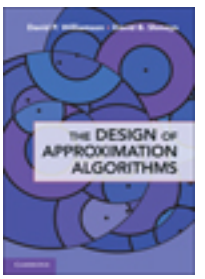

The Design of Approximation Algorithms

David P. Williamson, David B. Shmoys

Book DOI: http://dx.doi.org/10.1017/CBO9780511921735

Online ISBN: 9780511921735

Hardback ISBN: 9780521195270

Chapter

1 - An Introduction to Approximation Algorithms pp. 3-26

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511921735.002

Cambridge University Press

CHAPTER 1

An Introduction to Approximation Algorithms

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. This 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 $P \neq NP$, 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. Throughout 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* α-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* α *of the value of an optimal solution.*

For an α-approximation algorithm, we will call α 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 might 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 *a priori* guarantee for any input, but there is also an *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_ϵ *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 nontrivial 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 $O(n^{\epsilon^{-1}})$-approximation algorithm for the maximum clique problem, unless* P = NP.

To see how strong this theorem is, observe that it is completely trivial 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, nonintegral 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 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 and then the constraints. Given the discussion above, the problem of finding a minimum-weight set cover is equivalent to the following integer program:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} w_j x_j \\ \text{subject to} \quad & \sum_{j:e_i \in S_j} x_j \geq 1, \qquad i = 1, \ldots, n, \\ & x_j \in \{0, 1\}, \qquad j = 1, \ldots, m. \end{aligned} \tag{1.1}$$

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 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, so solving the integer program above for any set cover input in polynomial time would imply that P = 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:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} w_j x_j \\ \text{subject to} \quad & \sum_{j:e_i \in S_j} x_j \geq 1, \qquad i = 1, \ldots, n, \\ & x_j \geq 0, \qquad j = 1, \ldots, m. \end{aligned} \tag{1.2}$$

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 a solution to the integer program of objective function value that is within a certain factor f of the value of the linear program Z^*_{LP}. Thus, the integer solution will cost no more than $f \cdot \text{OPT}$. In the following section, we will show how one can similarly round the solution to something called the dual of the linear programming relaxation. In Section 1.5, we will see that in fact one does not need to solve the dual of the linear programming relaxation, but in fact can quickly construct a dual feasible solution with the properties needed to allow a good rounding. In Section 1.6, a type of algorithm called a greedy algorithm will be given; in this case, linear programming need not be used at all, but one can use the dual to improve the analysis of the algorithm. Finally, in Section 1.7, we will see how randomized rounding of the solution to the linear programming relaxation can lead to an approximation algorithm for the set cover problem.

Because we will frequently be referring to linear programs and linear programming, we will often abbreviate these terms by the acronym LP. Similarly, IP stands for either integer program or integer programming.

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 $f_i = |\{j : e_i \in S_j\}|$ 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. □

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

$$\begin{aligned} \sum_{j \in I} w_j &\leq \sum_{j=1}^{m} w_j \cdot (f \cdot x_j^*) \\ &= f \sum_{j=1}^{m} w_j x_j^* \\ &= f \cdot Z_{LP}^* \\ &\leq f \cdot \mathrm{OPT}, \end{aligned}$$

where the final inequality follows from the argument above that $Z_{LP}^* \leq \mathrm{OPT}$. □

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 *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 α is significantly smaller than f; in this case we know that $\sum_{j \in I} w_j = \alpha Z_{LP}^* \leq \alpha \, \mathrm{OPT}$, and the solution is within a factor of α of optimal. The algorithm can easily compute α, given that it computes I and solves the LP relaxation.

