# Chapter 21
# Approximations and Randomization

Carla P. Gomes and Ryan Williams

## 21.1 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. Computer scientists study 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. 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.

A large class of common optimization problems are classified as *NP-hard*. 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 (TSP), 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

C.P. Gomes
Department of Computer Science, Cornell University, Ithaca, NY, USA
e-mail: gomes@cs.cornell.edu

R. Williams (✉)
Computer Science Department, Stanford University, Stanford, CA, USA
e-mail: rrw@cs.stanford.edu

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.

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 simulated annealing and tabu search (see Chaps. 9 and 10), 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 run-time. 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 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 $\alpha$, an algorithm $\mathcal{A}$ is an $\alpha$-approximation algorithm for a given minimization problem $\Pi$ if its solution is at most $\alpha$ times the optimum, considering all the possible instances of problem $\Pi$.

The focus of this chapter is on the design of approximation algorithms for NP-hard optimization problems. We will show how standard algorithm design techniques such as greedy and local search methods, dynamic programming, and classical methods of discrete optimization such as linear programming and semidefinite programming 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 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, *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 Sect. 21.2 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 TSP, 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 Sect. 21.3 we provide a tour of the main approximation classes, including a brief introduction to techniques to proof lower bounds on approximability. In Sect. 21.4 we describe some promising areas of application of approximation algorithms. Section 21.5 summarizes the chapter and we conclude by suggesting additional sources of information on approximation and randomization methods.

## 21.2 Approximation Strategies

### *21.2.1 Preliminaries*

#### 21.2.1.1 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 $I$,

$m_I : Sol(I) \to R$ 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 $\Pi$ 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 of $I$ is all undirected graphs, while $Sol(I)$ is all possible subsets of vertices in a graph.

#### 21.2.1.2 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 $\Pi$ be an optimization problem. We say that an algorithm $A$ feasibly solves $\Pi$ 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 $\Pi$. 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.

#### 21.2.1.3 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 $I$. Note we may also represent a decision problem as a subset $S$ of the set of all possible instances: members of $S$ represent instances where the measure is 1.

Informally, P (polynomial time) is defined as the class of decision problems $\Pi$ for which there exists a corresponding algorithm $A_\Pi$. such that every instance $I \in \Pi$. is solved by $A_\Pi$ 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.

NP (non-deterministic polynomial time) is defined as the class of decision problems $\Pi$ for which there exists a corresponding decision problem $\Pi'$ 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, we can determine if $I$ is in an NP problem efficiently if, given an instance $I$, one is also provided with a short "proof" $C$, which is of length polynomial in $I$. 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 $\Pi$. 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.

### *21.2.2 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.

#### 21.2.2.1 Greedy Vertex Cover

The minimum vertex cover problem was defined in the previous section. Variants on the problem come up in many areas of optimization research. A simple greedy algorithm is a 2-approximation to the problem, and no better approximation algorithms are known! In fact, it is widely believed that one cannot approximate minimum vertex cover better than $2 - \varepsilon$ for any $\varepsilon > 0$, unless $P = NP$ (see Khot and Regev 2003). The 2-approximation 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 21.1.** *Greedy-VC is a 2-approximation algorithm for Minimum Vertex Cover.*

*Proof.* First, we claim $S$ 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 the algorithm had completed, which is a contradiction.

The next step is to show that any vertex cover of the graph contains at least $|S|/2$ vertices. $|S|/2$ is the number of edges we chose during the run of the algorithm, and none of them share any endpoints. Hence at least one vertex in any vertex cover of the given graph must be assigned to each edge we choose. It follows that any optimal solution has at least $|S|/2$ nodes, so our algorithm has a $|S|/|S^*| = 2$ approximation ratio. □

Sometimes when one proves that an algorithm has a certain approximation ratio, the analysis is somewhat *loose*, and may not reflect the best possible 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 Fig. 21.1.

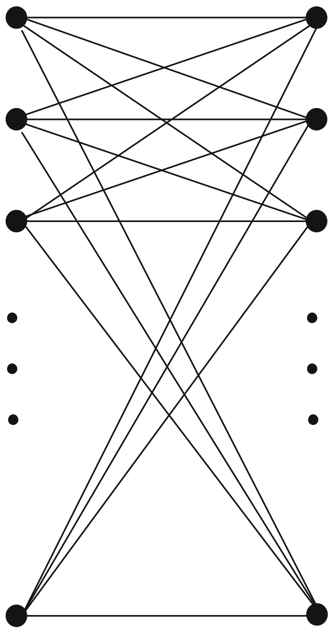


**Fig. 21.1** Instances that correspond to bipartite graphs $K_{n,n}$. When running greedy-VC on these instances, the algorithm will select all $2n$ vertices

### 21.2.2.2 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 MAXSAT? 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 its corresponding variable in such a way that all clauses containing it 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 21.2.** *Greedy-MAXSAT is a 1/2-approximation algorithm for MAXSAT.*

*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 $\frac{1}{2}l_i$, respectively. After $v_i$ is set so that $l_i$ is true (so both $l_i$ and $\frac{1}{2}l_i$ disappear from $F$), there are at least $m - m_{POS} - m_{NEG}$ clauses left, on $n-1$ variables.

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

### 21.2.2.3 Greedy MAX-CUT

Our next example shows how local search may be employed in designing approximation algorithms. Local search is inherently a greedy strategy: when we have a feasible solution $x$, we try 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:

*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.

$$|\{\{u,v\} \in E | u \in S, v \in T\}|.$$

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 21.3.** *Local-Cut is a 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.

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

Using the well-known degree-edge equation $\sum_{v\in V}\deg(v) = 2m$, we conclude that

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

It follows that the approximation ratio of the algorithm is

$$\frac{m^*}{m} \geq \frac{1}{2}. \qquad \square$$

It turns out that MAX-CUT admits much better approximation ratios than 1/2; a relaxation of the problem to a semidefinite linear program yields a 0.8786 approximation (see Goemans and Williamson 1995). However, like many optimization problems, MAX-CUT cannot be approximated arbitrarily well $(1-\varepsilon, \text{forall } \varepsilon > 0)$ unless $\mathrm{P} = \mathrm{NP}$. That is to say, it is unlikely that MAX-CUT is in the PTAS complexity class.

