This book is designed to be a textbook for graduate-level courses in approximation algorithms. After some experience teaching minicourses in the area in the mid-90s, we sat down and wrote out an outline of the book. Then I (DPW), at the time an IBM Research Staff Member, taught several iterations of the course following the outline we had devised, in Columbia University’s Industrial Engineering and Operations Research department in Spring 1998, in Cornell University’s School of Operations Research and Industrial Engineering in Fall 1998, and at the Massachusetts Institute of Technology’s Laboratory for Computer Science in Spring 2000. The lecture notes from these courses were made available, and we got enough positive feedback on them from students and from professors teaching such courses elsewhere that we felt we were on the right track. Since then, there have been many exciting developments in the area, and we have added many of them into our book; I taught additional iterations of the course at Cornell in Fall 2006 and Fall 2009 (this time as a Cornell faculty member) in order to field test some of the writing of the newer results.

The courses were developed for students who have already had a class, undergraduate or graduate, in algorithms, and who were comfortable with the idea of mathematical proofs about the correctness of algorithms. The book assumes this level of preparation. The book also assumes some basic knowledge of probability theory (for instance, how to compute the expected value of a discrete random variable). Finally, we assume that the reader knows something about NP-completeness, at least enough to know that there might be good reason for wanting fast, approximate solutions to NP-hard discrete optimization problems. At one or two points in the book, we do an NP-completeness reduction to show that it can be hard to find approximate solutions to such problems; we include a short appendix on the problem class NP and the notion of NP-completeness for those unfamiliar with the concepts. However, the reader unfamiliar with such reductions can also safely skip over such proofs.

In addition to serving as a graduate textbook, we have also envisioned the book as a way for students to get the background to read current research in the area of approximation algorithms. In particular, we wanted a book that we could hand our own Ph.D. students just starting in the field, and say “Here, read this.”

We further hope that the book will serve as a reference to the area of approximation algorithms for researchers who are generally interested in the heuristic solution of discrete optimization problems; such problems appear in areas as diverse as traditional operations research planning problems (such as facility location and network design) to computer science prob-
lems in database and programming language design to advertising issues in viral marketing. We hope that the book helps researchers understand the techniques available in the area of approximation algorithms for approaching such problems.

We have taken several particular perspectives in writing the book. The first is that we wanted to organize the material around certain principles of designing approximation algorithms, around algorithmic ideas that have been used in different ways and applied to different optimization problems. The title “The Design of Approximation Algorithms” was carefully chosen. The book is structured around these design techniques. The introduction applies several of them to a single problem, the set cover problem. The book then splits into two parts. In the first part, each chapter is devoted to a single algorithmic idea (e.g., “greedy and local search algorithms”, “rounding data and dynamic programming”), and the idea is then applied to several different problems. The second part revisits all of the same algorithmic ideas, but gives more sophisticated treatments of them; the results covered here are usually much more recent. The layout allows us to look at several central optimization problems repeatedly throughout the book, returning to them as a new algorithmic idea leads to a better result than the previous one. In particular, we revisit such problems as the uncapacitated facility location problem, the prize-collecting Steiner tree problem, the bin-packing problem, and the maximum cut problem several times throughout the course of the book.

The second perspective is that we treat linear and integer programming as a central aspect in the design of approximation algorithms. This perspective is from our background in the operations research and mathematical programming communities. It is a little unusual in the computer science community, and students coming from a computer science background may not be familiar with the basic terminology of linear programming. We introduce the terms we need in the first chapter, and we include a brief introduction to the area in an appendix.

The third perspective we took in writing the book is that we have limited ourselves to results that are simple enough for classroom presentation while remaining central to the topic at hand. Most of the results in the book are ones that we have taught ourselves in class at one point or another. We bent this rule somewhat in order to cover the recent, exciting work by Arora, Rao and Vazirani [22] applying semidefinite programming to the uniform sparsest cut problem. The proof of this result is the most lengthy and complicated of the book.

We are grateful to a number of people who have given us feedback about the book at various stages in its writing. We are particularly grateful to James Davis, Lisa Fleischer, Isaac Fung, Igor Gorodezky, Rajiv Gandhi, Nick Harvey, Anna Karlin, Vijay Kothari, Katherine Lai, Gwen Spencer, and Anke van Zuylen for very detailed comments on a number of sections of the book. Additionally, the following people spotted typos, gave us feedback, helped us understand particular papers, and made useful suggestions: Bruno Abrahao, Hyung-Chan An, Matthew Andrews, Eliot Anshelevich, Sanjeev Arora, Ashwinkumar B.V., Moses Charikar, Chandra Chekuri, Joseph Cheriyan, Chao Ding, Dmitriy Drusvyatskiy, Michel Goemans, Sudipto Guha, Anupam Gupta, Sanjeev Khanna, Lap Chi Lau, Renato Paes Leme, Jiawei Qian, Yogeshwer Sharma, Viktor Simjanski, Mohit Singh, Éva Tardos, Mike Todd, Di Wang, and Ann Williamson. We also thank a number of anonymous reviewers who made useful comments. Eliot Anshelevich, Joseph Cheriyan, Lisa Fleischer, Michel Goemans, Nicole Immorlica, and Anna Karlin used various drafts of the book in their courses on approximation algorithms and gave us useful feedback about the experience of using the book. We received quite a number of useful comments from the students in Anna’s class: Benjamin Birnbaum, Punyashloka Biswal, Elisa Celis, Jessica Chang, Mathias Hallman, Alyssa Joy Harding, Trinh Huynh, Alex Jaffe, Karthik Mohan, Katherine Moore, Cam Thach Nguyen, Richard Pang, Adrian Sampson, William Austin Webb, and Kevin Zatloukal. Our editor at Cambridge, Lauren Cowles, im-

pressed us with her patience in waiting for this book to be completed and gave us a good deal of useful advice.

We are also grateful to our wives and families – to Ann, Abigail, Daniel, and Ruth, and to Éva, Rebecca, and Amy – for their patience and support during the writing of this volume.

Finally, we hope the book conveys some of our enthusiasm and enjoyment of the area of approximation algorithms. We hope that you, dear reader, will enjoy it too.

David P. Williamson
David B. Shmoys

Ithaca, New York, USA

June 2010
# Table of Contents

- **Preface**  
  - 3

- **I An introduction to the techniques**  
  - 11

  - 1 An introduction to approximation algorithms  
    - 1.1 The whats and whys of approximation algorithms  
    - 1.2 An introduction to the techniques and to linear programming: the set cover problem  
    - 1.3 A deterministic rounding algorithm  
    - 1.4 Rounding a dual solution  
    - 1.5 Constructing a dual solution: the primal-dual method  
    - 1.6 A greedy algorithm  
    - 1.7 A randomized rounding algorithm  
  - 35

  - 2 Greedy algorithms and local search  
    - 2.1 Scheduling jobs with deadlines on a single machine  
    - 2.2 The $k$-center problem  
    - 2.3 Scheduling jobs on identical parallel machines  
    - 2.4 The traveling salesman problem  
    - 2.5 Maximizing float in bank accounts  
    - 2.6 Finding minimum-degree spanning trees  
    - 2.7 Edge coloring  
  - 65

  - 3 Rounding data and dynamic programming  
    - 3.1 The knapsack problem  
    - 3.2 Scheduling jobs on identical parallel machines  
    - 3.3 The bin-packing problem  
  - 81

  - 4 Deterministic rounding of linear programs  
    - 4.1 Minimizing the sum of completion times on a single machine  
    - 4.2 Minimizing the weighted sum of completion times on a single machine  
  - 84
4.3 Solving large linear programs in polynomial time via the ellipsoid method  .  .  .  .  .  . 86 
4.4 The prize-collecting Steiner tree problem  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 88 
4.5 The uncapacitated facility location problem  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 91 
4.6 The bin-packing problem  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 95 

