In *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*
Edited by Edmund K. Burke and Graham Kendall

Chapter 18

APPROXIMATION ALGORITHMS

Carla P. Gomes
Department of Computer Science
Cornell University
Ithaca, NY, USA

Ryan Williams
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

18.1 INTRODUCTION

Most interesting real-world optimization problems are very challenging from a computational point of view. In fact, quite often, finding an optimal or even a near-optimal solution to a large-scale optimization problem may require computational resources far beyond what is practically available. There is a substantial body of literature exploring the computational properties of optimization problems by considering how the computational demands of a solution method grow with the size of the problem instance to be solved (see e.g. Chapter 11 or Aho et al., 1979). A key distinction is made between problems that require computational resources that grow polynomially with problem size versus those for which the required resources grow exponentially. The former category of problems are called efficiently solvable, whereas problems in the latter category are deemed *intractable* because the exponential growth in required computational resources renders all but the smallest instances of such problems unsolvable.

It has been determined that a large class of common optimization problems are classified as *NP-hard*. See Chapter 11 for more details. It is widely believed—though not yet proven (Clay Mathematics Institute, 2003)—that NP-hard problems are intractable, which means that there does not exist an efficient algorithm (i.e. one that scales polynomially) that is guaranteed to find an optimal solution for such problems. Examples of NP-hard optimization

tasks are the minimum traveling salesman problem, the minimum graph coloring problem, and the minimum bin-packing problem. As a result of the nature of NP-hard problems, progress that leads to a better understanding of the structure, computational properties, and ways of solving one of them, *exactly* or *approximately*, also leads to better algorithms for solving hundreds of other different but related NP-hard problems. Several thousand computational problems, in areas as diverse as economics, biology, operations research, computer-aided design and finance, have been shown to be NP-hard. (See Aho et al., 1979, for further description and discussion of these problems.)

A natural question to ask is whether *approximate* (i.e. near-optimal) solutions can possibly be found efficiently for such hard optimization problems. Heuristic local search methods, such as tabu search and simulated annealing (see Chapters 6 and 7), are often quite effective at finding near-optimal solutions. However, these methods do not come with rigorous guarantees concerning the quality of the final solution or the required maximum runtime. In this chapter, we will discuss a more theoretical approach to this issue consisting of so-called "approximation algorithms", which are efficient algorithms that can be proven to produce solutions of a certain quality. We will also discuss classes of problems for which no such efficient approximation algorithms exist, thus leaving an important role for the quite general, heuristic local search methods.

The design of good approximation algorithms is a very active area of research where one continues to find new methods and techniques. It is quite likely that these techniques will become of increasing importance in tackling large real-world optimization problems.

In the late 1960s and early 1970s a precise notion of approximation was proposed in the context of multiprocessor scheduling and bin packing (Graham, 1966; Garey et al., 1972; Johnson, 1974). Approximation algorithms generally have two properties. First, they provide a feasible solution to a problem instance in polynomial time. In most cases, it is not difficult to devise a procedure that finds *some* feasible solution. However, we are interested in having some assured quality of the solution, which is the second aspect characterizing approximation algorithms. The quality of an approximation algorithm is the maximum "distance" between its solutions and the optimal solutions, evaluated over all the possible instances of the problem. Informally, an algorithm approximately solves an optimization problem if it always returns a feasible solution whose measure is close to optimal, for example within a factor bounded by a constant or by a slowly growing function of the input size. Given a constant α, an algorithm $\mathcal{A}$ is an α-approximation algorithm for a given minimization problem Π if its solution is at most α times the optimum, considering all the possible instances of problem Π.

The focus of this chapter is on the design of approximation algorithms for NP-hard optimization problems. We will show how standard algorithm de-

sign techniques such as greedy and local search methods have been used to devise good approximation algorithms. We will also show how randomization is a powerful tool for designing approximation algorithms. Randomized algorithms are interesting because, in general, such approaches are easier to analyze and implement, and faster than deterministic algorithms (Motwani and Raghavan, 1995). A randomized algorithm is simply an algorithm that performs some of its choices randomly; it "flips a coin" to decide what to do at some stages. As a consequence of its random component, different executions of a randomized algorithm may result in different solutions and runtime, even when considering the same instance of a problem. We will show how one can combine randomization with approximation techniques in order to efficiently approximate NP-hard optimization problems. In this case, the approximation solution, the approximation ratio, and the runtime of the approximation algorithm may be random variables. Confronted with an optimization problem, the goal is to produce a randomized approximation algorithm with runtime provably bounded by a polynomial and whose feasible solution is close to the optimal solution, *in expectation.* Note that these guarantees hold for every instance of the problem being solved. The only randomness in the performance guarantee of the randomized approximation algorithm comes from the algorithm itself, and not from the instances.

Since we do not know of efficient algorithms to find optimal solutions for NP-hard problems, a central question is whether we can efficiently compute good approximations that are close to optimal. It would be very interesting (and practical) if one could go from exponential to polynomial time complexity by relaxing the constraint on optimality, especially if we guarantee at most a relatively small error.

Good approximation algorithms have been proposed for some key problems in combinatorial optimization. The so-called APX complexity class includes the problems that allow a polynomial-time approximation algorithm with a performance ratio bounded by a constant. For some problems, we can design even better approximation algorithms. More precisely we can consider a family of approximation algorithms that allows us to get as close to the optimum as we like, as long as we are willing to trade quality with time. This special family of algorithms is called an *approximation scheme* (AS) and the so-called PTAS class is the class of optimization problems that allow for a *polynomial time approximation scheme* that scales polynomially in the size of the input. In some cases we can devise approximation schemes that scale polynomially, both in the size of the input and in the magnitude of the approximation error. We refer to the class of problems that allow such *fully polynomial time approximation schemes* as FPTAS.

Nevertheless, for some NP-hard problems, the approximations that have been obtained so far are quite poor, and in some cases no one has ever been able

to devise approximation algorithms within a constant factor of the optimum. Initially it was not clear if these weak results were due to our lack of ability in devising good approximation algorithms for such problems or to some inherent structural property of the problems that excludes them from having good approximations. We will see that indeed there are limitations to approximation which are *intrinsic* to some classes of problems. For example, in some cases there is a lower bound on the constant factor of the approximation, and in other cases, we can provably show that there are no approximations within *any* constant factor from the optimum. Essentially, there is a wide range of scenarios going from NP-hard optimization problems that allow approximations to *any* required degree, to problems not allowing approximations at all. We will provide a brief introduction to proof techniques used to derive non-approximability results.

We believe that the best way to understand the ideas behind approximation and randomization is to study instances of algorithms with these properties, through examples. Thus in each section, we will first introduce the intuitive concept, then reinforce its salient points through well-chosen examples of prototypical problems. Our goal is far from trying to provide a comprehensive survey of approximation algorithms or even the best approximation algorithms for the problems introduced. Instead, we describe different design and evaluation techniques for approximation and randomized algorithms, using clear examples that allow for relatively simple and intuitive explanations. For some problems discussed in the chapter there are approximations with better performance guarantees but requiring more sophisticated proof techniques that are beyond the scope of this introductory tutorial. In such cases we will point the reader to the relevant literature results. In summary, our goals for this chapter are as follows:

1 Present the fundamental ideas and concepts underlying the notion of approximation algorithms.

2 Provide clear examples that illustrate different techniques for the design and evaluation of efficient approximation algorithms. The examples include accessible proofs of the approximation bounds.

3 Introduce the reader to the classification of optimization problems according to their polynomial-time approximability, including basic ideas on polynomial-time inapproximability.

4 Show the power of randomization for the design of approximation algorithms that are in general faster and easier to analyze and implement than the deterministic counterparts.

5 Show how we can use a randomized approximation algorithm as a heuristic to guide a complete search method (empirical results).