#### 21.2.2.4 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 $C \in N$, and a number of items $n \in N$, 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 $C$.

The following is left as an exercise for the reader.

**Lemma 21.1.** *Greedy-FKS solves Maximum Fraction Knapsack in polynomial time.*

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

**Theorem 21.4.** *Greedy-KS is a 1/2-approximation for Maximum Integer Knapsack.*

*Proof.* Fix an instance of the problem. Let $P = \sum_{j \in S} p_i$, 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 $P_F^* = P_I^*$. By the analysis of the algorithm for Fraction Knapsack, $P_F^* = P + p_j$, where $\varepsilon \in (0,1]$ is the fraction chosen for item $j$ in the algorithm. Therefore,

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

and we are done. □

Later, we will see how this algorithm can be extended (in conjunction with dynamic programming) to get a PTAS for Maximum Integer Knapsack. A PTAS is quite powerful; such a scheme can approximately solve a problem with arbitrarily close ratios to the optimal solution. However, many problems provably do not have a PTAS, unless $P = NP$.

## *21.2.3 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.

### 21.2.3.1 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 online 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 *onï¿½line* 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 21.5.** *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 one 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 21.6.** *First-Bin is a 11/9-approximation to Minimum Bin Packing.*

#### 21.2.3.2 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: $\max\{\sum_{i:a(i)=j} t_i | j \in \{1, \ldots, k\}\}$.

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 21.7.** *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, $s_{i,j} \leq 1/k \sum_{i=1}^{n} t_i$, i.e. $s_{i,j}$ is less $1/k$ of the total $k$ time of all jobs (recall that $k$ is the number of machines).

Notice $B = 1/k \sum_{i=1}^{n} t_i = m^*$, 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

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

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

Minimum Job Scheduling also has a PTAS, as we will discover in the next section.

## *21.2.4 Dynamic Programming*

Dynamic programming is, in essence, a formal name given to the tried-and-true principle of *saving your work*. It is often the case in algorithmic design that, when optimal solutions to small pieces of an instance are known, those solutions can be merged in a way that yields an optimal solution for the entire instance. This is precisely the idea behind dynamic programming: we solve pieces of the problem and save their solutions in order to solve the entire problem later on. We will demonstrate how dynamic programming may be used to extend the previous 2-approximation for Minimum Job Scheduling into a PTAS for the problem. Dynamic programming will also be employed to get a FPTAS for Maximum Integer Knapsack.

### 21.2.4.1 PTAS for Minimum Job Scheduling

The algorithm we present is based on one given by Vazirani (2004). It has the property that, for any fixed $\varepsilon > 0$ provided beforehand, it returns a $(1+\varepsilon)$-approximate solution. Further, the runtime is polynomial in the input size, *provided that* $\varepsilon$ *is constant*. This allows us to specify a runtime that has $1/\varepsilon$ in the exponent (which we will do). It is typical to view this algorithm as a *family* of successively better (but also slower) approximation algorithms, each running with a successively smaller $\varepsilon > 0$. This is intuitively why they are called an approximation scheme; *scheme* is meant to suggest that a variety of algorithms are used.

Our treatment here will be lighter than in other sections due to the complexity of the algorithm. We will name a special case problem that is polynomial time solvable. The polytime algorithm will serve as a subroutine to a parameterized Job Scheduling procedure, where we are allowed to specify the completion time in the input. The PTAS will use this parameterized procedure to perform a binary search over the possible completion times, finding an approximate minimum completion time.

The special case problem we consider is Minimum Unit-Time $c$-Job Scheduling, a variant on Minimum Job Scheduling which is polynomial time solvable. It is similar to job scheduling, except (a) the number of distinct possible times for jobs is a constant $c$, and (b) the completion time is fixed to be 1. (We stress that the number of jobs is not constant—the number of distinct times for jobs is.) The problem now is to minimize the number of machines necessary to schedule these jobs so they may be completed in one time unit. (The astute reader will find that this modified problem is essentially bin packing.)

**Theorem 21.8.** *Minimum Unit-Time c-Job Scheduling is polynomial time solvable.*

*Proof.* Arbitrarily order the possible job times $T_1,\dots,T_c$. Observe that any instance (a multi-set $S = \{t_1,\dots,t_n\}$ of items) of this problem may be specified as a $c$-tuple $(n_1,\dots,n_c)$ of natural numbers: the $i$th component of the $c$-tuple $(n_i)$ gives the number of job with time $T_i$.

Define $OPT(n_1,\dots,n_c)$ to be the optimal solution (minimum number of machines necessary) for the instance $(n_1,\dots,n_c)$. We will determine $OPT$ for a given instance via dynamic programming, i.e. by determining its value on all possible $(i_1,\dots,i_c)$, where each $i_j$ ranges from 0 to $n_j$. (Note there are only $O(n^c)$ such $c$-tuples.)

Initially, we determine for all $1 \le j \le c$, and $0 \le i_j \le n_j$, those $(i_1,\dots,i_c)$ for which $OPT(i_1,\dots,i_c) = 1$ (exactly one machine suffices). Doing this takes constant time for each of the $O(n^c)$ $c$-tuples. Add these $c$-tuples to a *saved value* set, $S$.

For the remaining $(i_1,\dots,i_c)$ (those whose $OPT$ value is larger than 1), we determine their $OPT$ solution via the inductive assignment

$$OPT(i_1,\dots,i_c) := 1 + \min_{(j_1,\dots,j_c)\in S} OPT(i_1 - j_c,\dots,i_c - j_c)$$

provided that $\forall l = 1,\dots,c, i_l \ge j_l$. A bit of thought shows that every possible $c$-tuple's $OPT$ value may be computed in this way, provided the $c$-tuples are processed in the correct order.

An $OPT$ value for a fixed $c$-tuple, assuming all of the other necessary $OPT$ values are present, requires at most $O(n^c)$ time to compute. As there are $O(n^c)$ possible $c$-tuples, the overall runtime is $O(n^{2c})$. □

As expected, this dynamic programming algorithm will be a component in our description of the PTAS. However, before we give the PTAS, we need to describe another procedure that will use the dynamic programming algorithm.