5 Random sampling and randomized rounding of linear programs  .  .  .  .  .  .  .  .  .  .  . 105 
5.1 Simple algorithms for MAX SAT and MAX CUT  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 106 
5.2 Derandomization  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 108 
5.3 Flipping biased coins  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 110 
5.4 Randomized rounding  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 111 
5.5 Choosing the better of two solutions  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 114 
5.6 Non-linear randomized rounding  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 116 
5.7 The prize-collecting Steiner tree problem  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 118 
5.8 The uncapacitated facility location problem  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 120 
5.9 Scheduling a single machine with release dates  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 124 
5.10 Chernoff bounds  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 128 
5.11 Integer multicommodity flows  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 132 
5.12 Random sampling and coloring dense 3-colorable graphs  .  .  .  .  .  .  .  .  .  .  . 133 

6 Randomized rounding of semidefinite programs  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 141 
6.1 A brief introduction to semidefinite programming  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 141 
6.2 Finding large cuts  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 143 
6.3 Approximating quadratic programs  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 147 
6.4 Finding a correlation clustering  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 150 
6.5 Coloring 3-colorable graphs  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 153 

7 The primal-dual method  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 161 
7.1 The set cover problem: a review  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 161 
7.2 Choosing variables to increase: the feedback vertex set problem in undirected graphs  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 164 
7.3 Cleaning up the primal solution: the shortest s-t path problem  .  .  .  .  .  .  .  . 168 
7.4 Increasing multiple variables at once: the generalized Steiner tree problem  .  . 170 
7.5 Strengthening inequalities: the minimum knapsack problem  .  .  .  .  .  .  .  .  . 178 
7.6 The uncapacitated facility location problem  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 180 
7.7 Lagrangean relaxation and the k-median problem  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 184 

8 Cuts and metrics  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 195 
8.1 The multiway cut problem and a minimum-cut based algorithm  .  .  .  .  .  .  . 196 
8.2 The multiway cut problem and an LP rounding algorithm  .  .  .  .  .  .  .  .  .  . 197 
8.3 The multicut problem  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 203 
8.4 Balanced cuts  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  . 208 
8.5 Probabilistic approximation of metrics by tree metrics  .  .  .  .  .  .  .  .  .  .  .  . 211 
8.6 An application of tree metrics: Buy-at-bulk network design  .  .  .  .  .  .  .  .  . 216 
8.7 Spreading metrics, tree metrics, and linear arrangement  .  .  .  .  .  .  .  .  .  .  . 220


## Table of Contents

II Further uses of the techniques 231

9 Further uses of greedy and local search algorithms 233
  9.1 A local search algorithm for the uncapacitated facility location problem 234
  9.2 A local search algorithm for the k-median problem 239
  9.3 Minimum-degree spanning trees 243
  9.4 A greedy algorithm for the uncapacitated facility location problem 247

10 Further uses of rounding data and dynamic programming 257
  10.1 The Euclidean traveling salesman problem 257
  10.2 The maximum independent set problem in planar graphs 269

11 Further uses of deterministic rounding of linear programs 281
  11.1 The generalized assignment problem 282
  11.2 Minimum-cost bounded-degree spanning trees 286
  11.3 Survivable network design and iterated rounding 297

12 Further uses of random sampling and randomized rounding of linear programs 309
  12.1 The uncapacitated facility location problem 310
  12.2 The single-source rent-or-buy problem 313
  12.3 The Steiner tree problem 316
  12.4 Everything at once: finding a large cut in a dense graph 322

13 Further uses of randomized rounding of semidefinite programs 333
  13.1 Approximating quadratic programs 334
  13.2 Coloring 3-colorable graphs 340
  13.3 Unique games 344

14 Further uses of the primal-dual method 355
  14.1 The prize-collecting Steiner tree problem 355
  14.2 The feedback vertex set problem in undirected graphs 360

15 Further uses of cuts and metrics 369
  15.1 Low distortion embeddings and the sparsest cut problem 369
  15.2 Oblivious routing and cut-tree packings 376
  15.3 Cut-tree packings and the minimum bisection problem 382
  15.4 The uniform sparsest cut problem 385

16 Techniques in proving the hardness of approximation 407
  16.1 Reductions from NP-complete problems 407
  16.2 Reductions that preserve approximation 412
  16.3 Reductions from probabilistically checkable proofs 420
  16.4 Reductions from label cover 425
  16.5 Reductions from unique games 437

17 Open Problems 447

A Linear programming 453

*Electronic web edition. Copyright 2010 by David P. Williamson and David B. Shmoys.
To be published by Cambridge University Press*
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP-completeness</td>
<td>457</td>
</tr>
<tr>
<td>Bibliography</td>
<td>461</td>
</tr>
<tr>
<td>Author index</td>
<td>481</td>
</tr>
</tbody>
</table>
Part I

An introduction to the techniques
1.1 The whats and whys of approximation algorithms

Decisions, decisions. The difficulty of sifting through large amounts of data in order to make an informed choice is ubiquitous in today’s society. One of the promises of the information technology era is that many decisions can now be made rapidly by computers, from deciding inventory levels, to routing vehicles, to organizing data for efficient retrieval. The study of how to make decisions of these sorts in order to achieve some best possible goal, or objective, has created the field of discrete optimization.

Unfortunately, most interesting discrete optimization problems are NP-hard. Thus unless P = NP, there are no efficient algorithms to find optimal solutions to such problems, where we follow the convention that an efficient algorithm is one that runs in time bounded by a polynomial in its input size. The rest of the book concerns itself with the answer to the question “What should we do in this case?”

An old engineering slogan says “Fast. Cheap. Reliable. Choose two.” Similarly, if P ≠ NP, we can’t simultaneously have algorithms that (1) find optimal solutions (2) in polynomial-time (3) for any instance. At least one of these requirements must be relaxed in any approach to dealing with an NP-hard optimization problem.

One approach relaxes the “for any instance” requirement, and finds polynomial-time algorithms for special cases of the problem at hand. This is useful if the instances one desires to solve fall into one of these special cases, but this is not frequently the case.

A more common approach is to relax the requirement of polynomial-time solvability. The goal is then to find optimal solutions to problems by clever exploration of the full set of possible solutions to a problem. This is often a successful approach if one is willing to take minutes, or even hours, to find the best possible solution; perhaps even more importantly, one is never certain that for the next input encountered, the algorithm will terminate in any reasonable amount of time. This is the approach taken by those in the field of operations research and mathematical programming who solve integer programming formulations of discrete optimization problems, or those in the area of artificial intelligence who consider techniques such as A* search or constraint programming.
By far the most common approach, however, is to relax the requirement of finding an optimal solution, and instead settle for a solution that is “good enough”, especially if it can be found in seconds or less. There has been an enormous study of various types of heuristics and metaheuristics such as simulated annealing, genetic algorithms, and tabu search, to name but a few. These techniques often yield good results in practice.

The approach of this book falls into this third class. We relax the requirement of finding an optimal solution, but our goal is to relax this as little as we possibly can. Through the rest of this book, we will consider approximation algorithms for discrete optimization problems. We try to find a solution that closely approximates the optimal solution in terms of its value. We assume that there is some objective function mapping each possible solution of an optimization problem to some nonnegative value, and an optimal solution to the optimization problem is one that either minimizes or maximizes the value of this objective function. Then we define an approximation algorithm as follows.

Definition 1.1: An \( \alpha \)-approximation algorithm for an optimization problem is a polynomial-time algorithm that for all instances of the problem produces a solution whose value is within a factor of \( \alpha \) of the value of an optimal solution.

For an \( \alpha \)-approximation algorithm, we will call \( \alpha \) the performance guarantee of the algorithm. In the literature, it is also often called the approximation ratio or approximation factor of the algorithm. In this book we will follow the convention that \( \alpha > 1 \) for minimization problems, while \( \alpha < 1 \) for maximization problems. Thus a \( \frac{1}{2} \)-approximation algorithm for a maximization problem is a polynomial-time algorithm that always returns a solution whose value is at least half the optimal value.

Why study approximation algorithms? We list several reasons.

• Because we need algorithms to get solutions to discrete optimization problems. As we mentioned above, with our current information technology there are an increasing number of optimization problems that need to be solved, and most of these are NP-hard. In some cases, an approximation algorithm is a useful heuristic for finding near-optimal solutions when the optimal solution is not required.

• Because algorithm design often focuses first on idealized models rather than the “real-world” application. In practice, many discrete optimization problems are quite messy, and have many complicating side constraints that make it hard to find an approximation algorithm with a good performance guarantee. But often approximation algorithms for simpler versions of the problem give us some idea of how to devise a heuristic that will perform well in practice for the actual problem. Furthermore, the push to prove a theorem often results in a deeper mathematical understanding of the problem’s structure, which then leads to a new algorithmic approach.