6 Present promising application areas for approximation and randomized algorithms.

7 Provide additional sources of information on approximation and randomization methods.

In Section 18.2 we introduce precise notions and concepts used in approximation algorithms. In this section we describe key design techniques for approximation algorithms. We use clear prototypical examples to illustrate the main techniques and concepts, such as the minimum vertex cover, the knapsack problem, the maximum satisfiability problem, the traveling salesman problem, and the maximum cut problem. As mentioned earlier, we are not interested in providing the best approximation algorithms for these problems, but rather in illustrating how standard algorithm techniques can be used effectively to design and evaluate approximation algorithms. In Section 18.3 we provide a tour of the main approximation classes, including a brief introduction to techniques to proof lower bounds on approximability. In Section 18.4 we describe some promising areas of application of approximation algorithms. Section 18.6 summarizes the chapter and provides additional sources of information on approximation and randomization methods.

18.2 APPROXIMATION STRATEGIES

18.2.1 Preliminaries

Optimization Problems We will define optimization problems in a traditional way (Aho et al., 1979; Ausiello et al., 1999). Each optimization problem has three defining features: the structure of the input *instance*, the criterion of a feasible *solution* to the problem, and the *measure* function used to determine which feasible solutions are considered to be optimal. It will be evident from the problem name whether we desire a feasible solution with a minimum or maximum measure. To illustrate, the minimum vertex cover problem may be defined in the following way.

Minimum Vertex Cover

Instance: An undirected graph $G = (V, E)$.
Solution: A subset $S \subseteq V$ such that for every $\{u, v\} \in E$, either $u \in S$ or $v \in S$.
Measure: $|S|$.

We use the following notation for items related to an instance I.

- $Sol(I)$ is the set of feasible solutions to I,
- $m_I : Sol(I) \rightarrow \mathbb{R}$ is the measure function associated with I, and
- $Opt(I) \subseteq Sol(I)$ is the feasible solutions with optimal measure (be it minimum or maximum).

Hence, we may completely specify an optimization problem Π by giving a set of tuples $\{(I, Sol(I), m_I, Opt(I))\}$ over all possible instances I. It is important to keep in mind that $Sol(I)$ and I may be over completely different domains. In the above example, the set I is all undirected graphs, while $Sol(I)$ is all possible subsets of vertices in a graph.

Approximation and Performance Roughly speaking, an algorithm approximately solves an optimization problem if it always returns a feasible solution whose measure is close to optimal. This intuition is made precise below.

Let Π be an optimization problem. We say that an algorithm A *feasibly solves* Π if given an instance $I \in \Pi$, $A(I) \in Sol(I)$; that is, A returns a feasible solution to I.

Let A feasibly solve Π. Then we define the *approximation ratio* $\alpha(A)$ of A to be the minimum possible ratio between the measure of $A(I)$ and the measure of an optimal solution. Formally,

$$\alpha(A) = \min_{I \in \Pi} \frac{m_I(A(I))}{m_I(Opt(I))}$$

For minimization problems, this ratio is always at least 1. Respectively, for maximization problems, it is always at most 1.

Complexity Background We define a decision problem as an optimization problem in which the measure is 0–1 valued. That is, solving an instance I of a decision problem corresponds to answering a *yes/no* question about I (where *yes* corresponds to a measure of 1, and *no* corresponds to a measure of 0). We may therefore represent a decision problem as a subset S of the set of all possible instances: members of S represent instances with measure 1.

Informally, P (polynomial time) is defined as the class of decision problems Π for which there exists a corresponding algorithm A_Π such that every instance $I \in \Pi$ is solved by A_Π within a polynomial ($|I|^k$ for some constant k) number of steps on any "reasonable" model of computation. Reasonable models include single-tape and multi-tape Turing machines, random access machines, pointer machines, etc.

While P is meant to represent a class of problems that can be efficiently solved, NP (nondeterministic polynomial time) is a class of decision problems

Π that can be efficiently *checked*. More formally, NP is the class of decision problems Π for which there exists a corresponding decision problem Π' in P and constant k satisfying

$$I \in \Pi \text{ if and only if there exists } C \in \{0, 1\}^{|I|^k} \text{ such that } (I, C) \in \Pi'$$

In other words, one can determine if an instance I is in an NP problem efficiently if one is also provided with a certain short string C, which is of length polynomial in I. For example, consider the NP problem of determining if a graph G has a path that travels through all nodes exactly once (this is known as the Hamiltonian path problem). Here, the instances I are graphs, and the proofs C are Hamiltonian paths. If one is given G along with a full description C of a path, it is easy to verify that C describes a Hamiltonian path by checking that

1 the path contains all nodes in G,

2 no node appears more than once in the path, and

3 any two adjacent nodes in the path have an edge between them in G.

However, no polynomial time algorithm is known for finding a Hamiltonian path when one is only given the graph G, and this is the fundamental difference between P and NP. In fact, the Hamiltonian path problem is not only in NP but is also *NP-hard*, see Section 18.1 and Chapter 11.

For $\Pi \in NP$, notice that while a short proof always exists if $I \in \Pi$, it need not be the case that short proofs exist for instances not in Π. Thus, while P problems are considered to be those which are *efficiently decidable*, NP problems are those considered to be *efficiently verifiable* via a short proof.

We will also consider the optimization counterparts to P and NP, which are PO and NPO, respectively. Informally, PO is the class of optimization problems where there exists a polynomial time algorithm that always returns an optimal solution to every instance of the problem, whereas NPO is the class of optimization problems where the measure function is polynomial time computable, and an algorithm can determine whether or not a possible solution is feasible in polynomial time. Our focus here will be on approximating solutions to the "hardest" of NPO problems, those problems where the corresponding decision problem is NP-hard. Interestingly, some NPO problems of this type can be approximated very well, whereas others can hardly be approximated at all.

18.2.2 The Greedy Method

Greedy approximation algorithms are designed with a simple philosophy in mind: repeatedly make choices that get one closer and closer to a feasible solution for the problem. These choices will be optimal according to an imperfect

but easily computable heuristic. In particular, this heuristic tries to be as opportunistic as possible in the short run. (This is why such algorithms are called greedy—a better name might be "short-sighted"). For example, suppose my goal is to find the shortest walking path from my house to the theater. If I believed that the walk via Forbes Avenue is about the same length as the walk via Fifth Avenue, then if I am closer to Forbes than Fifth, it would be reasonable to walk towards Forbes and take that route.

Clearly, the success of this strategy depends on the correctness of my belief that the Forbes path is indeed just as good as the Fifth path. We will show that for some problems, choosing a solution according to an opportunistic, imperfect heuristic achieves a non-trivial approximation algorithm.

Greedy Vertex Cover The minimum vertex cover problem was defined in the preliminaries (Section 18.2.1). Variants on the problem come up in many areas of optimization research. We will describe a simple greedy algorithm that is a 2-approximation to the problem; that is, the cardinality of the vertex cover returned by our algorithm is no more than twice the cardinality of a minimum cover. The algorithm is as follows.

> Greedy-VC: Initially, let S be an empty set. Choose an arbitrary edge $\{u, v\}$. Add u and v to S, and remove u and v from the graph. Repeat until no edges remain in the graph. Return S as the vertex cover.

THEOREM 18.1 *Greedy-VC is a 2-approximation algorithm for Minimum Vertex Cover.*

Proof. First, we claim S as returned by Greedy-VC is indeed a vertex cover. Suppose not; then there exists an edge e which was not covered by any vertex in S. Since we only remove vertices from the graph that are in S, an edge e would remain in the graph after Greedy-VC had completed, which is a contradiction.

Let S^* be a minimum vertex cover. We will now show that S^* contains at least $|S|/2$ vertices. It will follow that $|S^*| \geq |S|/2$, hence our algorithm has a $|S|/|S^*| \leq 2$ approximation ratio.