Let $\varepsilon > 0$ be fixed, and $(T = \{t_1,\dots,t_n\}, k)$ be a job scheduling instance (where $k$ is number of machines to be used). From the analysis of the 2-approximation, there exists a (polytime computable) lower bound $B$ on the optimum, and an upper bound, which is $2B$. Let $V \in [B, 2B]$. The next algorithm, Parameter-JS, returns a $(1+\varepsilon)$-approximate solution for $T$, given that the parameter $V$ is the desired completion time. Let Unit-cJS be the algorithm from the above theorem.

Parameter-JS$(S,V,\varepsilon)$: Define a job time $t_j$ to be short if $t_j < V$. Remove the short jobs from $T$. For the remaining jobs, *round their sizes down* to the nearest power of $(1+\varepsilon)$, times $\varepsilon V$. More precisely, for each $t_j$ remaining in $T$, compute the unique $x$ such that $t_j \in [(1+\varepsilon)x \cdot \varepsilon V, (1+\varepsilon)^{x+1} \cdot \varepsilon V]$, and set $t'_j := (1+\varepsilon)^x \cdot \varepsilon V$.

Let $T' = \{t'_j | t_j \in T\}$. Now the possible times for jobs in $T'$ are $\varepsilon V$, $(1+\varepsilon)V$, $(1+\varepsilon)2V$, and so on. There are therefore

$$k = \left\lceil \frac{\log(1/\varepsilon)}{\log(1+\varepsilon)} \right\rceil$$

distinct possible times. Further, all of these times are a multiple of $V$.

Let $T'' = \{t_j/V | t_j \in T\}$. Call Unit-cJS on $T''$ and the above $k$, getting back a solution $S$. As Unit-cJS assumes the completion time to be 1, the resulting Unit-cJS solution for $T''$ is also a solution for $T'$, where the completion time is $V$. Greedily schedule the short jobs back in this solution, on whatever machines will accommodate them (without exceeding the completion time $V$). Introduce new machines for handling short jobs only when the job cannot be scheduled on any of the existing machines without exceeding $V$. Return the resulting solution.

Let us observe a lemma, which follows from a simple analysis of the short jobs' impact on the Unit-kJS solution for $T'$.

**Lemma 21.2.** *Parameter-JS on $T', V$, and $\varepsilon$ returns a job schedule for a minimum number of machines, with completion time that is at most $(1+\varepsilon)V$.*

*Proof.* (Sketch) Each job's time $t'_j \in T'$ is rounded down by at most a $(1+\varepsilon)$ factor from the original $t_j$, the solution returned by the algorithm is feasible if the minimum completion time in the original instance $T$ (modulo short jobs) is $(1+\varepsilon)V$. □

Let $OPT(T,V)$ be the minimum number of machines needed to schedule the jobs of instance T when the completion time is exactly $V$, and let $M(T,V)$ be the number of machines used by Parameter-JS$(S,V,\varepsilon)$.

**Lemma 21.3.** $M'(T,V) \leq OPT(T,V)$.

The proof involves an analysis of short jobs, and is left to the reader.

Finally, we are in a position to describe the PTAS. We wish to find the optimal completion time $V^*$ for a given number of machines $k$, which is located somewhere in the interval $[B, 2B]$ (recall that $B$ and $2B$ are lower and upper bounds on the optimal, respectively). The above lemma implies

$$\min\{V | M(T,V) = k\} = \min\{V | OPT(T,V) = k\} = V^*.$$

To find $V^*$, the PTAS will perform a binary search over fractions in the interval $[B, 2B]$, calling Parameter-JS on the current $V$-value, until the remaining interval size is at most $B$. At this point, we will argue that the solution obtained is approximately close to an optimal one:

```
PTAS-MJS$(T,k,\varepsilon)$:
  Initially, $F := B$ (first), $L := 2B$ (last).
  Repeat $\lceil \log_2(3/\varepsilon) \rceil$ times:
    Set $M := (F+L)/2$, the *midpoint* of the interval.
```

Call Parameter-JS$(S, M, \varepsilon/3)$, getting a solution $S'$.
If $S'$ uses at most $k$ machines, then set $L := M$; if not, set $F := M$. End Repeat.
Return $L$ and $S$.

Based on the runtime of Parameter-JS, it is easy to verify that PTAS-MJS runs in $O(\log_2(1/\varepsilon)n^{\lceil 2\log_{1+\varepsilon}(1/\varepsilon)\rceil})$ time.

**Theorem 21.9.** *PTAS-MJS is a* $(1+\varepsilon)$*-approximation for Minimum Job Scheduling.*

*Proof.* We first prove that the $L$ returned is at most $(1+\varepsilon)V^*$. When the repeat-loop is completed, $|F-L| \le \varepsilon B$, as the initial length of the interval is $B$ and each iteration of the loop decreases that length by a factor of 2.

Hence
$$L - B = \min\{V | M'(T,V) \le k\} \le L$$
implying (via the lemma)
$$L = \min\{V | M'(T,V) \le k\} + \varepsilon B \le \min\{V | OPT(T,V) \le k\} + \varepsilon V^* = (1+\varepsilon)V^*.$$
Applying Lemma 21.2, we infer that the value of $S$ returned by PTAS-MJS is
$$M'(T,V) \le (1+\varepsilon/3)M \le (1+\varepsilon/3)L \le (1+\varepsilon/3)2V^* \le (1+\varepsilon)V^*.$$
Therefore, the $S'$ returned is within $1+\varepsilon$ of the optimal minimum solution. □

### 21.2.4.2 FPTAS for Knapsack

Our next example for which we can get arbitrarily good approximations will have the added benefit that, as the approximation ratio of the algorithm improves, the runtime does not get larger than a fixed polynomial in $n$. This is known as a *fully* polynomial time approximation scheme.

The strategy we will use for developing the FPTAS is similar to that for Job Scheduling, but the algorithm itself will be much simpler. We first look at a special case of Maximum Integer Knapsack where some parameter in the problem is held constant, and show it to be polynomial time solvable via dynamic programming. Then we use this polynomial time algorithm to derive an approximation algorithm for the general problem.

Recall that in the Minimum Integer Knapsack problem we have 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 $i = 1, \ldots, n$. The objective is to pack a knapsack of capacity $C$ with items of cost at most $C$, and maximum profit. Here, the special case problem will be Maximum $k$-Profit Integer Knapsack; in this restriction, all of the items' profits are bounded from above by a constant $k$.

