Let $X$ be a non-negative random variable such that $E[X]$ is finite. Then for any $t > 0$, $P[X \geq t] \leq E[X]/t$.

Proof. The proof is in some sense obvious, especially in the discrete case. Here is a sketch. Define a new random variable $Y$ where $Y(\omega) = X(\omega)$ if $X(\omega) < t$ and $Y(\omega) = t$ if $X(\omega) \geq t$. $Y$ is non-negative and $Y \leq X$ point-wise and hence
\[ E[Y] \leq E[X]. \] We also see that:

\[
E[X] \geq E[Y] = \sum_{\omega: X(\omega) < t} X(\omega) p(\omega) + \sum_{\omega: X(\omega) \geq t} t p(\omega)
\geq t \sum_{\omega: X(\omega) \geq t} p(\omega) \quad \text{(since } X \text{ is non-negative)}
\geq t P[X \geq t].
\]

The continuous case follows by replacing sums by integrals.

Markov’s inequality is tight under the assumption. It is useful to construct an example. The more information we have about a random variable the better we can bound its deviation from the expectation.

**Theorem B.2 (Chebyshev’s Inequality).** Let \( X \) be a random variable with \( E[X] \) and \( \text{Var}[X] \) are finite. Then \( P[|X| \geq t] \leq \frac{E[X^2]}{t^2} \) and \( P[|X - E[X]| \geq t] \leq 1/t^2. \)

**Proof.** Consider the non-negative random variable \( Y = X^2 \). \( P[|X| \geq t] = P[Y \geq t^2] \) and we apply Markov’s inequality to the latter. The second inequality is similar by considering \( Y = (X - E[X])^2 \).

**Chernoff-Hoeffding Bounds:** We will use several times various forms of the Chernoff-Hoeffding bounds that apply to a random variable that is a finite sum of bounded and independent random variables. There are several versions of these bounds. First we state a general bound that is applicable to non-negative random variables and is dimension-free in that it depends only the expectation rather than the number of variables.

**Theorem B.3 (Chernoff-Hoeffding).** Let \( X_1, X_2, \ldots, X_n \) be independent binary random variables and let \( a_1, a_2, \ldots, a_n \) be coefficients in \([0, 1]\). Let \( X = \sum_i a_i X_i \). Then

- For any \( \mu \geq E[X] \) and any \( \delta > 0 \), \( P[X > (1 + \delta)\mu] \leq \left( \frac{e^\delta}{(1+\delta)^{1+\delta}} \right)^{\mu}. \)
- For any \( \mu \leq E[X] \) and any \( \delta > 0 \), \( P[X < (1 - \delta)\mu] \leq e^{-\mu\delta^2/2}. \)

The following corollary bounds the deviation from the mean in both directions.

**Corollary B.4.** Under the conditions of Theorem B.3, the following hold:

- If \( \delta > 2e - 1 \), \( P[X \geq (1 + \delta)\mu] \leq 2^{-(1+\delta)}\mu. \)
- For any \( U \) there is a constant \( c(U) \) such that for \( 0 < \delta < U \), \( P[X \geq (1 + \delta)\mu] \leq e^{-c(U)\delta^2\mu}. \) In particular, combining with the lower tail bound,

\[
P[|X - \mu| \geq \delta \mu] \leq 2e^{-c(U)\delta^2\mu}.
\]
We refer the reader to the standard books on randomized algorithms [55] and [51] for the derivation of the above bounds.

If we are interested only in the upper tail we also have the following bounds which show the dependence of $\mu$ on $n$ to obtain an inverse polynomial probability.

**Corollary B.5.** Under the conditions of Theorem B.3, there is a universal constant $\alpha$ such that for any $\mu \geq \max\{1, E[X]\}$, and sufficiently large $n$ and for $c \geq 1$, $P[X > \frac{\alpha c \ln n}{\ln n} \cdot \mu] \leq 1/n^c$. Similarly, there is a constant $\alpha$ such that for any $\epsilon > 0$, $P[X \geq (1 + \epsilon)\mu + \alpha c \log n / \epsilon] \leq 1/n^c$.

**Remark B.1.** If the $X_i$ are in the range $[0, b]$ for some $b$ not equal to 1 one can scale them appropriately and then use the standard bounds.

Some times we need to deal with random variables that are in the range $[-1, 1]$. Consider the setting where $X = \sum_i X_i$ where for each $i$, $X_i \in [-1, 1]$ and $E[X_i] = 0$, and the $X_i$ are independent. In this case $E[X] = 0$ and we can no longer expect a dimension-free bound. Suppose each $X_i$ is 1 with probability $1/2$ and $-1$ with probability $1/2$. Then $X = \sum_i X_i$ corresponds to a 1-dimensional random walk and even though the expected value is 0 the standard deviation of $X$ is $\Theta(\sqrt{n})$. One can show that $P[|X| \geq t\sqrt{n}] \leq 2e^{-t^2/2}$. For these settings we can use the following bounds.

**Theorem B.6.** Let $X_1, X_2, \ldots, X_n$ be independent random variables such that for each $i$, $X_i \in [a_i, b_i]$. Let $X = \sum_i a_i X_i$ and let $\mu = E[X]$. Then

$$P[|X - \mu| \geq t] \leq 2\exp\left(-\frac{t^2}{\sum_i (b_i - a_i)^2}\right).$$

In particular if $b_i - a_i \leq 1$ for all $i$ then

$$P[|X - \mu| \geq t] \leq 2\exp\left(-\frac{t^2}{2}\right).$$

Note that $\text{Var}[X] = \sum_i \text{Var}[X_i]$. One can show a bound based of the following form

$$P[|X - \mu| \geq t] \leq 2\exp\left(-\frac{t^2}{2\text{Var}[X] + M/3}\right)$$

where $|X_i| \leq M$ for all $i$.

**Remark B.2.** Compare the Chebyshev bound to the Chernoff-Hoeffding bounds for the same variance.

Sariel Har-Peled maintains a cheat sheet of Chernoff bounds and also has an interesting derivation. See his notes [28].
Statistical Estimators, Reducing Variance and Boosting: Randomized algorithms compute a function $f$ of the input. In many cases they producing an unbiased estimator, via a random variable $X$, for the function value. That is, the algorithm will have the property that the $\mathbb{E}[X]$ is the desired value. Note that the randomness is internal to the algorithm and not part of the input (we can also consider randomness in the input). Having an estimator is not often useful. We will also typically try to evaluate $\text{Var}[X]$ and then we can use Chebyshev’s inequality. One way to reduce the variance of the estimate is to run the algorithm in parallel (with separate random bits) and get estimators $X_1, X_2, \ldots, X_h$ and use $X = \frac{1}{h} \sum_i X_i$ as the final estimator. Note that $\text{Var}[X] = \frac{1}{h} \sum_i \text{Var}[X_i]$ since the $X_i$ are independent. Thus the variance has been reduced by a factor of $h$. A different approach is to use the median value of $X_1, X_2, \ldots, X_h$ as the final estimator. We can then use Chernoff-Hoeffding bounds to get a much better dependence on $h$. In fact both approaches can be combined.