Since the edges we chose in Greedy-VC do not share any endpoints, it follows that

- $|S|/2$ is the number of edges we chose and
- S^* must have chosen at least one vertex from each edge we chose.

It follows that $|S^*| \geq |S|/2$. □

Sometimes when one proves that an algorithm has a certain approximation ratio, the analysis is somewhat "loose", and may not reflect the best possible

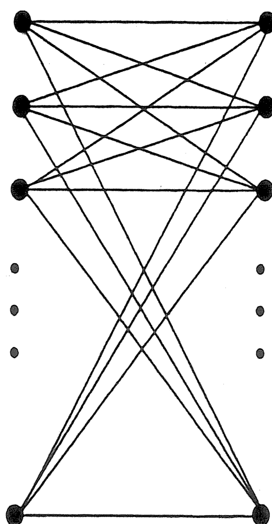

Figure 18.1. A *bipartite* graph is one for which its vertices can be assigned one of two colors (say, *red* or *blue*), in such a way that all edges have endpoints with different colors. Above is a sketch of a *complete* bipartite graph with n nodes colored red and n nodes colored blue. When running Greedy-VC on these instances (for any natural number n), the algorithm will select all $2n$ vertices. This shows the approximation ratio of 2 is tight for Greedy-VC.

ratio that can be derived. It turns out that Greedy-VC is no better than a 2-approximation. In particular, there is an infinite set of Vertex Cover instances where Greedy-VC provably chooses exactly twice the number of vertices necessary to cover the graph, namely in the case of complete bipartite graphs; see Figure 18.1. When such a situation occurs, where the approximation ratio derived for an algorithm is indeed the best possible in the worst case, we say that the bound derived is *tight*.

One final remark should be noted on Vertex Cover. While the above algorithm is indeed quite simple, no better approximation algorithms are known! In fact, it is widely believed that one cannot approximate minimum vertex cover better than $2 - \epsilon$ for any $\epsilon > 0$, unless $P = NP$, see Khot and Regev (2003).

Greedy MAX-SAT The MAX-SAT problem has been very well-studied; variants of it arise in many areas of discrete optimization. To introduce it requires a bit of terminology.

We will deal solely with Boolean variables (that is, those which are either true or false), which we will denote by x_1, x_2, etc. A *literal* is defined as either a variable or the negation of a variable (e.g. x_7, $\neg x_{11}$ are literals). A *clause* is defined as the OR of some literals (*e.g.* $(\neg x_1 \vee x_7 \vee \neg x_{11})$ is a clause). We say that a Boolean formula is in *conjunctive normal form* (CNF) if it is presented as an AND of clauses (*e.g.* $(\neg x_1 \vee x_7 \vee \neg x_{11}) \wedge (x_5 \vee \neg x_2 \vee \neg x_3)$ is in CNF).

Finally, the MAX-SAT problem is to find an assignment to the variables of a Boolean formula in CNF such that the maximum number of clauses are set

to true, or are *satisfied.* Formally:

MAX-SAT

Instance: A Boolean formula F in CNF.
Solution: An assignment a, which is a function from each of the variables in F to $\{true, false\}$.
Measure: The number of clauses in F that are set to true (are satisfied) when the variables in F are assigned according to a.

What might be a natural greedy strategy for approximately solving MAX-SAT? One approach is to pick a variable that satisfies many clauses if it is set to a certain value. Intuitively, if a variable occurs negated in several clauses, setting the variable to *false* will satisfy several clauses; hence this strategy should approximately solve the problem well. Let $n(l_i, F)$ denote the number of clauses in F where the literal l_i appears.

> Greedy-MAXSAT: Pick a literal l_i with maximum $n(l_i, F)$ value. Set the corresponding variable of l_i such that all clauses containing l_i are satisfied, yielding a reduced F. Repeat until no variables remain in F.

It is easy to see that Greedy-MAXSAT runs in polynomial time (roughly quadratic time, depending on the computational model chosen for analysis). It is also a "good" approximation for the MAX-SAT problem.

THEOREM 18.2 *Greedy-MAXSAT is a $\frac{1}{2}$-approximation algorithm for MAX-SAT.*

Proof. Proof by induction on the number of variables n in the formula F. Let m be the total number of clauses in F. If $n = 1$, the result is obvious. For $n > 1$, let l_i have maximum $n(l_i, F)$ value, and v_i be its corresponding variable. Let m_{POS} and m_{NEG} be the number of clauses in F that contain l_i and $\neg l_i$, respectively. After v_i is set so that l_i is true (so both l_i and $\neg l_i$ disappear from F), there are at least $m - m_{\mathrm{POS}} - m_{\mathrm{NEG}}$ clauses left, on $n - 1$ variables.

By induction hypothesis, Greedy-MAXSAT satisfies at least $(m - m_{\mathrm{POS}} - m_{\mathrm{NEG}})/2$ of these clauses, therefore the total number of clauses satisfied is at least $(m - m_{\mathrm{POS}} - m_{\mathrm{NEG}})/2 + m_{\mathrm{POS}} = m/2 + (m_{\mathrm{POS}} - m_{\mathrm{NEG}})/2 \geq m/2$, by our greedy choice of picking the l_i that occurred most often. □

Greedy MAX-CUT Our next example shows how local search (in particular, *hill climbing*) may be employed in designing approximation algorithms. Hill climbing is inherently a greedy strategy: when one has a feasible solution x, one tries to improve it by choosing some feasible y that is "close" to x, but has a better measure (lower or higher, depending on minimization or

maximization). Repeated attempts at improvement often result in "locally" optimal solutions that have a good measure relative to a globally optimal solution (i.e. a member of $Opt(I)$). We illustrate local search by giving an approximation algorithm for the NP-complete MAX-CUT problem which finds a locally optimal satisfying assignment. It is important to note that not all local search strategies try to find a local optimum—for example, simulated annealing attempts to *escape* from local optima in the hopes of finding a global optimum (Kirkpatrick et al., 1983). See Chapter 7 for more details.

MAX-CUT

Instance: An undirected graph $G = (V, E)$.
Solution: A cut of the graph, i.e. a pair (S, T) such that $S \subseteq V$ and $T = V - S$.
Measure: The *cut size*, which is the number of edges crossing the cut, i.e. $|\{\{u, v\} \in E \mid u \in S, v \in T\}|$.

Our local search algorithm repeatedly improves the current feasible solution by changing one vertex's place in the cut, until no more improvement can be made. We will prove that at such a local maximum, the cut size is at least $m/2$.

> Local-Cut: Start with an arbitrary cut of V. For each vertex, determine if moving it to the other side of the partition increases the size of the cut. If so, move it. Repeat until no such movements are possible.

First, observe that this algorithm repeats at most m times, as each movement of a vertex increases the size of the cut by at least 1, and a cut can be at most m in size.

THEOREM 18.3 *Local-Cut is a $\frac{1}{2}$-approximation algorithm for MAX-CUT.*

Proof. Let (S, T) be the cut returned by the algorithm, and consider a vertex v. After the algorithm finishes, observe that the number of edges adjacent to v that cross (S, T) is more than the number of adjacent edges that do not cross, otherwise v would have been moved. Let $\deg(v)$ be the degree of v. Then our observation implies that at least $\deg(v)/2$ edges out of v cross the cut returned by the algorithm.