**Theorem 21.10.** *Maximum k-Profit Integer Knapsack is solvable in polynomial time.*

*Proof.* Given an instance of the problem with $n$ items (labeled $1,\ldots,n$), since all profits are bounded from above by $k$, $kn$ is an upper bound on the optimum profit for the instance. We will now set up a dynamic program that computes optimum solution for subsets of the instance, from 0 profit up to $kn$ profit.

For every $i = 1,\ldots,n$ and $j = 1,\ldots,kn$, define $OPT(i,j)$ to be the minimum capacity of a packing over items $1,\ldots,i$ that has profit exactly $j$ (and $OPT(i,j) = \infty$ if no packing exists).

Note that $OPT(1,j) = c_1$ if $p_1 = j$, and $\infty$ otherwise. To determine $OPT$ for $i > 1$, we use the following inductive equation:

$$OPT(i,j) := \begin{cases} \min\{OPT(i-1,j), OPT(i,j-p_i)+c_i\} & \text{if} \quad p_i < j \\ OPT(i-1,j) & \text{otherwise.} \end{cases}$$

Intuitively, this just means that we either add the $i$th item to the solution or not, depending on which is smaller and whether or not the profit of item $i$ exceeds the total profit $j$. For each $OPT(i,j)$ value, we also save a subset of $\{1,\ldots,i\}$ with profit $j$ that achieves the $OPT(i,j)$ value.

Using the inductive equation, we can compute all $OPT$ values in $O(kn^2)$ time. Notice that the maximum profit possible for the instance is

$$\max\{j : j = 1,\ldots,kn, OPT(n,j) < \infty\}$$

i.e. the maximum profit over the feasible $OPT$ values. Hence, after the $OPT$ values are computed, it takes $O(kn)$ time to find an optimal solution $S'$ corresponding to the maximum profit. □

The central idea behind our FPTAS for the general Integer Knapsack problem is to *discard least significant bits* of the profits of items. That is, we divide all of the profits of items by some parameter (depending on ε), reducing the instance to one that has a small upper bound $k$ on the profit size. Then the above algorithm will be used to determine an exact solution $S'$ for this reduced instance, which roughly corresponds to an approximate solution for the original instance. Finally, we will use a trick from the 2-approximation for Knapsack: we will either return the solution $S'$, or the most profitable item of cost at most $C$, whichever is better.

Fix an $\varepsilon > 0$. Let Dyn-MBIK be the polynomial-time algorithm for Maximum $k$-Profit Integer Knapsack from Theorem 21.10.

FPTAS-MIK: For all $i = 1,\ldots,n$, re-define $p_i := \lceil \frac{p_i \cdot n}{\max_j \varepsilon p_j} \rceil$. Run Dynï¿½MBIK, and get a solution $S'$. Let $i$ be s.t. $p_i = \max_j p_j$. Return $S'$. or $\{i\}$, depending on which has higher profit.

**Theorem 21.11.** *FPTAS-MIK runs in* $O(n^3/\varepsilon)$ *time, and is a* $(1-\varepsilon)$*-approximation.*

Notice that the runtime is now polynomial in $n$ and $1/\varepsilon$, so this algorithm indeed qualifies as an FPTAS for the problem.

*Proof.* First, from the analysis of Dyn-MBIK, the runtime of FPTAS-MIK is

$$\mathrm{O}\left(n^2 \max_i \left\lceil \frac{p_i \frac{1}{2} n}{\max_j \varepsilon p_j} \right\rceil\right),$$

i.e. $\mathrm{O}(n^3/\varepsilon)$ time.

Now we will show that the algorithm is a $(1-\varepsilon)$-approximation. Let $S^*$ be a subset of $\{1,\ldots,n\}$ that is an optimal solution for an instance, let $S'$ be the subset returned by the Dyn-MBIK call, and let $P(S^*)$ (respectively $P(S')$) be the corresponding profits. Our goal is to show that the profit of the set returned by the algorithm (call it $P'$) is at least $(1-\varepsilon)P(S^*)$.

Let $P(S^*)$ be the profit of $S^*$, under the redefined profits given in the algorithm. It is straightforward to verify that

$$P(S^*) \le \max_j \varepsilon p_j + (\max_j p_j/n)\frac{1}{2}P'(S^*).$$

Dyn-MBIK returns the optimal solution under redefined profits, hence

$$P(S') \ge (\max_j \varepsilon p_j/n)\frac{1}{2}P'(S^*) \ge P(S^*) - \max_j \varepsilon p_j.$$

By definition of the algorithm, the profit of the solution returned $P'$ is at least $\max_j p_j$. Since $P' \le P(S^*)$, it follows that

$$P' \ge P(S') \ge P(S^*) - \max_j \varepsilon p_j \ge P(S^*) - \varepsilon P' \ge (1-\varepsilon)P(S^*).$$ □

## 21.2.5 LP-Based Algorithms

Linear programming (LP) plays an important role on the design of algorithms for combinatorial optimization problems (see e.g. Chvatal 1983; Papadimitriou and Steiglitz 1982; Schrijver 2003). The use of LP for the design and analysis of approximation algorithms for NP-hard problems dates back to the 1970s (Chvatal 1979; Lovasz 1975; see also Wolsey 1980). In this section we show how we can use LP techniques to design approximation algorithms. We start by showing how LP rounding can be used to derive a 2-approximation algorithm for the minimum-weight vertex cover problem. We then describe the primal–dual method, based on LP duality. We provide background material on LP duality and its most important theorems.

#### 21.2.5.1 LP Rounding

A simple way of obtaining an approximation algorithm based on linear programming is to solve the problem as a linear program and convert its fractional solution into an integral solution, by *rounding* it. We use LP rounding to approximate the Minimum-Weight Vertex Cover problem (Hochbaum 1982, 1983).

*Minimum-Weight Vertex Cover*
*Instance:* An undirected graph $G = (V, E)$, and a positive weight function, $W : V \rightarrow R^+$ on the vertices.
*Solution:* A subset $S \subseteq V$ such that for every $\{u, v\} \in E$, either $u \in S$ or $v \in S$.
*Measure:* $\sum_{v \in S} w(v)$.