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:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} y_i \\ \text{subject to} \quad & \sum_{i:e_i \in S_j} y_i \leq w_j, \qquad j = 1, \ldots, m, \\ & y_i \geq 0, \qquad i = 1, \ldots, n. \end{aligned} \tag{1.3}$$

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 this dual 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 this inequality, 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 ϵ be the smallest difference between the right-hand side and left-hand side of all constraints involving e_k; that is, $\epsilon = \min_{j:e_k \in S_j}(w_j - \sum_{i:e_i \in S_j} y^*_i)$. 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 ϵ. 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 e_i; 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 $f \sum_{i=1}^{m} y^*_i$, 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

$$\begin{aligned}
\sum_{j \in I'} w_j &= \sum_{j \in I'} \sum_{i:e_i \in S_j} y^*_i \\
&= \sum_{i=1}^{n} |\{j \in I' : e_i \in S_j\}| \cdot y^*_i \\
&\leq \sum_{i=1}^{n} f_i y^*_i \\
&\leq f \sum_{i=1}^{n} y^*_i \\
&\leq f \cdot \mathrm{OPT}.
\end{aligned}$$

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:

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

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. The only way this can happen is that whenever $y_i^* > 0$ then $\sum_{j:e_i \in S_j} x_j^* = 1$, and whenever $x_j^* > 0$, then $\sum_{i:e_i \in S_j} y_i^* = w_j$. 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 $I' \supseteq I$.

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 $\sum_{i:e_i \in S_j} y_i = w_j$, 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, $T = \{j : \sum_{i:e_i \in S_j} y_i = w_j\}$. If T is a set cover, then we are done. If T is not a set cover, then some item e_i is uncovered, and as shown in the proof of Lemma 1.7 it is possible to improve the dual objective function by increasing y_i by some $\epsilon > 0$. More specifically, we can increase y_i by $\min_{j:e_i \in S_j}(w_j - \sum_{k:e_k \in S_j} y_k)$, so that the constraint becomes tight for the subset S_j that attains the minimum. Additionally, the modified dual solution remains feasible. Thus, we can add j to T, and element e_i is now covered by the sets in T. We repeat this process until T is a set cover. Since an additional element e_i is covered each time, the process is repeated at most n times. To complete the description of the algorithm, we need to give only an initial dual feasible solution.

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

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

We can use the solution $y_i = 0$ for each $i = 1, \ldots, n$; this is feasible since each w_j, $j = 1, \ldots, m$, is nonnegative. A formal description is given in Algorithm 1.1.

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 it 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.

$I \leftarrow \emptyset$
$\hat{S}_j \leftarrow S_j \quad \forall j$
while I is not a set cover **do**
  $\ell \leftarrow \arg\min_{j:\hat{S}_j \neq \emptyset} \frac{w_j}{|\hat{S}_j|}$
  $I \leftarrow I \cup \{\ell\}$
  $\hat{S}_j \leftarrow \hat{S}_j - S_\ell \quad \forall j$

Algorithm 1.2. A greedy algorithm for the set cover problem.

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 kth *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 that 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 kth 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 kth iteration. If the algorithm takes ℓ 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, $\hat{S}_j = S_j - \bigcup_{p \in I_k} S_p$. Then we claim that for the set j chosen in the kth iteration,

$$w_j \leq \frac{n_k - n_{k+1}}{n_k} \text{OPT}. \tag{1.5}$$

Given the claimed inequality (1.5), we can prove the theorem. Let I contain the indices of the sets in our final solution. Then

$$\begin{aligned}\sum_{j\in I} w_j &\le \sum_{k=1}^{\ell} \frac{n_k - n_{k+1}}{n_k}\,\mathrm{OPT} \\ &\le \mathrm{OPT}\cdot\sum_{k=1}^{\ell}\left(\frac{1}{n_k} + \frac{1}{n_k-1} + \cdots + \frac{1}{n_{k+1}+1}\right) \qquad (1.6)\\ &= \mathrm{OPT}\cdot\sum_{i=1}^{n}\frac{1}{i} \\ &= H_n\cdot\mathrm{OPT},\end{aligned}$$

where the inequality (1.6) follows from the fact that $\frac{1}{n_k} \le \frac{1}{n_k - i}$ for $0 \le i < n_k$.