Let m^* be the total number of edges crossing the cut returned. Each edge has two endpoints, so the sum $\sum_{v \in V}(\deg(v)/2)$ counts each crossing edge at most twice, i.e.

$$\sum_{v \in V}(\deg(v)/2) \le 2m^*$$

However, observe $\sum_{v \in V} \deg(v) = 2m$: when summing up all degrees of vertices, every edge gets counted exactly twice, once for each endpoint. We con-

clude that

$$m = \sum_{v \in V} (\deg(v)/2) \leq 2m^*$$

It follows that the approximation ratio of the algorithm is $\frac{m^*}{m} \geq \frac{1}{2}$. □

It turns out that MAX-CUT admits much better approximation ratios than $\frac{1}{2}$; a so-called *relaxation* of the problem to a semi-definite linear program yields a 0.8786 approximation (Goemans and Williamson, 1995). However, like many optimization problems, MAX-CUT cannot be approximated arbitrarily well ($1 - \epsilon$, for all $\epsilon > 0$) unless P = NP. That is to say, it is unlikely that MAX-CUT is in the *PTAS* complexity class.

Greedy Knapsack The knapsack problem and its special cases have been extensively studied in operations research. The premise behind it is classic: you have a knapsack of capacity C, and a set of items $1, \ldots, n$. Each item has a particular cost c_i of carrying it, along with a profit p_i that you will gain by carrying it. The problem is then to find a subset of items with cost at most C, having maximum profit.

Maximum Integer Knapsack

Instance: A capacity $C \in \mathbb{N}$, and a number of items $n \in \mathbb{N}$, with corresponding costs and profits $c_i, p_i \in \mathbb{N}$ for all $i = 1, \ldots, n$.
Solution: A subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in S} c_j \leq C$.
Measure: The *total profit* $\sum_{j \in S} p_j$.

Maximum Integer Knapsack, as formulated above, is NP-hard. There is also a "fractional" version of this problem (we call it Maximum Fraction Knapsack), which can be solved in polynomial time. In this version, rather than having to pick the entire item, one is allowed to choose *fractions* of items, like 1/8 of the first item, 1/2 of the second item, and so on. The corresponding profit and cost incurred from the items will be similarly fractional (1/8 of the profit and cost of the first, 1/2 of the profit and cost of the second, and so on).

One greedy strategy for solving these two problems is to pack items with the largest profit-to-cost ratio first, with the hopes of getting many small-cost high-profit items in the knapsack. It turns out that this algorithm will not give any constant approximation guarantee, but a tiny variant on this approach will give a 2-approximation for Integer Knapsack, and an exact algorithm for Fraction Knapsack. The algorithms for Integer Knapsack and Fraction Knapsack are, respectively:

- Greedy-IKS: Choose items with the largest profit-to-cost ratio first, until the total cost of items chosen is greater than C. Let j be the last item

chosen, and S be the set of items chosen before j. Return either $\{j\}$ or S, depending on which one is more profitable.

- Greedy-FKS: Choose items as in Greedy-IKS. When the item j makes the cost of the current solution greater than C, add the *fraction* of j such that the resulting cost of the solution is exactly C.

We omit a proof of the following. A full treatment can be found in Ausiello et al. (1999).

LEMMA 18.4 *Greedy-FKS solves Maximum Fraction Knapsack in polynomial time. That is, Greedy-FKS is a 1-approximation to Maximum Fraction Knapsack.*

We entitled the above as a lemma, because we will use it to analyze the approximation algorithm for Integer Knapsack.

THEOREM 18.5 *Greedy-KS is a $\frac{1}{2}$-approximation for Maximum Integer Knapsack.*

Proof. Fix an instance of the problem. Let $P = \sum_{i \in S} p_i$, the total profit of items in S, and j be the last item chosen (as specified in the algorithm). We will show that $P + p_j$ is greater than or equal to the profit of an optimal Integer Knapsack solution. It follows that one of S or $\{j\}$ has at least half the profit of the optimal solution.

Let S_I^* be an optimal Integer Knapsack solution to the given instance, with total profit P_I^*. Similarly, let S_F^* and P_F^* correspond to an optimal Fraction Knapsack solution. Observe that $P_F^* \leq P_I^*$.

By the analysis of the algorithm for Fraction Knapsack, $P_F^* = P + \epsilon p_j$, where $\epsilon \in (0, 1]$ is the fraction chosen for item j in the algorithm. Therefore

$$P + p_j \geq P + \epsilon p_j = P_F^* \geq P_I^*$$

and we are done. □

In fact, this algorithm can be extended to get a *polynomial time approximation scheme* (PTAS) for Maximum Integer Knapsack, (see (Ausiello et al., 1999)). A PTAS has the property that, for any fixed $\epsilon > 0$ provided, it returns a $(1 + \epsilon)$-approximate solution. Furthermore, the runtime is polynomial in the input size, *provided that ϵ is constant.* This allows us to specify a runtime that has $1/\epsilon$ in the exponent. It is typical to view a PTAS as a *family* of successively better (but also slower) approximation algorithms, each running with a successively smaller $\epsilon > 0$. This is intuitively why they are called an approximation *scheme*, as it is meant to suggest that a variety of algorithms are used. A PTAS is quite powerful; such a scheme can approximately solve a problem with ratios arbitrarily close to 1. However, we will observe that many problems provably do not have a PTAS, unless $P = NP$.

18.2.3 Sequential Algorithms

Sequential algorithms are used for approximations on problems where a feasible solution is a partitioning of the instance into subsets. A sequential algorithm "sorts" the items of the instance in some manner, and selects partitions for the instance based on this ordering.

Sequential Bin Packing We first consider the problem of Minimum Bin Packing, which is similar in nature to the knapsack problems.

Minimum Bin Packing

Instance: A set of items $S = \{r_1, \ldots, r_n\}$, where $r_i \in (0, 1]$ for all $i = 1, \ldots, n$.
Solution: Partition of S into bins $B_1, \ldots, B_M$ such that $\sum_{r_j \in B_i} r_j \leq 1$ for all $i = 1, \ldots, M$.
Measure: M.

An obvious algorithm for Minimum Bin Packing is an *on-line* strategy. Initially, let $j = 1$ and have a bin B_1 available. As one runs through the input (r_1, r_2, etc), try to pack the new item r_i into the last bin used, B_j. If r_i does not fit in B_j, create another bin B_{j+1} and put a_i in it. This algorithm is "on-line" as it processes the input in a fixed order, and thus adding new items to the instance while the algorithm is running does not change the outcome. Call this heuristic Last-Bin.

THEOREM 18.6 *Last-Bin is a 2-approximation to Minimum Bin Packing.*

Proof. Let R be the sum of all items, so $R = \sum_{r_i \in S} r_i$. Let m be the total number of bins used by the algorithm, and let m^* be the minimum number of bins possible for the given instance. Note that $m^* \geq R$, as the total number of bins needed is at least the total size of all items (each bin holds 1 unit). Now, given any pair of bins B_i and B_{i+1} returned by the algorithm, the sum of items from S in B_i and B_{i+1} is at least 1; otherwise, we would have stored the items of B_{i+1} in B_i instead. This shows that $m \leq 2R$. Hence $m \leq 2R \leq 2m^*$, and the algorithm is a 2-approximation. □

An interesting exercise for the reader is to construct a series of examples demonstrating that this approximation bound, like the one for Greedy-VC, is tight.

As one might expect, there exist algorithms that give better approximations than the above. For example, we do not even consider the previous bins $B_1, \ldots, B_{j-1}$ when trying to pack an a_i, only the last one is considered.

Motivated by this observation, consider the following modification to Last-Bin. Select each item a_i in decreasing order of size, placing a_i in the *first*

available bin out of $B_1, \ldots, B_j$. (So a new bin is only created if a_i cannot fit in any of the previous j bins.) Call this new algorithm First-Bin. An improved approximation bound may be derived, via an intricate analysis of cases.

THEOREM 18.7 *(Johnson, 1974) First-Bin is a $\frac{11}{9}$-approximation to Minimum Bin Packing.*

Sequential Job Scheduling One of the major problems in scheduling theory is how to assign jobs to multiple machines so that all of the jobs are completed efficiently. Here, we will consider job completion in the shortest amount of time possible. For the purposes of abstraction and simplicity, we will assume the machines are identical in processing power for each job.