In order to formulate this problem as a linear program, let us associate a variable $x(v)$ with each vertex $v \in V$, and require that $x(v) \in \{0, 1\}$, for each $v \in V$. The interpretation of the $x(v)$ is the following: if $x(v) = 1$, the vertex $v$ belongs to the vertex cover; if $x(v) = 0$, the vertex $v$ does not belong to the vertex cover. The constraint that requires that for any edge $(u, v)$, at least one of the $u$ and $v$ must be in the vertex cover, can be stated as $x(u) + x(v) = 1$. We obtain the following integer program:

*0–1 integer program*

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w(v)x(v) & \\ \text{subject to :} & x(u) + x(v) \geq 1 \text{ for each} & v \in V \\ & x(v) \in \{0, 1\} \quad\text{ for each} & v \in V. \end{array}$$

Finding the solution to this integer program is NP-hard. However, if we relax the constraint $x(v) \in \{0, 1\}$, and replace with $0 \leq x(v) \leq 1$, we obtain what is called its LP relaxation:

*LP relaxation of the 0–1 integer program*

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w(v)x(v) & \\ \text{subject to :} & x(u) + x(v) \geq 1 \text{ for each} & v \in V \\ & x(v) \geq 0 \quad\quad\quad\text{ for each} & v \in V \\ & x(v) \geq 1 \quad\quad\quad\text{ for each} & v \in V. \end{array}$$

Note that any feasible solution to the 0–1 integer program is also a feasible solution to its LP relaxation. Therefore, an optimal solution to the LP relaxation is a lower bound on the optimal solution of the 0–1 integer program.

LP-Rounding-MVC: Formulate the minimum-weight vertex cover as the above relaxed linear program. Compute its optimal solution $x^*$. Initially, we have an empty set $S$. For each vertex $v \in V$, if $x^*(v) \geq 1/2$, add $v$ to $S$. Return $S$ as the minimum-weight vertex cover.

**Theorem 21.12.** *LP-Rounding-MVC is a 2-approximation algorithm for Minimum Weight Vertex Cover.*

*Proof.* We start by noting that we can solve a linear program in polynomial time (see for example Papadimitriou and Steiglitz 1982). Let us now show that $S$ is indeed a vertex cover. Suppose not; then there exists an edge $e = (u,v)$ which was not covered by any vertex in $S$. That implies that $x(u) < 1/2$ and $x(v) < 1/2$. But, since we know that $x^*$ is a feasible solution to the relaxed linear program, it has to satisfy the constraint $x(u)+x(v) \geq 1$, which leads to a contradiction if we assume that both $x(u) < 1/2$ and $x(v) < 1/2$.

Next, we show that our procedure is a 2-approximation algorithm for the minimum weight vertex cover problem. Let us denote the value of the solution obtained with the LP-Rounding-MVC procedure by $m_{LPR-MWVC}$, the optimal value of the minimum weight vertex cover by $m_{MWVC}$, and the optimal value of the LP-relaxation of the formulation of the minimum weight vertex cover as an integer program by $m_{LP-MWVC}$:

$$\begin{aligned} m_{LPR-MWVC} &= \sum_{v\in S} w(v) \\ &\leq \sum_{v\in S} 2x^*(v)w(v) \leq \sum_{v\in V} 2x^*(v)w(v) \\ &= 2m_{LP-MWVC}. \end{aligned}$$

Since $m_{LP-MWVC} \leq m_{MWVC}$, it follows that

$$m_{LPR-MWVC} \leq 2m_{MWVC}. \qquad \square$$

### 21.2.6 Primal–Dual Method

Some of the most fundamental exact polynomial time algorithms exploit min–max relations that characterize the structure of several combinatorial problems. Most of such min–max relations are special cases of the duality theorem in LP. A great deal of the theory of approximation algorithms also exploits such min–max relations and is based on LP and the LP-duality theory. In this section we start by briefly reviewing some key concepts of the LP-duality. We then describe the primal–dual method and apply it to the minimum weight vertex cover problem.

#### 21.2.6.1 The LP Duality Theorem

Let us illustrate the importance of LP duality through an example:

$$\begin{array}{lll} \text{Minimize} & 15x_1 + 7x_2 + 5x_3 & \\ \text{subject to :} & 3x_1 + x_2 - x_3 & \geq 2 \\ & x_1 + x_2 + x_3 & \geq 1 \\ & x_1, x_2, x_3 & \geq 0. \end{array}$$

This problem is the standard form of an LP minimization problem, with all the constraints of the form "$\geq$", and all the variables constrained to be non-negative. Any minimization LP problem can be written in this format.

Instead of solving this linear program, we can try to provide bounds on its optimal solution, $z^*$. Let us consider a question of the form: "is $z^*$ at most 15?"

A feasible solution to our problem, with value at most 15, is a Yes certificate to that question. For example, the solution $x_1 = 1$, $x_2 = 1$, and $x_3 = 1$ is such a certificate since it satisfies all the constraints of the problem and the objective function value associated with that solution is $12 < 15$. In other words, any feasible solution of our problem provides an upper bound on $z^*$. On the other hand, if we are interested in lower bounds for $z^*$, a good estimate can be based on the bounds associated with the constraints. For example, if we consider the first constraint, given that all its coefficients are smaller than the coefficients of the objective function (we are dealing with a minimization problem), and that all the $x_i$ are non-negative, we can infer that

$$15x_1 + 7x_2 + 5x_3 = 3x_1 + x_2 - x_3 = 2.$$

In other words, the objective function value is at least 2. We can improve this bound by multiplying the first equation by 4 and the second equation by 2, obtaining a lower bound of 10, i.e.