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

$$\min_{j:\hat{S}_j\neq\emptyset} \frac{w_j}{|\hat{S}_j|} \le \frac{\mathrm{OPT}}{n_k}. \qquad (1.7)$$

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

$$\min_{j:\hat{S}_j\neq\emptyset} \frac{w_j}{|\hat{S}_j|} \le \frac{\sum_{j\in O} w_j}{\sum_{j\in O}|\hat{S}_j|} = \frac{\mathrm{OPT}}{\sum_{j\in O}|\hat{S}_j|} \le \frac{\mathrm{OPT}}{n_k},$$

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 $\frac{w_j}{|\hat{S}_j|} \le \frac{\mathrm{OPT}}{n_k}$. 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,

$$w_j \le \frac{|\hat{S}_j|\,\mathrm{OPT}}{n_k} = \frac{n_k - n_{k+1}}{n_k}\,\mathrm{OPT}\,. \qquad \square$$

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 $Z^*_{LP} \le \mathrm{OPT}$.

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

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, $\sum_{i=1}^{n} y'_i \le Z^*_{LP}$, so that $\sum_{j\in I} w_j = \sum_{i=1}^{n} y_i = H_g \sum_{i=1}^{n} y'_i \le H_g \cdot \mathrm{OPT}$. 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 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 $e_i \in \hat{S}_j$, we set $y_i = w_j/|\hat{S}_j|$. Since each $e_i \in \hat{S}_j$ 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, $w_j = \sum_{i:e_i \in \hat{S}_j} y_i$; that is, the weight of the subset S_j chosen in the kth iteration is equal to the sum of the duals y_i of the uncovered elements that are covered in the kth 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 kth iteration, so that $a_1 = |S_j|$, and $a_{\ell+1} = 0$. Let A_k be the uncovered elements of S_j covered in the kth iteration, so that $|A_k| = a_k - a_{k+1}$. If subset S_p is chosen in the kth iteration, then for each element $e_i \in A_k$ covered in the kth iteration,

$$y'_i = \frac{w_p}{H_g |\hat{S}_p|} \leq \frac{w_j}{H_g a_k},$$

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

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

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 than P = 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 α-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 holds, if there exists an α-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, however, it is quite likely that the solution is not a set cover. Nevertheless, this 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

$$\begin{aligned}
\Pr[e_i \text{ not covered}] &= \prod_{j:e_i \in S_j} (1 - x_j^*) \\
&\leq \prod_{j:e_i \in S_j} e^{-x_j^*} \\
&= e^{-\sum_{j:e_i \in S_j} x_j^*} \\
&\leq e^{-1},
\end{aligned}$$

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,

$$\begin{aligned}
\Pr[e_i \text{ not covered}] &= \prod_{j:e_i \in S_j} (1 - x_j^*)^{c \ln n} \\
&\leq \prod_{j:e_i \in S_j} e^{-x_j^*(c \ln n)} \\
&= e^{-(c \ln n) \sum_{j:e_i \in S_j} x_j^*} \\
&\leq \frac{1}{n^c},
\end{aligned}$$

as desired.

We now need to prove only 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

$$\begin{aligned}
E\left[\sum_{j=1}^{m} w_j X_j\right] &= \sum_{j=1}^{m} w_j \Pr[X_j = 1] \\
&\leq \sum_{j=1}^{m} w_j (c \ln n) x_j^* \\
&= (c \ln n) \sum_{j=1}^{m} w_j x_j^* = (c \ln n) Z_{LP}^*.
\end{aligned}$$

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 $\Pr[F] \geq 1 - \frac{1}{n^{c-1}}$, and we also know that