Minimum Job Scheduling

Instance: An integer k and a multi-set $T = \{t_1, \ldots, t_n\}$ of *times*, $t_i \in \mathbb{Q}$ for all $i = 1, \ldots, n$ (*i.e.* the t_i are fractions).
Solution: An assignment of jobs to machines, i.e. a function a from $\{1, \ldots, n\}$ to $\{1, \ldots, k\}$.
Measure: The completion time for all machines, assuming they run in parallel: $\max \left\{ \sum_{i:a(i)=j} t_i \mid j \in \{1, \ldots, k\} \right\}$.

The algorithm we propose for Job Scheduling is also on-line: when reading a new job with time t_i, assign it to the machine j that currently has the least amount of work; that is, the j with minimum $\sum_{i:a(i)=j} t_i$. Call this algorithm Sequential-Jobs.

THEOREM 18.8 *Sequential Jobs is a 2-approximation for Minimum Job Scheduling.*

Proof. Let j be a machine with maximum completion time, and let i be the index of the last job assigned to j by the algorithm. Let $s_{i,j}$ be the sum of all times for jobs prior to i that are assigned to j. (This may be thought of as the time that job i begins on machine j.) The algorithm assigned i to the machine with the least amount of work, hence all other machines j' at this point have larger $\sum_{i:a(i)=j'} t_i$. Therefore $s_{i,j} \leq \frac{1}{k} \sum_{i=1}^{n} t_i$, i.e. $s_{i,j}$ is less $1/k$ of the total time of all jobs (recall k is the number of machines).

Notice $B = \frac{1}{k} \sum_{i=1}^{n} t_i \leq m^*$, the completion time for an optimal solution, as the sum corresponds to the case where every machine takes exactly the same fraction of time to complete. Thus the completion time for machine j is

$$s_{i,j} + t_i \leq m^* + m^* = 2m^*$$

So the maximum completion time is at most twice that of an optimal solution. □

This is not the best one can do: Minimum Job Scheduling also has a PTAS (see Vazirani, 1983).

18.2.4 Randomization

Randomness is a powerful resource for algorithmic design. Upon the assumption that one has access to unbiased coins that may be flipped and their values (heads or tails) extracted, a wide array of new mathematics may be employed to aid the analysis of an algorithm. It is often the case that a simple randomized algorithm will have the same performance guarantees as a complicated deterministic (i.e. non-randomized) procedure.

One of the most intriguing discoveries in the area of algorithm design is that the addition of randomness into a computational process can sometimes lead to a significant speedup over purely deterministic methods. This may be intuitively explained by the following set of observations. A randomized algorithm can be viewed as a probability distribution on a set of deterministic algorithms. The behavior of a randomized algorithm can vary on a given input, depending on the random choices made by the algorithm; hence when we consider a randomized algorithm, we are implicitly considering a randomly chosen algorithm from a family of algorithms. If a substantial fraction of these deterministic algorithms perform well on the given input, then a strategy of restarting the randomized algorithm after a certain point in runtime will lead to a speed-up (Gomes et al., 1998).

Some randomized algorithms are able to efficiently solve problems for which no efficient deterministic algorithm is known, such as polynomial identity testing (see Motwani and Raghavan, 1995). Randomization is also a key component in the popular simulated annealing method for solving optimization problems (Kirkpatrick et al., 1983). For a long time, the problem of determining if a given number is prime (a fundamental problem in modern cryptography) was only efficiently solvable using randomization (Goldwasser and Kilian, 1986; Rabin, 1980; Solovay and Strassen, 1977). Very recently, a deterministic algorithm for primality was discovered (Agrawal et al., 2002).

Random MAX-CUT Solution We saw earlier a greedy strategy for MAX-CUT that yields a 2-approximation. Using randomization, we can give an extremely short approximation algorithm that has the same performance in approximation, and runs in expected polynomial time.

> Random-Cut: Choose a random cut (i.e. a random partition of the vertices into two sets). If there are less than $m/2$ edges crossing this cut, repeat.

THEOREM 18.9 *Random-Cut is a $\frac{1}{2}$-approximation algorithm for MAX-CUT that runs in expected polynomial time.*

Proof. Let X be a random variable denoting the number of edges crossing a cut. For $i = 1, \dots, m$, X_i will be an indicator variable that is 1 if the ith edge crosses the cut, and 0 otherwise. Then $X = \sum_{i=1}^{m} X_i$, so by linearity of expectation, $E[X] = \sum_{i=1}^{m} E[X_i]$.

Now for any edge $\{u, v\}$, the probability it crosses a randomly chosen cut is $1/2$. (Why? We randomly placed u and v in one of two possible partitions, so u is in the same partition as v with probability $1/2$.) Thus, $E[X_i] = 1/2$ for all i, so $E[X] = m/2$.

This only shows that by choosing a random cut, we expect to get at least $m/2$ edges crossing. We want a randomized algorithm that *always* returns a good cut, and its running time is a random variable whose expectation is polynomial. Let us compute the probability that $X \geq m/2$ when a random cut is chosen. In the worst case, when $X \geq m/2$ all of the probability is weighted on m, and when $X < m/2$ all of the probability is weighted on $m/2 - 1$. This makes the expectation of X as high as possible, while making the likelihood of obtaining an at-least-$m/2$ cut small. Formally,

$$m/2 = E[X] \leq (1 - \Pr[X \geq m/2])(m/2 - 1) + \Pr[X \geq m/2]m$$

Solving for $\Pr[X \geq m/2]$, it is at least $2/(m + 2)$. It follows that the expected number of iterations in the above algorithm is at most $(m + 2)/2$; therefore the algorithm runs in expected polynomial time, and always returns a cut of size at least $m/2$. □

We remark that, had we simply specified our approximation as "pick a random cut and stop", we would say that the algorithm runs in linear time, and has an *expected* approximation ratio of 1/2.

Random MAX-SAT Solution Previously, we studied a greedy approach for MAX-SAT that was guaranteed to satisfy half of the clauses. Here we will consider MAX-Ak-SAT, the restriction of MAX-SAT to CNF formulae with *at least k* literals per clause. Our algorithm is analogous to the one for MAX-CUT: *Pick a random assignment to the variables.* It is easy to show, using a similar analysis to the above, that the expected approximation ratio of this procedure is at least $1 - \frac{1}{2^k}$. More precisely, if m is the total number of clauses in a formula, the expected number of clauses satisfied by a random assignment is $m - m/2^k$.

Let c be an *arbitrary* clause of at least k literals. The probability that each of its literals were set to a value that makes them false is at most $1/2^k$, since there is a probability of $1/2$ for each literal and there are at least k of them. Therefore the probability that c is satisfied is at least $1 - 1/2^k$. Using a linearity of expectation argument (as in the MAX-CUT analysis) we infer that at least $m - m/2^k$ clauses are expected to be satisfied.

18.3 A TOUR OF APPROXIMATION CLASSES

We will now take a step back from our algorithmic discussions, and briefly describe a few of the common complexity classes associated with NP optimization problems.

18.3.1 PTAS and FPTAS

Definition PTAS and FPTAS are classes of optimization problems that some believe are closer to the proper definition of what is efficiently solvable, rather than merely P. This is because problems in these two classes may be approximated with constant ratios *arbitrarily close* to 1. However, with *PTAS*, as the approximation ratio gets closer to 1, the runtime of the corresponding approximation algorithm may grow exponentially with the ratio.