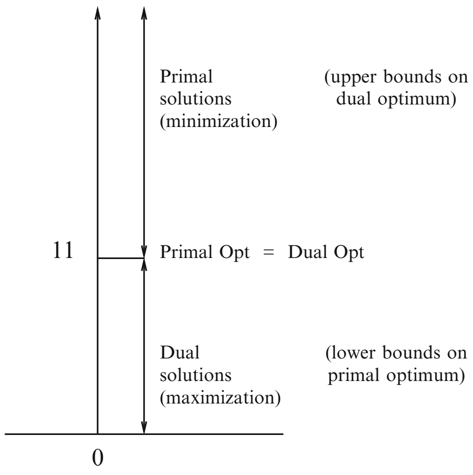
$$15x_1 + 7x_2 + 5x_3 = (12x_1 + 4x_2 - 4x_3) + (2x_1 + 2x_2 + 2x_3) = 10.$$

Basically, we are considering linear combinations of the constraints, multiplying the first constraint by some multiplier $y_1$ and the second by $y_2$. Because we have a minimization problem our goal is to have the coefficient of each $x_i$ in the linear combination of the constraints as close to the corresponding coefficient of $x_i$ in the objective function as possible, but not greater than it. Furthermore, the multipliers have to be non-negative to make sure that the direction of the inequality is not reversed. And of course, we would like our lower bound, given by the sum of the right-hand sides of the constraints multiplied by the corresponding multiplier, to be as large as possible.

This problem of finding the largest lower bound for our minimization problem can be formulated also as a linear program:

$$\begin{array}{rr} \text{Maximize} & 2y_1 + y_2 \\ \text{subject to:} & 3y_1 + y_2 \leq 15 \\ & y_1 + y_2 \leq 7 \\ & -y_1 + y_2 \leq 5 \\ & y_1, y_2, y_3 \geq 0. \end{array}$$

This second linear program is called the *dual* of the original one. The original linear program is called the *primal*. Note that $y_1$ can be interpreted as the multiplier of the first constraint of our primal program, the original minimization problem, and $y_2$ as the multiplier of the second constraint. In this framework, the first constraint of the dual program states that the linear combination of the constraint coefficients



Primal solutions (minimization)
(upper bounds on dual optimum)
11
Primal Opt = Dual Opt
Dual solutions (maximization)
(lower bounds on primal optimum)
0

**Fig. 21.2** Primal–dual relationships

of $x_1$ in the primal program (3 and 1) cannot be greater than $x_1$'s objective function coefficient (15). The objective function of this dual program states that we want to maximize our primal lower bound, i.e. the right-hand sides of our primal program, multiplied by the corresponding multipliers.

Every minimization LP problem in the standard form has a dual LP maximization problem. More interestingly, as we observed before, every feasible solution of the dual provides a lower bound on the optimum of the primal. The dual relationship also holds: every primal feasible solution is an upper bound on the optimum of the dual. In fact, if one has an optimal solution so does the other and the two optimal values coincide. In our example the optimal value is 11, which corresponds to the primal solution $x_1 = 0.5$, $x_2 = 0.5$, and $x_3 = 0$ and the dual solution $y_1 = 4$ and $y_2 = 3$ (see Fig. 21.2). The dual of a maximization LP problem is a minimization LP problem. The dual of the dual of a given problem is the problem itself.

What we have just observed in the example corresponds to a key theorem in linear programming, the duality theorem. In a more formal way, let us consider the pair of primal and dual linear programs:

*Primal linear program*

$$\begin{aligned} &\text{Minimize} \quad \textstyle\sum_{j=1}^{n} c_j x_j \\ &\text{subject to}: \textstyle\sum_{j=1}^{n} a_{ij} x_j \geq b_i, i = 1,\ldots,m \\ &\qquad\qquad\qquad x_j \geq 0, j = 1,\ldots,n \end{aligned}$$

*Dual linear program*

$$\begin{aligned} &\text{Minimize} \quad \textstyle\sum_{i=1}^{m} b_i y_i \\ &\text{subject to}: \textstyle\sum_{i=1}^{m} a_{ij} y_i \geq c_j, j = 1,\ldots,n \\ &\qquad\qquad\qquad y_i \geq 0, i = 1,\ldots,m \end{aligned}$$

where $a_{ij}$, $c_j$, and $b_i$ are given rational numbers. We can now state the duality theorem in a formal way.

**Theorem 21.13. Duality Theorem** *The primal program has an optimal solution if and only if its dual has an optimal solution. Furthermore, if* $x^* = (x_1^*, \dots, x_n^*)$ *and* $y = (y_1^*, \dots, y_m^*)$ *are the optimal solutions of the primal and the dual program, then*

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} a_{ij} y_i^*.$$

Note that given the duality relationships we could have the primal problem as a maximization problem and the dual problem as a minimization problem. The duality theorem provides a succinct way of proving optimality: an optimal solution of the dual problem provides a certificate of optimality for an optimal solution of the primal, and vice versa. As in our example, every feasible solution of the dual provides a lower bound on the optimal value of the primal and of course on the objective function value of any feasible primal solution. This corresponds to half of the duality theorem, referred to as the weak duality theorem. In the design of approximation algorithms, in general, the weak duality theorem is sufficient.

**Theorem 21.14. Weak Duality Theorem** *If* $x = (x_1, \dots, x_n)$ *and* $y = (y_1, \dots, y_m)$ *are the feasible solutions of the primal and the dual program, respectively, then*

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{i=1}^{m} b_i y_i.$$

*Proof.* Given the non-negativity of the $x_j$ and given that $y$ is feasible:

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{j=1}^{n} \left( \sum_{i=1}^{m} a_{ij} y_i \right) x_j$$
$$\sum_{j=1}^{n} \left( \sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{i=1}^{m} \left( \sum_{j=1}^{n} a_{ij} x_j \right) y_i$$
$$\sum_{i=1}^{m} \left( \sum_{j=1}^{n} a_{ij} x_j \right) y_i \geq \sum_{i=1}^{m} b_i y_i.$$

Therefore,

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{i=1}^{m} b_i y_i. \qquad \square$$

From the duality theorem we know that $x$ and $y$ are optimal solutions if and only if the objective function value of the primal and the dual are equal. We can break this condition down into important conditions, known as the complementary slackness conditions.