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

Since $w_j \geq 0$ for all j,

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

Thus,

$$\begin{aligned} E\left[\sum_{j=1}^{m} w_j X_j \middle| F\right] &= \frac{1}{\Pr[F]} \left(E\left[\sum_{j=1}^{m} w_j X_j\right] - E\left[\sum_{j=1}^{m} w_j X_j \middle| \bar{F}\right] \Pr[\bar{F}] \right) \\ &\leq \frac{1}{\Pr[F]} \cdot E\left[\sum_{j=1}^{m} w_j X_j\right] \\ &\leq \frac{(c \ln n) Z^*_{LP}}{1 - \frac{1}{n^{c-1}}} \\ &\leq 2c(\ln n) Z^*_{LP} \end{aligned}$$

for $n \geq 2$ and $c \geq 2$. □

While in this case there is a simpler and faster approximation algorithm that achieves a better performance guarantee, we will see in Chapter 5 that sometimes randomized algorithms are simpler to describe and analyze than deterministic algorithms. In fact, most of the randomized algorithms we present in this book can be *derandomized*: that is, a deterministic variant of them can be created that achieves the expected performance guarantee of the randomized algorithm. However, these deterministic algorithms are sometimes more complicated to describe. In addition, there are some cases in which the deterministic variant is easy to state, but the only way in which we know how to analyze the algorithm is by analyzing a corresponding randomized algorithm.

This brings us to the end of our introduction to approximation algorithms. In subsequent chapters, we will look at the techniques introduced here – as well as a few others – in greater depth, and see their application to many other problems.

Exercises

1.1 In the set cover problem, the goal is to find a collection of subsets indexed by I that minimizes $\sum_{j \in I} w_j$ such that

$$\left| \bigcup_{j \in I} S_j \right| = |E|.$$

Consider the *partial cover problem*, in which one finds a collection of subsets indexed by I that minimizes $\sum_{j \in I} w_j$ such that

$$\left| \bigcup_{j \in I} S_j \right| \geq p|E|,$$

where $0 < p < 1$ is some constant.

(a) Give a polynomial-time algorithm to find a solution to the partial cover problem in which the value is no more than $c(p) \cdot \text{OPT}$, where $c(p)$ is a constant that depends on p, and OPT is the value of the optimal solution to the set cover problem.
(b) Give an $f(p)$-approximation algorithm for the partial cover problem, such that f is non-decreasing in p and $f(1) \leq H_{|E|}$.

1.2 In the *directed Steiner tree problem*, we are given as input a directed graph $G = (V, A)$, nonnegative costs $c_{ij} \geq 0$ for arcs $(i, j) \in A$, a root vertex $r \in V$, and a set of terminals $T \subseteq V$. The goal is to find a minimum-cost tree such that for each $i \in T$ there exists a directed path from r to i.

Prove that for some constant c there can be no $c \log |T|$-approximation algorithm for the directed Steiner tree problem, unless P = NP.

1.3 In the *metric asymmetric traveling salesman problem*, we are given as input a complete directed graph $G = (V, A)$ with costs $c_{ij} \geq 0$ for all arcs $(i, j) \in A$, such that the arc costs obey the *triangle inequality*: for all $i, j, k \in V$, we have that $c_{ij} + c_{jk} \geq c_{ik}$. The goal is to find a *tour* of minimum cost, that is, a directed cycle that contains each vertex exactly once, such that the sum of the cost of the arcs in the cycle is minimized.

One approach to finding an approximation algorithm for this problem is to first find a minimum-cost strongly connected *Eulerian* subgraph of the input graph. A directed graph is strongly connected if for any pair of vertices $i, j \in V$ there is a path from i to j and a path from j to i. A directed graph is Eulerian if the indegree of each vertex equals its outdegree. Given a strongly connected Eulerian subgraph of the input to the problem, it is possible to use a technique called "shortcutting" (discussed in Section 2.4) to turn this into a tour of no greater cost by using the triangle inequality.