More formally, PTAS is the class of NPO problems Π that have an *approximation scheme*. That is, given $\epsilon > 0$, there exists a polynomial time algorithm A_ϵ such that

- If Π is a maximization problem, A_ϵ is a $(1+\epsilon)$ approximation, i.e. the ratio approaches 1 from the right.
- If Π is a minimization problem, it is a $(1-\epsilon)$ approximation (the ratio approaches 1 from the left).

As we mentioned, one drawback of a PTAS is that the $(1+\epsilon)$ algorithm could be exponential in $1/\epsilon$. The class FPTAS is essentially PTAS but with the additional requirement that the runtime of the approximation algorithm is polynomial in n and $1/\epsilon$.

A Few Known Results It is known that some NP-hard optimization problems cannot be approximated arbitrarily well unless $P = NP$. One example is a problem we looked at earlier, Minimum Bin Packing. This is a rare case in which there is a simple proof that the problem is not approximable unless $P = NP$.

THEOREM 18.10 *(Aho et al., 1979) Minimum Bin Packing is not in PTAS, unless $P = NP$. In fact, there is no $3/2 - \epsilon$ approximation for any $\epsilon > 0$, unless $P = NP$.*

To prove the result, we use a reduction from the Set Partition decision problem. Set Partitioning asks if a given set of natural numbers can be split into two sets that have equal sum.

Set Partition

Instance: A multi-set $S = \{r_1, \ldots, r_n\}$, where $r_i \in \mathbb{N}$ for all $i = 1, \ldots, n$.

Solution: A partition of S into sets S_1 and S_2; *i.e.* $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \varnothing$.
Measure: $m(S) = 1$ if $\sum_{r_i \in S_1} r_i = \sum_{r_j \in S_2} r_j$, and $m(S) = 0$ otherwise.

Proof. Let $S = \{r_1, \ldots, r_n\}$ be a Set Partition instance. Reduce it to Minimum Bin Packing by setting $C = \frac{1}{2}\sum_{j=1}^{n} s_j$ (half the total sum of elements in S), and considering a bin packing instance of *items* $S' = \{r_1/C, \ldots, r_n/C\}$.

If S can be partitioned into two sets of equal sum, then the minimum number of bins necessary for the corresponding S' is 2. On the other hand, if S cannot be partitioned in this way, the minimum number of bins needed for S' is at least 3, as every possible partitioning results in a set with sum greater than C. Therefore, if there were a polytime $(3/2 - \epsilon)$-approximation algorithm A, it could be used to solve Set Partition:

- If A (given S and C) returns a solution using at most $(3/2-\epsilon)2 = 3-2\epsilon$ bins, then there exists a Set Partition for S.
- If A returns a solution using at least $(3/2 - \epsilon)3 = 9/2 - 3\epsilon = 4.5 - 3\epsilon$ bins, then there is no Set Partition for S.

But for any $\epsilon \in (0, 3/2)$,

$$3 - 2\epsilon < 4.5 - 3\epsilon$$

Therefore this polynomial time algorithm distinguishes between those S that can be partitioned and those that cannot, so $P = NP$. □

A similar result holds for problems such as MAX-CUT, MAX-SAT, and Minimum Vertex Cover. However, unlike the result for Bin Packing, the proofs for these appear to require the introduction of *probabilistically checkable proofs*, which will be discussed later.

18.3.2 APX

APX is a (presumably) larger class than PTAS; the approximation guarantees for problems in it are strictly weaker. An NP optimization problem Π is in APX simply if there is a polynomial time algorithm A and constant c such that A is a c-approximation to Π.

A Few Known Results It is easy to see that $PTAS \subseteq APX \subseteq NPO$. When one sees new complexity classes and their inclusions, one of the first questions to be asked is: how likely is it that these inclusions could be made into equalities? Unfortunately, it is highly unlikely. The following relationship can be shown between the three approximation classes we have seen.

THEOREM 18.11 *(Ausiello et al., 1999)* $PTAS = APX \iff APX = NPO \iff P = NP$.

Therefore, if all NP optimization problems could be approximated within a constant factor, then $P = NP$. Further, if all problems that have constant approximations can be arbitrarily approximated, still $P = NP$. Another way of saying this is: if NP problems are hard to solve, then some of them are hard to approximate as well. Moreover, there exists a "hierarchy" of successively harder-to-approximate problems.

One of the directions stated follows from a theorem of the previous section: earlier, we saw a constant factor approximation to Minimum Bin Packing.. However, it does not have a PTAS unless $P = NP$. This shows the direction $PTAS = APX \Rightarrow P = NP$. One example of a problem that cannot be in APX unless $P = NP$ is the well-known Minimum Traveling Salesman problem.

Minimum Traveling Salesman

Instance: A set $C = \{c_1, \ldots, c_n\}$ of *cities*, and a distance function $d : C \times C \to \mathbb{N}$.
Solution: A path through the cities, i.e. a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$.
Measure: The cost of visiting cities with respect to the path, i.e.

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$$

It is important to note that when the distances in the problem instances always obey a Euclidean metric, Minimum Traveling Salesperson has a PTAS (Arora, 1998). Thus, we may say that it is the generality of possible distances in the above problem that makes it difficult to approximate. This is often the case with approximability: a small restriction on an inapproximable problem can suddenly turn it into a highly approximable one.

18.3.3 Brief Introduction to PCPs

In the 1990s, the work in probabilistically checkable proofs (PCPs) was *the* major breakthrough in proving hardness results, and arguably in theoretical computer science as a whole. In essence, PCPs only look at a few bits of a proposed proof, using randomness, but manage to capture all of NP. Because the number of bits they check is so small (a constant), when an efficient PCP exists for a given problem, it implies the hardness of *approximately solving* the same problem as well, within some constant factor.

The notion of a PCP arose from a series of meditations on proof-checking using randomness. We know NP represents the class of problems that have "short proofs" we can verify efficiently. As far as NP is concerned, all of the verification done is deterministic. When a proof is correct or incorrect, a polynomial time verifier answers "yes" or "no" with 100% confidence.

However, what happens when we relax the notion of total correctness to include probability? Suppose we permit the proof verifier to toss unbiased coins, and have *one-sided error*. That is, now a randomized verifier only accepts a correct proof with probability at least $1/2$, but still rejects any incorrect proof it reads. (We call such a verifier a *probabilistically checkable proof system*, i.e. a PCP.) This slight tweaking of what it means to verify a proof leads to an amazing characterization of NP: all NP decision problems can be verified by a PCP of the above type, which only flips $O(\log n)$ coins and only checks a *constant* ($O(1)$) number of bits of any given proof! The result involves the construction of highly intricate error-correcting codes. We shall not discuss it on a formal level here, but will cite the above in the notation of a theorem.

THEOREM 18.12 *(Arora et al., 1998)*

$$PCP[O(\log n), O(1)] = NP$$

One corollary of this theorem is that a large class of approximation problems do not admit a PTAS. In particular, we have the following theorem.

THEOREM 18.13 *For $\Pi \in$ {MAX-Ek-SAT, MAX-CUT, Minimum Vertex Cover} there exists a c such that Π cannot be c-approximated in polynomial time, unless $P = NP$.*

18.4 PROMISING APPLICATION AREAS FOR APPROXIMATION AND RANDOMIZED ALGORITHMS

18.4.1 Randomized Backtracking and Backdoors

Backtracking is one of the oldest and most natural methods used for solving combinatorial problems. In general, backtracking deterministically can take exponential time. Recent work has demonstrated that many real-world problems can be solved quite rapidly, when the choices made in backtracking are randomized. In particular, problems in practice tend to have small substructures within them. These substructures have the property that once they are solved properly, the entire problem may be solved. The existence of these so-called "backdoors" (Williams et al., 2003) to problems make them very tenable to solution using randomization. Roughly speaking, search heuristics will set the backdoor substructure early in the search, with a significant probability.

Therefore, by repeatedly restarting the backtracking mechanism after a certain (polynomial) length of time, the overall runtime that backtracking requires to find a solution is decreased tremendously.

