PART ONE

An Introduction to the Techniques

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 $j = 1, \dots, m$ and $\sum_{j:e_i \in S_j} x_j \geq 1$ for each $i = 1, \dots, m$ 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, $i = 1, \dots, n$; then $f = \max_{i=1,\dots,n} f_i$. 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

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

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 *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 $\sum_{j \in I} w_j = \alpha Z_{LP}^* \leq \alpha\,\text{OPT}$, 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

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