One way to find a strongly connected Eulerian subgraph is as follows: We first find a *minimum mean-cost cycle* in the graph. A minimum mean-cost cycle is a directed cycle that minimizes the ratio of the cost of the arcs in the cycle to the number of arcs in the cycle. Such a cycle can be found in polynomial time. We then choose one vertex of the cycle arbitrarily, remove all other vertices of the cycle from the graph, and repeat. We do this until only one vertex of the graph is left. Consider the subgraph consisting of all the arcs from all the cycles found.

(a) Prove that the subgraph found by the algorithm is a strongly connected Eulerian subgraph of the input graph.
(b) Prove that the cost of this subgraph is at most $2H_n \cdot \text{OPT}$, where $n = |V|$ and OPT is the cost of the optimal tour. Conclude that this algorithm is a $2H_n$-approximation algorithm for the metric asymmetric traveling salesman problem.

1.4 In the *uncapacitated facility location problem*, we have a set of clients D and a set of facilities F. For each client $j \in D$ and facility $i \in F$, there is a cost c_{ij} of assigning client

j to facility i. Furthermore, there is a cost f_i associated with each facility $i \in F$. The goal of the problem is to choose a subset of facilities $F' \subseteq F$ so as to minimize the total cost of the facilities in F' and the cost of assigning each client $j \in D$ to the nearest facility in F'. In other words, we wish to find F' so as to minimize $\sum_{i \in F'} f_i + \sum_{j \in D} \min_{i \in F'} c_{ij}$.

(a) Show that there exists some c such that there is no $(c \ln |D|)$-approximation algorithm for the uncapacitated facility location problem unless P = NP.
(b) Give an $O(\ln |D|)$-approximation algorithm for the uncapacitated facility location problem.

1.5 Consider the vertex cover problem.

(a) Prove that any extreme point of the linear program

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in V} w_i x_i \\ \text{subject to} \quad & x_i + x_j \geq 1, \qquad \forall (i, j) \in E, \\ & x_i \geq 0, \qquad i \in V, \end{aligned}$$

has the property that $x_i \in \{0, \frac{1}{2}, 1\}$ for all $i \in V$. (Recall that an *extreme point* x is a feasible solution that cannot be expressed as $\lambda x^1 + (1 - \lambda)x^2$ for $0 < \lambda < 1$ and feasible solutions x^1 and x^2 distinct from x.)
(b) Give a $\frac{3}{2}$-approximation algorithm for the vertex cover problem when the input graph is planar. You may use the facts that polynomial-time LP solvers return extreme points, and that there is a polynomial-time algorithm to 4-color any planar graph (i.e., the algorithm assigns each vertex one of four colors such that for any edge $(i, j) \in E$, vertices i and j have been assigned different colors).

1.6 In the *node-weighted Steiner tree problem*, we are given as input an undirected graph $G = (V, E)$, node weights $w_i \geq 0$ for all $i \in V$, edge costs $c_e \geq 0$ for all $e \in E$, and a set of terminals $T \subseteq V$. The cost of a tree is the sum of the weights of the nodes plus the sum of the costs of the edges in the tree. The goal of the problem is to find a minimum-weight tree that spans all the terminals in T.

(a) Show that there exists some c such that there is no $(c \ln |T|)$-approximation algorithm for the node-weighted Steiner tree problem unless P = NP.
(b) Give a greedy $O(\ln |T|)$-approximation algorithm for the node-weighted Steiner tree problem.

Chapter Notes

The term "approximation algorithm" was coined by David S. Johnson [179] in an influential and prescient 1974 paper. However, earlier papers had proved the performance guarantees of heuristics, including a 1967 paper of Erdős [99] on the maximum cut problem (to be discussed in Section 6.2), a 1966 paper of Graham [142] on a scheduling problem (to be discussed in Section 2.3), and a 1964 paper of Vizing [284] on the edge coloring problem (to be discussed in Section 2.7). Johnson's paper gave an $O(\log n)$-approximation algorithm for the unweighted set cover problem, as well as approximation algorithms for the maximum satisfiability problem (to be discussed in Section 5.1), vertex coloring (Sections 5.12, 6.5, and 13.2), and the maximum clique

problem. At the end of the paper, Johnson [179] speculates about the approximability of these various problems:

> The results described in this paper indicate a possible classification of optimization problems as to the behavior of their approximation algorithms. Such a classification must remain tentative, at least until the existence of polynomial-time algorithms for finding optimal solutions has been proved or disproved. In the meantime, many questions can be asked. Are there indeed $O(\log n)$ coloring algorithms? Are there any clique finding algorithms better than $O(n^{\epsilon})$ for all $\epsilon > 0$? Where do other optimization problems fit into the scheme of things? What is it that makes algorithms for different problems behave in the same way? Is there some stronger kind of reducibility than the simple polynomial reducibility that will explain these results, or are they due to some structural similarity between the problems as we define them? And what other types of behavior and ways of analyzing and measuring it are possible? (p. 278)

There has been substantial work done in attempting to answer these questions in the decades since Johnson's paper appeared, with significant progress; for instance, Theorem 1.4 shows that no clique algorithm of the kind Johnson mentions is possible unless P = NP.

Other books on approximation algorithms are available, including the textbooks of Ausiello, Crescenzi, Gambosi, Kann, Marchetti-Spaccamela, and Protasi [27] and of Vazirani [283], and the collection of surveys edited by Hochbaum [162]. Many books on algorithms and combinatorial optimization now contain sections on approximation algorithms, including the textbooks of Bertsimas and Tsitsiklis [47], Cook, Cunningham, Pulleyblank, and Schrijver [81], Cormen, Leiserson, Rivest, and Stein [82], Kleinberg and Tardos [198], and Korte and Vygen [203].

For solid introductions to linear programming, we suggest the books of Bertsimas and Tsitsiklis [47], Chvátal [79] and Ferris, Mangasarian, and Wright [112]. Bertsekas and Tsitsiklis [45], Durrett [93, 94], and Ross [256] provide basic introductions to probability theory; the first few chapters of the book of Mitzenmacher and Upfal [226] provide a brief introduction to probability theory in the context of computer algorithms.

The antivirus application of the set cover problem mentioned is due to Kephart, Sorkin, Arnold, Chess, Tesauro, and White [188].

Theorem 1.3 on the non-existence of approximation schemes for problems in MAX SNP is due to Arora, Lund, Motwani, Sudan, and Szegedy [19], building on earlier work of Feige, Goldwasser, Lovász, Safra, and Szegedy [108] and Arora and Safra [23]. Theorem 1.4 on the hardness of approximating the maximum clique problem is due to Håstad [158], with a strengthening due to Zuckerman [296].

The LP rounding algorithm of Section 1.3 and the dual rounding algorithm of Section 1.4 are due to Hochbaum [160]. The primal-dual algorithm of Section 1.5 is due to Bar-Yehuda and Even [35]. The greedy algorithm and the LP-based analysis of Section 1.6 are due to Chvátal [78]. The randomized rounding algorithm of Section 1.7 is apparently folklore. Johnson [179] and Lovász [218] give earlier greedy $O(\log n)$-approximation algorithms for the unweighted set cover problem.

Theorem 1.13 on the hardness of approximating the set cover problem is due to Lund and Yannakakis [220], with a strengthening due to Bellare, Goldwasser, Lund,

and Russell [43]. Theorem 1.14 on the hardness of the set cover problem is due to Feige [107]. Theorem 1.15 on the hardness of the vertex cover problem is due to Dinur and Safra [91], while Theorem 1.16, which uses the unique games conjecture, is due to Khot and Regev [194].

Exercise 1.3 is an unpublished result of Kleinberg and Williamson. The algorithm in Exercise 1.4 is due to Hochbaum [161]. Nemhauser and Trotter [231] show that all extreme points of the linear programming relaxation of vertex cover have value $\{0, \frac{1}{2}, 1\}$ (used in Exercise 1.5). Exercise 1.6 is due to Klein and Ravi [196].