• Because it provides a mathematically rigorous basis on which to study heuristics. Typically, heuristics and metaheuristics are studied empirically; they might work well, but we may not understand why. The field of approximation algorithms brings mathematical rigor to the study of heuristics, allowing us to prove how well the heuristic performs on all instances, or giving us some idea of the types of instances on which the heuristic will not perform well. Furthermore, the mathematical analyses of many of the approximation algorithms in this book have the property that not only is there an \textit{a priori} guarantee for any input, but there is also an \textit{a fortiori} guarantee that is provided on an input by input
basis, which allows us to conclude that specific solutions are in fact much more nearly optimal than promised by the performance guarantee.

- **Because it gives a metric for stating how hard various discrete optimization problems are.** Over the course of the twentieth century, the study of the power of computation has steadily evolved. In the early part of the century, researchers were concerned with what kinds of problems could be solved at all by computers in finite time, with the halting problem as the canonical example of a problem that could not be solved. The latter part of the century concerned itself with the efficiency of solution, distinguishing between problems that could be solved in polynomial time, and those that are NP-hard and (perhaps) cannot be solved efficiently. The field of approximation algorithms gives us a means of distinguishing between various optimization problems in terms of how well they can be approximated.

- **Because it’s fun.** The area has developed some very deep and beautiful mathematical results over the years, and it is inherently interesting to study these.

It is sometimes objected that requiring an algorithm to have a near-optimal solution for all instances of the problem – having an analysis for what happens to the algorithm in the worst possible instance – leads to results that are too loose to be practically interesting. After all, in practice, we would greatly prefer solutions within a few percent of optimal rather than, say, twice optimal. From a mathematical perspective, it is not clear that there are good alternatives to this worst-case analysis. It turns out to be quite difficult to define a “typical” instance of any given problem, and often instances drawn randomly from given probability distributions have very special properties not present in real-world data. Since our aim is mathematical rigor in the analysis of our algorithms, we must content ourselves with this notion of worst-case analysis. We note that the worst-case bounds are often due to pathological cases that do not arise in practice, so that approximation algorithms often give rise to heuristics that return solutions much closer to optimal than indicated by their performance guarantees.

Given that approximation algorithms are worth studying, the next natural question is whether there exist good approximation algorithms for problems of interest. In the case of some problems, we are able to obtain extremely good approximation algorithms; in fact, these problems have polynomial-time approximation schemes.

**Definition 1.2:** A polynomial-time approximation scheme (PTAS) is a family of algorithms \( \{A_\epsilon\} \), where there is an algorithm for each \( \epsilon > 0 \), such that \( A_\epsilon \) is a \((1 + \epsilon)\)-approximation algorithm (for minimization problems) or a \((1 - \epsilon)\)-approximation algorithm (for maximization problems).

Many problems have polynomial-time approximation schemes. In later chapters we will encounter the knapsack problem and the Euclidean traveling salesman problem, each of which has a PTAS.

However, there exists a class of problems that is not so easy. This class is called MAX SNP; although we will not define it, it contains many interesting optimization problems, such as the maximum satisfiability problem and the maximum cut problem, which we will discuss later in the book. The following has been shown.

**Theorem 1.3:** For any MAX SNP-hard problem, there does not exist a polynomial-time approximation scheme, unless \( P = NP \).

Finally, some problems are very hard. In the maximum clique problem, we are given as input an undirected graph \( G = (V,E) \). The goal is to find a maximum-size clique; that is, we wish to
find $S \subseteq V$ that maximizes $|S|$ so that for each pair $i, j \in S$, it must be the case that $(i, j) \in E$. The following theorem demonstrates that almost any non-trivial approximation guarantee is most likely unattainable.

**Theorem 1.4:** Let $n$ denote the number of vertices in an input graph, and consider any constant $\epsilon > 0$. Then there does not exist an $O(n^{e-1})$-approximation algorithm for the maximum clique problem, unless $P = NP$.

To see how strong this theorem is, observe that it is very easy to get an $n^{-1}$-approximation algorithm for the problem: just output a single vertex. This gives a clique of size 1, whereas the size of the largest clique can be at most $n$, the number of vertices in the input. The theorem states that finding something only slightly better than this completely trivial approximation algorithm implies that $P = NP$!

### 1.2 An introduction to the techniques and to linear programming: the set cover problem

One of the theses of this book is that there are several fundamental techniques used in the design and analysis of approximation algorithms. The goal of this book is to help the reader understand and master these techniques by applying each technique to many different problems of interest. We will visit some problems several times; when we introduce a new technique, we may see how it applies to a problem we have seen before, and show how we can obtain a better result via this technique. The rest of this chapter will be an illustration of several of the central techniques of the book applied to a single problem, the set cover problem, which we define below. We will see how each of these techniques can be used to obtain an approximation algorithm, and how some techniques lead to improved approximation algorithms for the set cover problem.

In the set cover problem, we are given a ground set of elements $E = \{e_1, \ldots, e_n\}$, some subsets of those elements $S_1, S_2, \ldots, S_m$ where each $S_j \subseteq E$, and a nonnegative weight $w_j \geq 0$ for each subset $S_j$. The goal is to find a minimum-weight collection of subsets that covers all of $E$; that is, we wish to find an $I \subseteq \{1, \ldots, m\}$ that minimizes $\sum_{j \in I} w_j$ subject to $\bigcup_{j \in I} S_j = E$. If $w_j = 1$ for each subset $j$, the problem is called the *unweighted* set cover problem.

The set cover problem is an abstraction of several types of problems; we give two examples here. The set cover problem was used in the development of an antivirus product, which detects computer viruses. In this case it was desired to find salient features that occur in viruses designed for the boot sector of a computer, such that the features do not occur in typical computer applications. These features were then incorporated into another heuristic for detecting these boot sector viruses, a neural network. The elements of the set cover problem were the known boot sector viruses (about 150 at the time). Each set corresponded to some three-byte sequence occurring in these viruses but not in typical computer programs; there were about 21,000 such sequences. Each set contained all the boot sector viruses that had the corresponding three byte sequence somewhere in it. The goal was to find a small number of such sequences (much smaller than 150) that would be useful for the neural network. By using an approximation algorithm to solve the problem, a small set of sequences was found, and the neural network was able to detect many previously unanalyzed boot sector viruses. The set cover problem also generalizes the *vertex cover problem*. In the vertex cover problem, we are given an undirected graph $G = (V, E)$ and a nonnegative weight $w_i \geq 0$ for each vertex $i \in V$. The goal is to find a minimum-weight subset of vertices $C \subseteq V$ such that for each edge $(i, j) \in E$, either $i \in C$ or $j \in C$. As in the set cover problem, if $w_i = 1$ for each vertex $i$,
the problem is an unweighted vertex cover problem. To see that the vertex cover problem is a special case of the set cover problem, for any instance of the vertex cover problem, create an instance of the set cover problem in which the ground set is the set of edges, and a subset $S_i$ of weight $w_i$ is created for each vertex $i \in V$ containing the edges incident to $i$. It is not difficult to see that for any vertex cover $C$, there is a set cover $I = C$ of the same weight, and vice versa.

A second thesis of this book is that linear programming plays a central role in the design and analysis of approximation algorithms. Many of the techniques introduced will use the theory of integer and linear programming in one way or another. Here we will give a very brief introduction to the area in the context of the set cover problem; we give a slightly less brief introduction in Appendix A, and the notes at the end of this chapter provide suggestions of other, more in-depth, introductions to the topic.

Each linear program or integer program is formulated in terms of some number of decision variables that represent some sort of decision that needs to be made. The variables are constrained by a number of linear inequalities and equalities called constraints. Any assignment of real numbers to the variables such that all of the constraints are satisfied is called a feasible solution. In the case of the set cover problem, we need to decide which subsets $S_j$ to use in the solution. We create a decision variable $x_j$ to represent this choice. In this case we would like $x_j$ to be 1 if the set $S_j$ is included in the solution, and 0 otherwise. Thus we introduce constraints $x_j \leq 1$ for all subsets $S_j$, and $x_j \geq 0$ for all subsets $S_j$. This is not sufficient to guarantee that $x_j \in \{0, 1\}$, so we will formulate the problem as an integer program to exclude fractional solutions (that is, non-integral solutions); in this case, we are also allowed to constrain the decision variables to be integers. Requiring $x_j$ to be integer along with the constraints $x_j \geq 0$ and $x_j \leq 1$ is sufficient to guarantee that $x_j \in \{0, 1\}$.