**Theorem 21.15.** Complementary slackness conditions *Let x and y be primal and dual feasible solutions. Necessary and sufficient conditions for the optimality of x and y are:*

- Primal complementary slackness conditions

$$\sum_{i=1}^{m} a_{ij} y_i = c_j \quad \textit{or} \quad x_j = 0 \quad \textit{(or both) for each} \quad j = 1, \ldots, n.$$

- Dual complementary slackness conditions

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad \textit{or} \quad y_i = 0 \quad \textit{(or both) for each} \quad i = 1, \ldots, m.$$

*Proof.* From the definitions of the primal and the dual:

$$c_j x_j \geq \left( \sum_{i=1}^{m} a_{ij} y_i \right) x_j (j = 1, \ldots, n),$$

$$\left( \sum_{j=1}^{n} a_{ij} x_j \right) y_i \geq b_i y_i (i = 1, \ldots, m).$$

Therefore,

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{j=1}^{n} \left( \sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{i=1}^{m} \left( \sum_{j=1}^{n} a_{ij} x_j \right) y_i \geq \sum_{i=1}^{m} b_i y_i.$$

For this equation to hold throughout, the two equations above have to hold in equality. For the first equation to hold in equality we have to have $x_j = 0$ or $c_j = \sum_{i=1}^{m} a_{ij} y_i$. Similarly, for the second equation to hold in equality we need $y_i = 0$ or $\sum_{j=1}^{n} a_{ij} x_j = b_i$. □

The complementary slackness conditions play a very important role in the design of both exact and approximation algorithms. Below we discuss how we can use them to design an approximation algorithm based on the primal–dual method for the minimum weighted vertex cover.

### *21.2.7 Primal–Dual Method Applied to Minimum Weight Vertex Cover*

Our approximation algorithm for the Minimum Weight Vertex Cover based on LP rounding requires that we solve the LP relaxation of its integer program formulation. If we are dealing with large graphs with many edges, this procedure is relatively expensive, since the number of constraints of our linear program corresponds to the

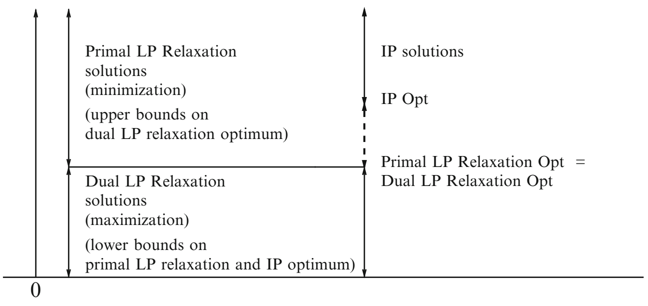


**Fig. 21.3** Feasible solutions of integer program and its primal and dual relaxations

number of edges in the graph. An alternative to this procedure is the primal–dual method, also based on linear programming.

The primal–dual method was originally proposed by Dantzig et al. (1956), inspired by work on a min–max relation for the assignment problem that led to the primal–dual *Hungarian Method* for solving the assignment problem (Egervary 1931; Kuhn 1955). Although the primal–dual method as proposed originally is no longer used to solve LP problems, it has found several applications to develop algorithms for combinatorial optimization problems. In fact, several fundamental algorithms in combinatorial optimization are based on the primal–dual method or can be understood in terms of it. Examples include, in addition to the Hungarian algorithm, Dijkstra's shortest-path algorithm, and Ford and Fulkerson's network flow algorithm. The primal–dual method has also been used to obtain approximation algorithms for several NP-hard optimization problems—see Goemans and Williamson (1997) for a survey.

The primal–dual method provides, in fact, a general framework to devise approximation algorithms for several NP-hard optimization problems. Most primal–dual approximation algorithms enforce one of the complementary slackness conditions relaxing the other. The method starts with the LP relaxation of the primal program and iteratively builds an integral solution to the primal and a feasible solution to the dual program. The primal and the dual programs are used to guide this procedure. At each iteration, the algorithm improves the feasibility of the primal solution and the optimality of the dual solution, in such a way that at the end, the final primal solution is integral and feasible.

As mentioned in the previous section, any feasible solution to the dual provides a lower bound to the optimal solution of the primal. The performance guarantee is obtained by comparing the two solutions (see Fig. 21.3). For many problems, the performance guarantee of primal–dual methods is similar to the one obtained with LP rounding, close to the *integrality gap* of the relaxation. The runtimes of primal–dual methods are in general much faster than solving the LP relaxation since such methods are more versatile and exploit the combinatorial structure of the problem. In fact, for several problems, once we formulate them as linear programs and have in place the duality framework, a simple combinatorial algorithm can be used without further need of the linear programming tools.

Let us consider again the LP relaxation of the integer program formulation of the minimum-weight vertex cover problem.

*LP relaxation of minimum-weight vertex cover IP problem*

$$\begin{array}{llll} \text{Minimize} & \sum_{v\in V} w(v)x(v) & & \\ \text{subject to:} & x(u)+x(v) & \geq 1 & \text{for each} \quad v \in V \\ & x(v) & \geq 0 & \text{for each} \quad v \in V. \end{array}$$

We obtain the dual of the LP relaxation of the IP by associating a multiplier to each of its constraints. Note that each constraint corresponds to an edge in the original graph. Therefore, to each edge $e \in E$ we associate a dual variable denoted by $y(e)$. The constraints of the dual problem state that, for each node $v \in V$, the sum of the dual variables associated with the edges incident to it have to be less than the weight of the node $v, w(v)$. The objective function of the dual is to maximize the lower bound, which corresponds to the sum of the multipliers $y(e), e \in E$.

*Dual LP relaxation of minimum-weight vertex cover IP problem*

$$\begin{array}{llll} \text{Maximize} & \sum_{e\in E} y(e) & & \\ \text{subject to:} & \sum_{u:e=(u,v)\in E} y(e) & \leq w(v) & \text{for each} \quad v \in V \\ & y(e) & \geq 0 & \text{for each} \quad e \in E. \end{array}$$

The weak-duality theorem states that

$$\text{Dual-LP-cost} = \text{Primal-LP-cost}.$$

Therefore,

$$\text{Dual-LP-cost} \leq \text{Dual-LP-cost}^* \leq \text{Primal-LP-cost}^*.$$

Since Primal-LP-cost$^*$ = OPT-IP, it is not necessary to find the optimal solution to the dual problem. We just need to find a feasible solution that will allow us to upper-bound the cost of the vertex cover as a function of the dual feasible solution cost.

The primal–dual weight vertex cover algorithm (PD-MVC) starts from a feasible dual solution in which all the $y(e)$ are set to zero and an infeasible primal solution corresponding to the empty set. It then improves the dual solution while moving towards a feasible primal solution using the complementary slackness conditions.