18.4.2 Approximations to Guide Complete Backtrack Search

A promising approach for solving combinatorial problems using complete (exact) methods draws on recent results on some of the best approximation algorithms based on linear programming (LP) relaxations (see Chvatal, 1983; Dantzig, 1998) and so-called randomized rounding techniques, as well as on results that uncovered the extreme variance or "unpredictability" in the runtime of complete search procedures, often explained by so-called heavy-tailed cost distributions (Gomes et al., 2000). Gomes and Shmoys (2002) propose a *complete* randomized backtrack search method that tightly couples constraint satisfaction problem (CSP) propagation techniques with randomized LP-based approximations. They use as a benchmark domain a purely combinatorial problem, the quasigroup (or Latin square) completion problem (QCP). Each instance consists of an n by n matrix with n^2 cells. A complete quasigroup consists of a coloring of each cell with one of n colors in such a way that there is no repeated color in any row or column. Given a partial coloring of the n by n cells, determining whether there is a valid completion into a full quasigroup is an NP-complete problem (Colbourn, 1984). The underlying structure of this benchmark is similar to that found in a series of real-world applications, such as timetabling, experimental design, and fiber optics routing problems (Laywine and Mullen, 1998; Kumar et al., 1999).

Gomes and Shmoys compare their results for the hybrid techniques CSP/LP strategy guided by the LP randomized rounding approximation with a CSP strategy and with a LP strategy. The results show that the hybrid techniques approach significantly improves over the pure strategies on hard instances. This suggest that the LP randomized rounding approximation provides powerful heuristic guidance to the CSP search.

18.4.3 Average Case Complexity and Approximation

While "worst case" complexity has a very rich theory, it often feels too restrictive to be relevant to practice. Perhaps NP-hard problems are hard only for some esoteric sets of instances that will hardly ever arise. To this end, researchers have proposed theories of "average case" complexity, which attempt to probabilistically analyze problems based on randomly chosen instances over distributions; for an introduction to this line of work, see Gurevich (1991). Recently, an intriguing thread of theoretical research has explored the connections between the average-case complexity of problems and their approxima-

tion hardness (Feige, 2002). For example, it is shown that if *random 3-SAT* is hard to solve in polynomial time (under reasonable definitions of "random" and "hard"), then NP-hard optimization problems such as Minimum Bisection are hard to approximate in the worst case. Conversely, this implies that improved approximation algorithms for some problems could lead to the average-case tractability of others. A natural research question to ask is: does an PTAS imply average-case tractability, or vice versa? We suspect that some statement of this form might be the case. In our defense, a recent paper (Beier and Vocking, 2003) shows that *Random* Maximum Integer Knapsack is exactly solvable in expected polynomial time! (Recall that there exists an PTAS for Maximum Integer Knapsack.)

18.5 TRICKS OF THE TRADE

One major initial motivation for the study of approximation algorithms was to provide a new theoretical avenue for analyzing and coping with hard problems. Faced with a brand-new interesting optimization problem, how might one apply the techniques discussed here? One possible scheme proceeds as follows:

1 First, try to prove your problem is NP-hard, or find evidence that it is not! Perhaps the problem admits an interesting exact algorithm, without the need for approximation.

2 Often, a very natural and intuitive idea is the basis for an approximation algorithm. How good is a randomly chosen feasible solution for the problem? (What is the expected value of a random solution?) How about a greedy strategy? Can you define a neighborhood such that local search does well?

3 Look for a problem (call it Π) that is akin to yours in some sense, and use an existing approximation algorithm for Π to obtain an approximation for your problem.

4 Try to prove it cannot be approximated well, by reducing some hard-to-approximate problem to your problem.

The first, third, and fourth points essentially hinge on one's resourcefulness: one's tenacity to scour the literature (and colleagues) for problems similar to the one at hand, as well as one's ability to see the relationships and reductions which show that a problem is indeed similar.

This chapter has been mainly concerned with the second point. To answer the questions of that point, it is crucial to prove *bounds* on optimal solutions, with respect to the feasible solutions that one's approaches obtain. For minimization (maximization) problems, one will need to prove *lower bounds* (re-

spectively, upper bounds) on some optimal solution for the problem. Devising lower (or upper) bounds can simplify the proof tremendously: one only needs to show that an algorithm returns a solution with value at most c times the lower bound to show that the algorithm is a c-approximation.

We have proven upper and lower bounds repeatedly (implicitly or explicitly) in our proofs for approximation algorithms throughout this chapter—it may be instructive for the reader to review each approximation proof and find where we have done it. For example, the greedy vertex cover algorithm (of choosing a maximal matching) works because even an optimal vertex cover covers at least one of the vertices in each edge of the matching. The number of edges in the matching is a lower bound on the number of nodes in a optimal vertex cover, and thus the number of nodes in the matching (which is twice the number of edges) is at most twice the number of nodes of an optimal cover.

18.6 CONCLUSIONS

We have seen the power of randomization in finding approximate solutions to hard problems. There are many available approaches for designing such algorithms, from solving a related problem and tweaking its solution (in linear programming relaxations) to constructing feasible solutions in a myopic way (via greedy algorithms). We saw that for some problems, determining an approximate solution is vastly easier than finding an exact solution, while other problems are just as hard to approximate as they are to solve.

In closing, we remark that the study of approximation and randomized algorithms is still a very young (but rapidly developing) field. It is our sincerest hope that the reader is inspired to contribute to the prodigious growth of the subject, and its far-reaching implications for problem solving in general.

SOURCES OF ADDITIONAL INFORMATION

Books on Algorithms

- Data Structures and Algorithms (Aho et al., 1983)
- Introduction to Algorithms (Cormen et al., 2001)
- The Design and Analysis of Algorithms (Kozen, 1992)
- Combinatorial Optimization: Algorithms and Complexity (Papadimitriou and Steiglitz, 1982)

Material on Linear Programming and Duality

- Chapter 3 of this book
- Linear Programming (Chvatal, 1983)

- Linear Programming and Extensions (Dantzig, 1998)
- Integer and Combinatorial Optimization (Nemhauser and Wolsey, 1988)
- Combinatorial Optimization: Algorithms and Complexity (Papadimitriou and Steiglitz, 1982)
- Combinatorial Optimization (Cook et al., 1988)
- Combinatorial Optimization Polyhedra and Efficiency (Schrijver, 2003)

Books on Approximation Algorithms

- Complexity and Approximation (Ausiello et al., 1999)
- Approximation Algorithms for NP-Hard Problems (Hochbaum, 1997)
- Approximation algorithms (Vazirani, 1983)

Books on Probabilistic and Randomized Algorithms

- An Introduction to Probability Theory and Its Applications (Feller, 1971)
- The Probabilistic Method (Alon and Spencer, 2000)
- Randomized Algorithms (Motwani and Raghavan, 1995)
- The Discrepancy Method (Chazelle, 2001)

Surveys

- Computing Near-Optimal Solutions to Combinatorial Optimization Problems (Shmoys, 1995)
- Approximation algorithms via randomized rounding: a survey (Srinivasan)

Courses and Lectures Notes Online

- Approximability of Optimization Problems, MIT, Fall 99 (Madhu Sudan), http://theory.lcs.mit.edu/ madhu/FT99/course.html
- Approximation Algorithms, Fields Institute, Fall 99 (Joseph Cheriyan), http://www.math.uwaterloo.ca/ jcheriya/App-course/course.html
- Approximation Algorithms, Johns Hopkins University, Fall 1998 (Lenore Cowen), http://www.cs.jhu.edu/ cowen/approx.html
- Approximation Algorithms, Technion, Fall 95 (Yuval Rabani), http://www.cs.technion.ac.il/ rabani/236521.95.wi.html