We also want to make sure that any feasible solution corresponds to a set cover, so we introduce additional constraints. In order to ensure that every element $e_i$ is covered, it must be the case that at least one of the subsets $S_j$ containing $e_i$ is selected. This will be the case if

$$\sum_{j : e_i \in S_j} x_j \geq 1,$$

for each $e_i$, $i = 1, \ldots, n$.

In addition to the constraints, linear and integer programs are also defined by a linear function of the decision variables called the objective function. The linear or integer program seeks to find a feasible solution that either maximizes or minimizes this objective function. Such a solution is called an optimal solution. The value of the objective function for a particular feasible solution is called the value of that solution. The value of the objective function for an optimal solution is called the value of the linear (or integer) program. We say we solve the linear program if we find an optimal solution. In the case of the set cover problem we want to find a set cover of minimum weight. Given the decision variables $x_j$ and constraints described above, the weight of a set cover given the $x_j$ variables is $\sum_{j=1}^{m} w_j x_j$. Thus the objective function of the integer program is $\sum_{j=1}^{m} w_j x_j$, and we wish to minimize this function.

Integer and linear programs are usually written in a compact form stating first the objective function, followed by the constraints. Given the discussion above, the problem of finding a
An introduction to approximation algorithms

minimum-weight set cover is equivalent to the following integer program:

\[
\begin{align*}
\text{minimize} & \quad \sum_{j=1}^{m} w_j x_j \\
\text{subject to} & \quad \sum_{j: e_i \in S_j} x_j \geq 1, \quad i = 1, \ldots, n, \\
& \quad x_j \in \{0, 1\}, \quad j = 1, \ldots, m.
\end{align*}
\] (1.1)

Let \( Z_{IP}^* \) denote the optimum value of this integer program for a given instance of the set cover problem. Since the integer program exactly models the problem, we have that \( Z_{IP}^* = \text{OPT} \), where \( \text{OPT} \) is the value of an optimum solution to the set cover problem.

In general, integer programs cannot be solved in polynomial time. This is clear because the set cover problem is NP-hard, and so solving the integer program above for any set cover input in polynomial time would imply that \( P = \text{NP} \). However, linear programs are polynomial-time solvable. In linear programs we are not allowed to require that decision variables are integers. Nevertheless, linear programs are still extremely useful: even in cases such as the set cover problem, we are still able to derive useful information from linear programs. For instance, if we replace the constraints \( x_j \in \{0, 1\} \) with the constraints \( x_j \geq 0 \), we obtain the following linear program, which can be solved in polynomial time:

\[
\begin{align*}
\text{minimize} & \quad \sum_{j=1}^{m} w_j x_j \\
\text{subject to} & \quad \sum_{j: e_i \in S_j} x_j \geq 1, \quad i = 1, \ldots, n, \\
& \quad x_j \geq 0, \quad j = 1, \ldots, m.
\end{align*}
\] (1.2)

We could also add the constraints \( x_j \leq 1 \), for each \( j = 1, \ldots, m \), but they would be redundant: in any optimal solution to the problem, we can reduce any \( x_j > 1 \) to \( x_j = 1 \) without affecting the feasibility of the solution and without increasing its cost.

The linear program (1.2) is a relaxation of the original integer program. By this we mean two things: first, every feasible solution for the original integer program (1.1) is feasible for this linear program; and second, the value of any feasible solution for the integer program has the same value in the linear program. To see that the linear program is a relaxation, note that any solution for the integer program such that \( x_j \in \{0, 1\} \) for each \( j = 1, \ldots, m \) and \( \sum_{j: e_i \in S_j} x_j \geq 1 \) for each \( i = 1, \ldots, m \) will certainly satisfy all the constraints of the linear program. Furthermore the objective functions of both the integer and linear programs are the same, so that any feasible solution for the integer program has the same value for the linear program. Let \( Z_{LP}^* \) denote the optimum value of this linear program. Any optimal solution to the integer program is feasible for the linear program and has value \( Z_{IP}^* \). Thus, any optimal solution to the linear program will have value \( Z_{LP}^* \leq Z_{IP}^* = \text{OPT} \), since this minimization linear program finds a feasible solution of lowest possible value. Using a polynomial-time solvable relaxation of a problem in order to obtain a lower bound (in the case of minimization problems) or an upper bound (in the case of maximization problems) on the optimum value of the problem is a concept that will appear frequently in this book.

In the following sections, we will give some examples of how the linear programming relaxation can be used to derive approximation algorithms for the set cover problem. In the next section, we will show that a fractional solution to the linear program can be rounded to
1.3 A deterministic rounding algorithm

Suppose that we solve the linear programming relaxation of the set cover problem. Let \( x^* \) denote an optimal solution to the LP. How then can we recover a solution to the set cover problem? Here is a very easy way to obtain a solution: given the LP solution \( x^* \), we include subset \( S_j \) in our solution if and only if \( x^*_j \geq 1/f \), where \( f \) is the maximum number of sets in which any element appears. More formally, let \( f_i = \left| \{ j : e_i \in S_j \} \right| \) be the number of sets in which element \( e_i \) appears, \( i = 1, \ldots, n \); then \( f = \max_{i=1,\ldots,n} f_i \). Let \( I \) denote the indices \( j \) of the subsets in this solution. In effect, we round the fractional solution \( x^* \) to an integer solution \( \hat{x} \) by setting \( \hat{x}_j = 1 \) if \( x^*_j \geq 1/f \), and \( \hat{x}_j = 0 \) otherwise. We shall see that it is straightforward to prove that \( \hat{x} \) is a feasible solution to the integer program, and \( I \) indeed indexes a set cover.

**Lemma 1.5:** The collection of subsets \( S_j, j \in I \), is a set cover.

**Proof.** Consider the solution specified by the lemma, and call an element \( e_i \) covered if this solution contains some subset containing \( e_i \). We show that each element \( e_i \) is covered. Because the optimal solution \( x^* \) is a feasible solution to the linear program, we know that \( \sum_{j : e_i \in S_j} x^*_j \geq 1 \) for element \( e_i \). By the definition of \( f_i \) and of \( f \), there are \( f_i \leq f \) terms in the sum, so at least one term must be at least \( 1/f \). Thus for some \( j \) such that \( e_i \in S_j, x^*_j \geq 1/f \). Therefore \( j \in I \), and element \( e_i \) is covered. \( \Box \)

We can also show that this rounding procedure yields an approximation algorithm.

**Theorem 1.6:** The rounding algorithm is an \( f \)-approximation algorithm for the set cover problem.

**Proof.** It is clear that the algorithm runs in polynomial time. By our construction, \( 1 \leq f \cdot x^*_j \) for each \( j \in I \). From this, and the fact that each term \( f w_j x^*_j \) is nonnegative for \( j = 1, \ldots, m \),
we see that
\[ \sum_{j \in I} w_j \leq \sum_{j=1}^{m} w_j \cdot (f \cdot x^*_j) \]
\[ = f \sum_{j=1}^{m} w_j x^*_j \]
\[ = f \cdot Z^*_LP \]
\[ \leq f \cdot \text{OPT}, \]
where the final inequality follows from the argument above that \( Z^*_LP \leq \text{OPT}. \)

In the special case of the vertex cover problem, \( f_i = 2 \) for each vertex \( i \in V \), since each edge is incident to exactly two vertices. Thus the rounding algorithm gives a 2-approximation algorithm for the vertex cover problem.