- Approximation Algorithms, Cornell University, Fall 98 (D. Williamson), http://www.almaden.ibm.com/cs/people/dpw/
- Approximation Algorithms, Tel Aviv University, Fall 01 (Uri Zwick), http://www.cs.tau.ac.il/7Ezwick/approx-alg-01.html
- Approximation Algorithms for Network Problems, (Cheriyan and Ravi) http://www.gsia.cmu.edu/afs/andrew/gsia/ravi/WWW/new-lecnotes.html
- Randomized algorithms, CMU, Fall 2000 (Avrim Blum), http://www-2.cs.cmu.edu/afs/cs/usr/avrim/www/Randalgs98/home.html
- Randomization and optimization by Devdatt Dubhashi, http://www.cs.chalmers.se/ dubhashi/ComplexityCourse/info2.html
- Topics in Mathematical Programming: Approximation Algorithms, Cornell University, Spring 99 (David Shmoys), http://www.orie.cornell.edu/ or739/index.html
- Course notes on online algorithms, randomized algorithms, network flows, linear programming, and approximation algorithms (Michel Goemans), http://www-math.mit.edu/ goemans/

Main Conferences Covering the Approximation and Randomization Topics

- IPCO: Integer Programming and Combinatorial Optimization
- ISMP: International Symposium on Mathematical Programming
- FOCS: Annual IEEE Symposium on Foundation of Computer Science
- SODA: Annual ACM-SIAM Symposium on Discrete Algorithms
- STOC: Annual ACM Symposium on Theory of Computing
- RANDOM: International Workshop on Randomization and Approximation Techniques in Computer Science
- APPROX: International Workshop on Approximation Algorithms for Combinatorial Optimization Problems

References

Agrawal, M., Kayal, N. and Saxena, N., 2002, Primes in P, www.cse.iitk.ac.in/news/primality.html

Aho, A. V., Hopcroft, J. E. and Ullman, J. D., 1979, *Computers and intractability: A guide to NP-Completeness,* Freeman, San Francisco.

Aho, A. V., Hopcroft, J. E. and Ullman, J. D., 1983, *Data structures and Algorithms,* Computer Science and Information Processing Series, Addison-Wesley, Reading, MA

Alon, N. and Spencer, J., 2000, *The Probabilistic Method,* Wiley, New York

Arora, S., 1998, Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems, *J. ACM* **45**:753–782.

Arora, S., Lund, C., Motwani, R., Sudan, M. and Szegedy, M., 1998, Proof verification and the hardness of approximation problems, *J. ACM* **45**:501–555.

Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A. and Protasi, M., 1999, *Complexity and Approximation,* Springer, Berlin.

Beier, R. and Vocking, B., 2003, Random knapsack in expected polynomial time, *Proc. ACM Symp. on Theory of Computing,* pp. 232–241.

Chazelle, B., 2001, *The Discrepancy Method,* Cambridge University Press, Cambridge.

Chvatal, V., 1979, A greedy heuristic for the set-covering, *Math. Oper. Res.* **4**:233–235.

Chvatal, V., 1983, *Linear Programming,* Freeman, San Francisco.

Clay Mathematics Institute, 2003, The millenium prize problems: P vs. NP, http://www.claymath.org/

Colbourn, C., 1984, The complexity of completing partial Latin squares, *Discrete Appl. Math.* **8**:25–30.

Cook, W., Cunningham, W., Pulleyblank, W. and Schrijver, A., 1988, *Combinatorial Optimization,* Wiley, New York.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C, 2001, *Introduction to Algorithms,* MIT Press, Cambridge, MA.

Dantzig, G., 1998, *Linear Programming and Extensions,* Princeton University Press, Princeton, NJ.

Feige, U., 2002, Relations between average case complexity and approximation complexity, in: *Proc. ACM Symp. on Theory of Computing*, pp. 534–543.

Feller, W., 1971, *An Introduction to Probability Theory and Its Applications,* Wiley, New York.

Garey, M. R., Graham, R. L. and Ulman, J. D., 1972, Worst case analysis of memory allocation algorithms, in: *Proc. ACM Symp. on Theory of Computing,* pp. 143–150.

Goemans, M. X. and Williamson, D. P., 1995, Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming, *J. ACM* **42**:1115–1145.

Goldwasser, S. and Kilian, J., 1986, Almost all primes can be quickly certified, in: *Proc. Annual IEEE Symp. on Foundations of Computer Science,* pp. 316–329.

Gomes, C., Selman, B., Crato, N. and Kautz, H., 2000, Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, *J. Autom. Reason.* **24**:67–100.

Gomes, C. P., Selman, B. and Kautz, H., 1998, Boosting combinatorial search through randomization, in: *Proc. 15th National Conf. on Artificial Intelligence (AAAI-98),* AAAI Press, Menlo Park, CA.

Gomes, C. P. and Shmoys, D., 2002, The promise of LP to boost CSP techniques for combinatorial problems, in: *Proc. 4th Int. Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR'02),* Le Croisic, France, N. Jussien and F. Laburthe, eds, pp. 291–305.

Graham, R. L., 1966, Bounds for certain multiprocessing anomalies, *Bell Syst. Tech. J.* **45**:1563–1581.

Gurevich, Y., 1991, Average Case Complexity, in: *Proc. 18th Int. Colloquium on Automata, Languages, and Programming (ICALP'91),* Lecture Notes in Computer Science, Vol. 510, pp. 615–628, Springer, Berlin.

Hochbaum, D. S., 1997, ed., *Approximation Algorithms for NP-Hard Problems,* PWS Publishing Company, Boston, MA.

Johnson, D. S., 1974, Approximation algorithms for combinatorial problems, *J. Comput. Syst. Sci.* **9**:256–278.

Khot, S. and Regev, O., 2003, Vertex cover might be hard to approximate within 2-e, in: *Proc. IEEE Conf. on Computational Complexity.*

Kirkpatrick, S., Gelatt, C. and Vecchi, M., 1983, Optimization by simulated annealing, *Science* **220**:671–680.

Kozen, D., 1992, *The design and analysis of algorithms,* Springer, New York.

Kumar, S. R., Russell, A. and Sundaram, R., 1999, Approximating Latin square extensions, *Algorithmica* **24**:128–138.

Laywine, C. and Mullen, G., 1998, *Discrete Mathematics using Latin Squares,* Discrete Mathematics and Optimization Series, Wiley-Interscience, New York.

Motwani, R. and Raghavan, P., 1995, *Randomized Algorithm,* Cambridge University Press, Cambridge.

Nemhauser, G. and Wolsey, L., 1988, *Integer and Combinatorial Optimization,* Wiley, New York.

Papadimitriou, C. and Steiglitz, K., 1982, *Combinatorial Optimization: Algorithms and Complexity,* Prentice-Hall, Englewood Cliffs, NJ.

Rabin, M. (1980). Probabilistic algorithm for testing primality, *J. Number Theory* **12**:128–138.

Schrijver, A., 2003, *Combinatorial Optimization Polyhedra and Efficiency,* Springer, Berlin.

Shmoys, D., 1995, Computing near-optimal solutions to combinatorial optimization problems, in: *Combinatorial Optimization,* Discrete Mathematics

and Theoretical Computer Science Series, W. Cook, L. Lovasz and P. Seymour, eds, American Mathematical Society, Providence, RI.

Solovay, R. and Strassen, V., 1977, A fast Monte Carlo test for primality, *SIAM J. Comput.* **6**:84–86.

Srinivasan, A., Approximation algorithms via randomized rounding: a survey. Available from: citeseer.nj.nec.com/493290.html

Vazirani, V., 1983, *Approximation Algorithms,* Springer, Berlin.

Williams, R., Gomes, C. P. and Selman, B., 2003, Backdoors to typical case complexity, In *Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI).*