This particular algorithm allows us to have an \textit{a fortiori} guarantee for each input. While we know that for any input, the solution produced has cost at most a factor of \( f \) more than the cost of an optimal solution, we can for any input compare the value of the solution we find with the value of the linear programming relaxation. If the algorithm finds a set cover \( I \), let \( \alpha = \sum_{j \in I} w_j / Z^*_LP \). From the proof above, we know that \( \alpha \leq f \). However, for any given input, it could be the case that \( \alpha \) is significantly smaller than \( f \); in this case we know that \( \sum_{j \in I} w_j = \alpha Z^*_LP \leq \alpha \text{OPT} \), and the solution is within a factor of \( \alpha \) of optimal. The algorithm can easily compute \( \alpha \), given that it computes \( I \) and solves the LP relaxation.

\[ \textbf{1.4 Rounding a dual solution} \]

Often it will be useful to consider the dual of the linear programming relaxation of a given problem. Again, we will give a very brief introduction to the concept of the dual of a linear program in the context of the set cover problem, and more in-depth introductions to the topic will be cited in the notes at the end of this chapter.

To begin, we suppose that each element \( e_i \) is charged some nonnegative price \( y_i \geq 0 \) for its coverage by a set cover. Intuitively, it might be the case that some elements can be covered with low-weight subsets, while other elements might require high-weight subsets to cover them; we would like to be able to capture this distinction by charging low prices to the former and high prices to the latter. In order for the prices to be reasonable, it cannot be the case that the sum of the prices of elements in a subset \( S_j \) is more than the weight of the set, since we are able to cover all of those elements by paying weight \( w_j \). Thus for each subset \( S_j \) we have the following limit on the prices:

\[ \sum_{i : e_i \in S_j} y_i \leq w_j. \]

We can find the highest total price that the elements can be charged by the following linear program:

\[ \text{maximize} \quad \sum_{i=1}^{n} y_i \]
\[ \text{subject to} \quad \sum_{i : e_i \in S_j} y_i \leq w_j, \quad j = 1, \ldots, m, \tag{1.3} \]
\[ y_i \geq 0, \quad i = 1, \ldots, n. \]
This linear program is the dual linear program of the set cover linear programming relaxation (1.2). We can in general derive a dual linear program for any given linear program, but we will not go into the details of how to do so; see Appendix A or the references in the notes at the end of the chapter. If we derive a dual for a given linear program, the given program is sometimes called the primal linear program. For instance, the original linear programming relaxation (1.2) of the set cover problem is the primal linear program of the dual (1.3). Notice that the dual above has a variable \( y_i \) for each constraint of the primal linear program (that is, for the constraint \( \sum_{j : e_i \in S_j} x_j \geq 1 \)), and has a constraint for each variable \( x_j \) of the primal. This is true of dual linear programs in general.

Dual linear programs have a number of very interesting and useful properties. For example, let \( x \) be any feasible solution to the set cover linear programming relaxation, and let \( y \) be any feasible set of prices (that is, any feasible solution to the dual linear program). Then consider the value of the dual solution \( y \):

\[
\sum_{i=1}^{n} y_i \leq \sum_{i=1}^{n} y_i \sum_{j : e_i \in S_j} x_j,
\]

since for any \( e_i \), \( \sum_{j : e_i \in S_j} x_j \geq 1 \) by the feasibility of \( x \). Then rewriting the right-hand side of the inequality above, we have

\[
\sum_{i=1}^{n} y_i \sum_{j : e_i \in S_j} x_j = \sum_{j=1}^{m} x_j \sum_{i : e_i \in S_j} y_i.
\]

Finally, noticing that since \( y \) is a feasible solution to the dual linear program, we know that \( \sum_{i : e_i \in S_j} y_i \leq w_j \) for any \( j \), so that

\[
\sum_{j=1}^{m} x_j \sum_{i : e_i \in S_j} y_i \leq \sum_{j=1}^{m} x_j w_j.
\]

So we have shown that

\[
\sum_{i=1}^{n} y_i \leq \sum_{j=1}^{m} w_j x_j;
\]

that is, any feasible solution to the dual linear program has a value no greater than any feasible solution to the primal linear program. In particular, any feasible solution to the dual linear program has a value no greater than the optimal solution to the primal linear program, so for any feasible \( y \), \( \sum_{i=1}^{n} y_i \leq Z_{LP}^* \). This is called the weak duality property of linear programs. Since we previously argued that \( Z_{LP}^* \leq \text{OPT} \), we have that for any feasible \( y \), \( \sum_{i=1}^{n} y_i \leq \text{OPT} \).

This is a very useful property that will help us in designing approximation algorithms.

Additionally, there is a quite amazing strong duality property of linear programs. Strong duality states that as long as there exist feasible solutions to both the primal and dual linear programs, their optimal values are equal. Thus if \( x^* \) is an optimal solution to the set cover linear programming relaxation, and \( y^* \) is an optimal solution to the dual linear program, then

\[
\sum_{j=1}^{m} w_j x_j^* = \sum_{i=1}^{n} y_i^*.
\]
Information from a dual linear program solution can sometimes be used to derive good approximation algorithms. Let $y^*$ be an optimal solution to the dual LP (1.3), and consider the solution in which we choose all subsets for which the corresponding dual inequality is tight; that is, the inequality is met with equality for subset $S_j$, and $\sum_{i:e_i \in S_j} y_i^* = w_j$. Let $I'$ denote the indices of the subsets in this solution. We will prove that this algorithm also is an $f$-approximation algorithm for the set cover problem.

**Lemma 1.7:** The collection of subsets $S_j$, $j \in I'$, is a set cover.

**Proof.** Suppose that there exists some uncovered element $e_k$. Then for each subset $S_j$ containing $e_k$, it must be the case that
\[
\sum_{i:e_i \in S_j} y_i^* < w_j. \tag{1.4}
\]
Let $\epsilon$ be the smallest difference between the right-hand side and left-hand side of all constraints involving $e_k$; that is, $\epsilon = \min_{j:e_i \in S_j} \left( w_j - \sum_{i:e_i \in S_j} y_i^* \right)$. By inequality (1.4), we know that $\epsilon > 0$.

Consider now a new dual solution $y'$ in which $y'_k = y_k^* + \epsilon$ and every other component of $y'$ is the same as in $y^*$. Then $y'$ is a dual feasible solution since for each $j$ such that $e_k \in S_j$,
\[
\sum_{i:e_i \in S_j} y'_i = \sum_{i:e_i \in S_j} y_i^* + \epsilon \leq w_j,
\]
by the definition of $\epsilon$. For each $j$ such that $e_k \notin S_j$,
\[
\sum_{i:e_i \in S_j} y'_i = \sum_{i:e_i \in S_j} y_i^* \leq w_j,
\]
as before. Furthermore, $\sum_{i=1}^n y'_i > \sum_{i=1}^n y_i^*$, which contradicts the optimality of $y^*$. Thus it must be the case that all elements are covered and $I'$ is a set cover.

**Theorem 1.8:** The dual rounding algorithm described above is an $f$-approximation algorithm for the set cover problem.

**Proof.** The central idea is the following “charging” argument: when we choose a set $S_j$ to be in the cover we “pay” for it by charging $y_i^*$ to each of its elements; each element is charged at most once for each set that contains it (and hence at most $f$ times), and so the total cost is at most $f \sum_{i=1}^n y_i^*$, or $f$ times the dual objective function.

More formally, since $j \in I'$ only if $w_j = \sum_{i:e_i \in S_j} y_i^*$, we have that the cost of the set cover $I'$ is
\[
\sum_{j \in I'} w_j = \sum_{j \in I'} \sum_{i:e_i \in S_j} y_i^* \\
= \sum_{i=1}^n \left| \{ j \in I' : e_i \in S_j \} \right| \cdot y_i^* \\
\leq \sum_{i=1}^n f y_i^* \\
\leq f \sum_{i=1}^n y_i^* \\
\leq f \cdot \text{OPT}.
\]
The second equality follows from the fact that when we interchange the order of summation, the coefficient of $y^*_i$ is, of course, equal to the number of times that this term occurs overall. The final inequality follows from the weak duality property discussed previously.

In fact, it is possible to show that this algorithm can do no better than the algorithm of the previous section; to be precise, we can show that if $I$ indexes the solution returned by the primal rounding algorithm of the previous section, then $I \subseteq I'$. This follows from a property of optimal linear programming solutions called complementary slackness. We showed earlier the following string of inequalities for any feasible solution $x$ to the set cover linear programming relaxation, and any feasible solution $y$ to the dual linear program:

$$
\sum_{i=1}^{n} y_i \leq \sum_{i=1}^{n} y_i \sum_{j \in S_j} x_j = \sum_{j=1}^{m} x_j \sum_{i \in S_j} y_i \leq \sum_{j=1}^{m} x_j w_j.
$$

Furthermore, we claimed that strong duality implies that for optimal solutions $x^*$ and $y^*$, $\sum_{i=1}^{n} y^*_i = \sum_{j=1}^{m} w_j x^*_j$. Thus for any optimal solutions $x^*$ and $y^*$, the two inequalities in the chain of inequalities above must in fact be equalities. This can only happen if whenever $y^*_i > 0$ then $\sum_{j \in S_j} x^*_j = 1$, and whenever $x^*_j > 0$, then $\sum_{i \in S_j} y^*_i = w_j$. That is, whenever a linear programming variable (primal or dual) is nonzero, the corresponding constraint in the dual or primal is tight. These conditions are known as the complementary slackness conditions. Thus if $x^*$ and $y^*$ are optimal solutions, the complementary slackness conditions must hold. The converse is also true: if $x^*$ and $y^*$ are feasible primal and dual solutions respectively, then if the complementary slackness conditions hold, the values of the two objective functions are equal and therefore the solutions must be optimal.

In the case of the set cover program, if $x^*_j > 0$ for any primal optimal solution $x^*$, then the corresponding dual inequality for $S_j$ must be tight for any dual optimal solution $y^*$. Recall that in the algorithm of the previous section, we put $j \in I$ when $x^*_j \geq 1/f$. Thus $j \in I$ implies that $j \in I'$, so that $I' \subseteq I$.

### 1.5 Constructing a dual solution: the primal-dual method

One of the disadvantages of the algorithms of the previous two sections is that they require solving a linear program. While linear programs are efficiently solvable, and algorithms for them are quick in practice, special purpose algorithms are often much faster. Although in this book we will not usually be concerned with the precise running times of the algorithms, we will try to indicate their relative practicality.

The basic idea of the algorithm in this section is that the dual rounding algorithm of the previous section uses relatively few properties of an optimal dual solution. Instead of actually solving the dual LP, we can construct a feasible dual solution with the same properties. In this case, constructing the dual solution is much faster than solving the dual LP, and hence leads to a much faster algorithm.

The algorithm of the previous section used the following properties. First, we used the fact that $\sum_{i=1}^{n} y_i \leq \text{OPT}$, which is true for any feasible dual solution $y$. Second, we include $j \in I'$ precisely when $\sum_{i \in S_j} y_i = w_j$, and $I'$ is a set cover. These two facts together gave the proof that the cost of $I'$ is no more than $f$ times optimal.

Importantly, it is the proof of Lemma 1.7 (that we have constructed a feasible cover) that shows how to obtain an algorithm that constructs a dual solution. Consider any feasible dual solution $y$, and let $T$ be the set of the indices of all tight dual constraints; that is, $T = \{ j :
An introduction to approximation algorithms

Algorithm 1.1: Primal-dual algorithm for the set cover problem.

\[
y \leftarrow 0 \\
I \leftarrow \emptyset \\
\textbf{while} \; \text{there exists } e_i \notin \bigcup_{j \in I} S_j \; \textbf{do} \\
\quad \text{Increase the dual variable } y_i \text{ until there is some } \ell \text{ with } e_i \in S_\ell \text{ such that } \\
\quad \quad \sum_{j : e_j \in S_\ell} y_j = w_\ell \\
\quad I \leftarrow I \cup \{\ell\}
\]

This yields the following theorem.

**Theorem 1.9:** Algorithm 1.1 is an \( f \)-approximation algorithm for the set cover problem.

This type of algorithm is called a primal-dual algorithm by analogy with the primal-dual method used in other combinatorial algorithms. Linear programming problems, network flow problems, and shortest path problems (among others) all have primal-dual optimization algorithms; we will see an example of a primal-dual algorithm for the shortest s-t path problem in Section 7.3. Primal-dual algorithms start with a dual feasible solution, and use dual information to infer a primal, possibly infeasible, solution. If the primal solution is indeed infeasible, the dual solution is modified to increase the value of the dual objective function. The primal-dual method has been very useful in designing approximation algorithms, and we will discuss it extensively in Chapter 7.

We observe again that this particular algorithm allows us to have an a fortiori guarantee for each input, since we can compare the value of the solution obtained with the value of the dual solution generated by the algorithm. This ratio is guaranteed to be at most \( f \) by the proof above, but might be significantly better.

### 1.6 A greedy algorithm

At this point, the reader might be forgiven for feeling a slight sense of futility: we have examined several techniques for designing approximation algorithms for the set cover problem, and they have all led to the same result, an approximation algorithm with performance guarantee \( f \). But, as in life, perseverance and some amount of cleverness often pay dividends in designing approximation algorithms. We show in this section that a type of algorithm called a greedy algorithm gives an approximation algorithm with a performance guarantee that is often significantly better than \( f \). Greedy algorithms work by making a sequence of decisions; each decision is made to optimize that particular decision, even though this sequence of locally optimal (or “greedy”) decisions might not lead to a globally optimal solution. The advantage of greedy
algorithms is that they are typically very easy to implement, and hence greedy algorithms are a commonly used heuristic, even when they have no performance guarantee.

We now present a very natural greedy algorithm for the set cover problem. Sets are chosen in a sequence of rounds. In each round, we choose the set that gives us the most “bang for the buck”; that is, the set that minimizes the ratio of its weight to the number of currently uncovered elements it contains. In the event of a tie, we pick an arbitrary set that achieves the minimum ratio. We continue choosing sets until all elements are covered. Obviously, this will yield a polynomial-time algorithm, since there can be no more than $m$ rounds, and in each we compute $O(m)$ ratios, each in constant time. A formal description is given in Algorithm 1.2.

Before we state the theorem, we need some notation and a useful mathematical fact. Let $H_k$ denote the $k$th harmonic number: that is, $H_k = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{k}$. Note that $H_k \approx \ln k$.

The following fact is one that we will use many times in the course of this book. It can be proven with simple algebraic manipulations.

**Fact 1.10**: Given positive numbers $a_1, \ldots, a_k$ and $b_1, \ldots, b_k$, then

$$
\min_{i=1,\ldots,k} \frac{a_i}{b_i} \leq \frac{\sum_{i=1}^{k} a_i}{\sum_{i=1}^{k} b_i} \leq \max_{i=1,\ldots,k} \frac{a_i}{b_i}.
$$

**Theorem 1.11**: Algorithm 1.2 is an $H_n$-approximation algorithm for the set cover problem.

**Proof.** The basic intuition for the analysis of the algorithm is as follows. Let OPT denote the value of an optimal solution to the set cover problem. We know that an optimal solution covers all $n$ elements with a solution of weight OPT; therefore there must be some subset that covers its elements with an average weight of at most $OPT/n$. Similarly, after $k$ elements have been covered, the optimal solution can cover the remaining $n-k$ elements with a solution of weight OPT, which implies that there is some subset which covers its remaining uncovered elements with an average weight of at most $OPT/(n-k)$. So in general the greedy algorithm pays about $OPT/(n-k+1)$ to cover the $k$th uncovered element, giving a performance guarantee of $\sum_{k=1}^{n} \frac{1}{n-k+1} = H_n$.

We now formalize this intuition. Let $n_k$ denote the number of elements that remain uncovered at the start of the $k$th iteration. If the algorithm takes $\ell$ iterations, then $n_1 = n$, and we set $n_{\ell+1} = 0$. Pick an arbitrary iteration $k$. Let $I_k$ denote the indices of the sets chosen in iterations 1 through $k-1$, and for each $j = 1, \ldots, m$, let $\hat{S}_j$ denote the set of uncovered elements in $S_j$ at the start of this iteration; that is, $\hat{S}_j = S_j - \bigcup_{\ell \in I_k} S_{\ell}$. Then we claim that for the set $j$ chosen in the $k$th iteration,

$$
w_j \leq \frac{n_k - n_{k+1}}{n_k} \text{OPT}.
$$

Electronic web edition. Copyright 2010 by David P. Williamson and David B. Shmoys.
To be published by Cambridge University Press
Given the claimed inequality (1.5), we can prove the theorem. Let $I$ contain the indices of the sets in our final solution. Then

\[
\sum_{j \in I} w_j \leq \sum_{k=1}^{\ell} \frac{n_k - n_{k+1}}{n_k} \text{OPT}
\]

\[
\leq \text{OPT} \cdot \sum_{k=1}^{\ell} \left( \frac{1}{n_k} + \frac{1}{n_k - 1} + \cdots + \frac{1}{n_{k+1} + 1} \right)
\]

\[
= \text{OPT} \cdot \sum_{i=1}^{n_k} \frac{1}{i}
\]

\[
= H_n \cdot \text{OPT},
\]

where the inequality (1.6) follows from the fact that $\frac{1}{n_k} \leq \frac{1}{n_{k-1}}$ for each positive $i$.

To prove the claimed inequality (1.5), we shall first argue that in the $k$th iteration,

\[
\min_{j: \hat{S}_j \neq \emptyset} \frac{w_j}{|\hat{S}_j|} \leq \frac{\text{OPT}}{n_k}.
\]

(1.7)

If we let $O$ contain the indices of the sets in an optimal solution, then inequality (1.7) follows from Fact 1.10, by observing that

\[
\min_{j: \hat{S}_j \neq \emptyset} \frac{w_j}{|\hat{S}_j|} \leq \frac{\sum_{j \in O} w_j}{\sum_{j \in O} |\hat{S}_j|} = \frac{\text{OPT}}{\sum_{j \in O} |\hat{S}_j|} \leq \frac{\text{OPT}}{n_k},
\]

where the last inequality follows from the fact that since $O$ is a set cover, the set $\bigcup_{j \in O} \hat{S}_j$ must include all remaining $n_k$ uncovered elements. Let $j$ index a subset that minimizes this ratio, so that $\frac{w_j}{|\hat{S}_j|} \leq \frac{\text{OPT}}{n_k}$. If we add the subset $S_j$ to our solution, then there will be $|\hat{S}_j|$ fewer uncovered elements, so that $n_{k+1} = n_k - |\hat{S}_j|$. Thus

\[
w_j \leq \frac{|\hat{S}_j| \cdot \text{OPT}}{n_k} = \frac{n_k - n_{k+1}}{n_k} \cdot \text{OPT}.
\]

We can improve the performance guarantee of the algorithm slightly by using the dual of the linear programming relaxation in the analysis. Let $g$ be the maximum size of any subset $S_j$; that is, $g = \max_j |S_j|$. Recall that $Z_{LP}^*$ is the optimum value of the linear programming relaxation for the set cover problem. The following theorem immediately implies that the greedy algorithm is an $H_g$-approximation algorithm, since $Z_{LP}^* \leq \text{OPT}$.

**Theorem 1.12:** Algorithm 1.2 returns a solution indexed by $I$ such that $\sum_{j \in I} w_j \leq H_g \cdot Z_{LP}^*$.

**Proof.** To prove the theorem, we will construct an infeasible dual solution $y$ such that $\sum_{j \in I} w_j = \sum_{i=1}^{n} y_i$. We will then show that $y' = \frac{1}{H_g} y$ is a feasible dual solution. By the weak duality theorem, $\sum_{i=1}^{n} y'_i \leq Z_{LP}^*$, so that $\sum_{j \in I} w_j = \sum_{i=1}^{n} y_i = H_g \sum_{i=1}^{n} y'_i \leq H_g \cdot \text{OPT}$. We will see at the end of the proof the reason we choose to divide the infeasible dual solution $y$ by $H_g$.

The name dual fitting has been given to this technique of constructing an infeasible dual solution whose value is equal to the value of the primal solution constructed, and such that

**Electronic web edition. Copyright 2010 by David P. Williamson and David B. Shmoys. To be published by Cambridge University Press**
scaling the dual solution by a single value makes it feasible. We will return to this technique in Section 9.4.

To construct the infeasible dual solution \( y \), suppose we choose to add subset \( S_j \) to our solution in iteration \( k \). Then for each \( e_i \in S_j \), we set \( y_i = w_j/|S_j| \). Since each \( e_i \in S_j \) is uncovered in iteration \( k \), and is then covered for the remaining iterations of the algorithm (because we added subset \( S_j \) to the solution), the dual variable \( y_i \) is set to a value exactly once; in particular, it is set in the iteration in which element \( e_i \) is covered. Furthermore, \( w_j = \sum_{i \in e_i \in S_j} y_i \); that is, the weight of the subset \( S_j \) chosen in the \( k \)th iteration is equal to the sum of the duals \( y_i \) of the uncovered elements that are covered in the \( k \)th iteration. This immediately implies that \( \sum_{j \in I} w_j = \sum_{i=1}^n y_i \).

It remains to prove that the dual solution \( y' = \frac{1}{H_g} y \) is feasible. We must show that for each subset \( S_j \), \( \sum_{i : e_i \in S_j} y'_i \leq w_j \). Pick an arbitrary subset \( S_j \). Let \( a_k \) be the number of elements in this subset that are still uncovered at the beginning of the \( k \)th iteration, so that \( a_1 = |S_j| \), and \( a_{k+1} = 0 \). Let \( A_k \) be the uncovered elements of \( S_j \) covered in the \( k \)th iteration, so that \( |A_k| = a_k - a_{k+1} \). If subset \( S_p \) is chosen in the \( k \)th iteration, then for each element \( e_i \in A_k \) covered in the \( k \)th iteration,

\[
y'_i = \frac{w_p}{H_g |S_p|} \leq \frac{w_j}{H_g a_k},
\]

where \( \hat{S}_p \) is the set of uncovered elements of \( S_p \) at the beginning of the \( k \)th iteration. The inequality follows because if \( S_p \) is chosen in the \( k \)th iteration, it must minimize the ratio of its weight to number of uncovered elements it contains. Thus

\[
\sum_{i : e_i \in S_j} y'_i = \sum_{k=1}^{\ell} \sum_{i : e_i \in A_k} y'_i \\
\leq \sum_{k=1}^{\ell} (a_k - a_{k+1}) \frac{w_j}{H_g a_k} \\
\leq \frac{w_j}{H_g} \sum_{k=1}^{\ell} \frac{a_k - a_{k+1}}{a_k} \\
\leq \frac{w_j}{H_g} \sum_{k=1}^{\ell} \left( \frac{1}{a_k} + \frac{1}{a_k - 1} + \cdots + \frac{1}{a_{k+1} + 1} \right) \\
\leq \frac{w_j}{H_g} \sum_{i=1}^{\ell} \frac{1}{i} \\
= \frac{w_j}{H_g} |S_j| \\
\leq w_j,
\]

where the final inequality follows because \( |S_j| \leq g \). Here we see the reason for scaling the dual solution by \( H_g \), since we know that \( H_{|S_j|} \leq H_g \) for all sets \( j \).

It turns out that no approximation algorithm for the set cover problem with performance guarantee better than \( H_n \) is possible, under an assumption slightly stronger that \( \mathcal{P} \neq \mathsf{NP} \).

**Theorem 1.13:** If there exists a \( c \ln n \)-approximation algorithm for the unweighted set cover problem for some constant \( c < 1 \), then there is an \( O(n^{O(\log \log n)}) \)-time deterministic algorithm for each NP-complete problem.
Theorem 1.14: There exists some constant $c > 0$ such that if there exists a $c \ln n$-approximation algorithm for the unweighted set cover problem, then $P = NP$.

We will discuss results of this sort at more length in Chapter 16; in Theorem 16.32 we show how a slightly weaker version of these results can be derived. Results of this type are sometimes called hardness theorems, as they show that it is NP-hard to provide near-optimal solutions for a certain problem with certain performance guarantees.

The $f$-approximation algorithms for the set cover problem imply a 2-approximation algorithm for the special case of the vertex cover problem. No algorithm with a better constant performance guarantee is known at this point in time. Additionally, two hardness theorems, Theorems 1.15 and 1.16 below, have been shown.

Theorem 1.15: If there exists an $\alpha$-approximation algorithm for the vertex cover problem with $\alpha < 10\sqrt{5} - 21 \approx 1.36$, then $P = NP$.

The following theorem mentions a conjecture called the unique games conjecture that we will discuss more in Section 13.3 and Section 16.5. The conjecture is roughly that a particular problem (called unique games) is NP-hard.

Theorem 1.16: Assuming the unique games conjecture, if there exists an $\alpha$-approximation algorithm for the vertex cover problem with constant $\alpha < 2$, then $P = NP$.

Thus, assuming $P \neq NP$ and the NP-completeness of the unique games problem, we have found essentially the best possible approximation algorithm for the vertex cover problem.

1.7 A randomized rounding algorithm

In this section, we consider one final technique for devising an approximation algorithm for the set cover problem. Although the algorithm is slower and has no better guarantee than the greedy algorithm of the previous section, we include it here because it introduces the notion of using randomization in approximation algorithms, an idea we will cover in depth in Chapter 5.

As with the algorithm in Section 1.3, the algorithm will solve a linear programming relaxation for the set cover problem, and then round the fractional solution to an integral solution. Rather than doing so deterministically, however, the algorithm will do so randomly using a technique called randomized rounding. Let $x^*$ be an optimal solution to the LP relaxation. We would like to round fractional values of $x^*$ to either 0 or 1 in such a way that we obtain a solution $\hat{x}$ to the integer programming formulation of the set cover problem without increasing the cost too much. The central idea of randomized rounding is that we interpret the fractional value $x^*_j$ as the probability that $\hat{x}_j$ should be set to 1. Thus each subset $S_j$ is included in our solution with probability $x^*_j$, where these $m$ events (that $S_j$ is included in our solution) are independent random events. We assume some basic knowledge of probability theory throughout this text; for those who need some additional background, see the notes at the end of the chapter for suggested references.

Let $X_j$ be a random variable that is 1 if subset $S_j$ is included in the solution, and 0 otherwise. Then the expected value of the solution is

$$E \left[ \sum_{j=1}^{m} w_j X_j \right] = \sum_{j=1}^{m} w_j \Pr[X_j = 1] = \sum_{j=1}^{m} w_j x^*_j = Z^*_{LP},$$

or just the value of the linear programming relaxation, which is no more than OPT! As we will see below, however, it is quite likely that the solution is not a set cover. Nevertheless, this
A randomized rounding algorithm illustrates why randomized rounding can provide such good approximation algorithms in some cases, and we will see further examples of this in Chapter 5.

Let us now calculate the probability that a given element $e_i$ is not covered by this procedure. This is the probability that none of the subsets containing $e_i$ are included in the solution, or

$$\prod_{j : e_i \in S_j} (1 - x_j^*).$$

We can bound this probability by using the fact that $1 - x \leq e^{-x}$ for any $x$, where $e$ is the base of the natural logarithm. Then

$$\Pr[e_i \text{ not covered}] = \prod_{j : e_i \in S_j} (1 - x_j^*)$$

$$\leq \prod_{j : e_i \in S_j} e^{-x_j^*}$$

$$= e^{-\sum_{j : e_i \in S_j} x_j^*}$$

$$\leq e^{-1},$$

where the final inequality follows from the LP constraint that $\sum_{j : e_i \in S_j} x_j^* \geq 1$. Although $e^{-1}$ is an upper bound on the probability that a given element is not covered, it is possible to approach this bound arbitrarily closely, so in the worst case it is quite likely that this randomized rounding procedure does not produce a set cover.

How small would this probability have to be in order for it to be very likely that a set cover is produced? And perhaps even more fundamentally, what is the “right” notion of “very likely”? The latter question has a number of possible answers; one natural way to think of the situation is to impose a guarantee in keeping with our focus on polynomial-time algorithms. Suppose that, for any constant $c$, we could devise a polynomial-time algorithm whose chance of failure is at most an inverse polynomial $n^{-c}$; then we say that we have an algorithm that works with high probability. To be more precise, we would have a family of algorithms, since it might be necessary to give progressively slower algorithms, or ones with worse performance guarantees, to achieve analogously more fail-safe results. If we could devise a randomized procedure such that $\Pr[e_i \text{ not covered}] \leq \frac{1}{n^c}$ for some constant $c \geq 2$, then

$$\Pr[\text{there exists an uncovered element}] \leq \sum_{i=1}^{n} \Pr[e_i \text{ not covered}] \leq \frac{1}{n^{c-1}},$$

and we would have a set cover with high probability. In fact, we can achieve such a bound in the following way: for each subset $S_j$, we imagine a coin that comes up heads with probability $x_j^*$, and we flip the coin $c \ln n$ times. If it comes up heads in any of the $c \ln n$ trials, we include $S_j$ in our solution, otherwise not. Thus the probability that $S_j$ is not included is $(1 - x_j^*)^{c \ln n}$. Furthermore,

$$\Pr[e_i \text{ not covered}] = \prod_{j : e_i \in S_j} (1 - x_j^*)^{c \ln n}$$

$$\leq \prod_{j : e_i \in S_j} e^{-x_j^* (c \ln n)}$$

$$= e^{-(c \ln n) \sum_{j : e_i \in S_j} x_j^*}$$

$$\leq \frac{1}{n^c},$$

Electronic web edition. Copyright 2010 by David P. Williamson and David B. Shmoys. To be published by Cambridge University Press
as desired.

We now only need to prove that the algorithm has a good expected value given that it produces a set cover.

**Theorem 1.17:** The algorithm is a randomized $O(\ln n)$-approximation algorithm that produces a set cover with high probability.

**Proof.** Let $p_j(x_j^*)$ be the probability that a given subset $S_j$ is included in the solution as a function of $x_j^*$. By construction of the algorithm, we know that $p_j(x_j^*) = 1 - (1 - x_j^*)^{c\ln n}$. Observe that if $x_j^* \in [0, 1]$ and $c\ln n \geq 1$, then we can bound the derivative $p'_j$ at $x_j^*$ by

$$p'_j(x_j^*) = (c\ln n)(1 - x_j^*)^{(c\ln n)-1} \leq (c\ln n).$$

Then since $p_j(0) = 0$, and the slope of the function $p_j$ is bounded above by $c\ln n$ on the interval $[0, 1]$, $p_j(x_j^*) \leq (c\ln n)x_j^*$ on the interval $[0, 1]$. If $X_j$ is a random variable that is 1 if the subset $S_j$ is included in the solution, and 0 otherwise, then the expected value of the random procedure is

$$E \left[ \sum_{j=1}^{m} w_j X_j \right] = \sum_{j=1}^{m} w_j \Pr[X_j = 1]$$

$$\leq \sum_{j=1}^{m} w_j (c\ln n)x_j^*$$

$$= (c\ln n) \sum_{j=1}^{m} w_j x_j^* = (c\ln n)Z_{LP}^*.$$

However, we would like to bound the expected value of the solution given that a set cover is produced. Let $F$ be the event that the solution obtained by the procedure is a feasible set cover, and let $\bar{F}$ be the complement of this event. We know from the previous discussion that $\Pr[F] \geq 1 - \frac{1}{ne^{-1}}$, and we also know that

$$E \left[ \sum_{j=1}^{m} w_j X_j \bigg| F \right] = \Pr[F] E \left[ \sum_{j=1}^{m} w_j X_j \bigg| F \right] + E \left[ \sum_{j=1}^{m} w_j X_j \bigg| \bar{F} \right] \Pr[\bar{F}].$$

Since $w_j \geq 0$ for all $j$,

$$E \left[ \sum_{j=1}^{m} w_j X_j \bigg| \bar{F} \right] \geq 0.$$

Thus

$$E \left[ \sum_{j=1}^{m} w_j X_j \bigg| F \right] = \frac{1}{\Pr[F]} \left( E \left[ \sum_{j=1}^{m} w_j X_j \right] - E \left[ \sum_{j=1}^{m} w_j X_j \bigg| \bar{F} \right] \Pr[\bar{F}] \right)$$

$$\leq \frac{1}{\Pr[F]} \cdot E \left[ \sum_{j=1}^{m} w_j X_j \right]$$

$$\leq \frac{(c\ln n)Z_{LP}^*}{1 - \frac{1}{ne^{-1}}}$$

$$\leq 2c(\ln n)Z_{LP}^*.$$