PD-MVC: Initially, we have an empty set $S$ and set all the dual variables $y(e)$ to 0. Choose an edge $e = (u, v)$ not covered by $S$. Increase the value of the dual variable $y(e)$ until a constraint of the Dual-LP relaxation of the IP becomes tight (i.e. satisfied in equality). The vertex for which the constraint becomes tight, say $v$ is then added to the cover $S$. All the edges incident to the vertex $v$ are removed from the graph. Repeat until no edges remain in the graph. Return $S$ as the minimum-weight vertex.

**Theorem 21.16.** *PD-MVC is a 2-approximation algorithm for Minimum Weight Vertex Cover.*

*Proof.* By construction, $S$ is a feasible solution. We now show that indeed this procedure provides a solution at most twice the value of the optimal solution to the Minimum Weight Vertex Cover problem.

We start by observing that

$$w(v) = \sum_{u:e=(u,v)\in E} y(e) \quad \text{for each} \quad v \in S. \tag{21.1}$$

Let us denote by mPD-MVC the value obtained by the primal–dual method and, as before, by mMWVC the optimal solution value of the Minimum Weight Vertex Cover problem:

$$m_{PD-MVC} = \sum_{v\in S} w(v) = \sum_{v\in S} \sum_{u:e=(u,v)\in E} y(e) \leq \sum_{v\in V} \sum_{u:e=(u,v)\in E} y(e) = 2\sum_{e\in E} y(e). \tag{21.2}$$

Given that

$$\sum_{e\in E} y(e) \leq m_{MWVC} \tag{21.3}$$

it follows that

$$m_{PD-MVC} = 2m_{MWVC}. \tag{21.4}$$

□

## 21.2.8 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 surprising discoveries in the area of algorithm design is that the addition of randomness into the 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). More recently, a deterministic algorithm for primality has been discovered (Agrawal et al. 2004).

#### 21.2.8.1 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 $< m/2$ edges crossing this cut, repeat.

**Theorem 21.17.** *Random-Cut is a 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 $i = 1,\ldots,m$, $X_i$ will be an indicator variable that is 1 if the $i$th 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 $X = m/2$ when a random cut is chosen. In the worst case, when $X = m/2$ 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 $m/2$. □

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.

#### 21.2.8.2 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 formulas 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 $1-1/2^k$. 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.

## 21.3 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.

### *21.3.1 PTAS and FPTAS*

#### 21.3.1.1 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 $\Pi$ that have an approximation scheme. That is, given $\varepsilon > 0$, there exists a polynomial time algorithm $A_\varepsilon$ such that:

- If $\Pi$ is a maximization problem, $A_\varepsilon$ is a $(1+\varepsilon)$ approximation, i.e. the ratio approaches 1 from the right.

- If $\Pi$ is a minimization problem, it is a $(1-\varepsilon)$ approximation (the ratio approaches 1 from the left).

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

#### 21.3.1.2 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 21.18 (Aho et al. 1979).** *Minimum Bin Packing is not in PTAS, unless P = NP. In fact, there is no $3/2-\varepsilon$ approximation for any $\varepsilon > 0$, 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 $S = \{r_1, \dots, r_n\}$, where $r_i \in \mathbb{N}$ for all $i = 1, \dots, n$.
*Solution:* A partition of $S$ into sets $S_1$ and $S_2$; i.e. $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$.
*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 $S = r_1, \dots, r_n$ be a Set Partition instance. Reduce it to Minimum Bin Packing by setting $C = 1/2 \sum_{j=1} s_j$ (half the total sum of elements in $S$), and considering a bin packing instance of items $S' = \{r_1/C, \dots, r_n/C\}$. 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 $(3/2-\varepsilon)$-approximation algorithm $A$, it could be used to solve Set Partition:

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

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

$$3-2 < 4.5-3\varepsilon.$$

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 require the introduction of *probabilistically checkable proofs*, which we will be discussed later.

### *21.3.2 APX*

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

#### 21.3.2.1 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 21.19 (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 can 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 TSP:

*Minimum Traveling Salesman*
*Instance:* A set $C = \{c_1, \dots, c_n\}$ of cities, and a distance function $d : C \times C \to \mathbb{N}$.
*Solution:* A path through the cities, i.e. a permutation $\pi : \{1, \dots, n\} \to \{1, \dots, n\}$.
*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 Salesman 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.

### *21.3.3 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 check efficiently. As far as NP is concerned, all of the checking we do 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 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.) 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 $\mathrm{O}(\log n)$ coins and only checks a constant $(\mathrm{O}(1))$ number of bits of any given proof! The result involves the construction of highly intricate error-correcting codes. We do not discuss this on a formal level here, but shall cite the above in the notation of a theorem.

**Theorem 21.20 (Arora et al. 1998).** *$PCP[\mathrm{O}(\log n), \mathrm{O}(1)] = NP$.*

One corollary of this theorem is that a large class of approximation problems do not admit a PTAS, in particular, as follows.

**Theorem 21.21.** *For $\Pi \in$ {MAX-Ek-SAT, MAX-CUT, Minimum Vertex Cover}, there exists a c such that $\Pi$ cannot be c-approximated in polynomial time, unless $P = NP$.*

## 21.4 Promising Areas for Future Application

### *21.4.1 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.

### *21.4.2 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 LP relaxations and randomized rounding techniques, as well as on results that uncovered the extreme variance or *unpredictability* in the runï¿½time 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 quasi-group (or Latin square) completion problem. Each instance consists of an $n$ by $n$ matrix with $n^2$ cells. A complete quasi-group 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 quasi-group 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 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 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.

### *21.4.3 Average-Case Complexity and Approximation*

Recently, an intriguing thread of theoretical research has explored the connections between the average-case complexity of problems and their approximation hardness (Feige 2002). For example, it is shown that if *random 3SAT* 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 FPTAS 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 FPTAS for Maximum Integer Knapsack.)

## 21.5 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? Is there a relaxation of the problem (where integer solutions are relaxed, and real solutions are allowed) that can be solved efficiently? For many computational problems, there are linear programming relaxations which can be used to approximately solve the original problem.
3. Look for a problem (call it $\Pi$) that is related to yours, and is known to have good approximation algorithms. Try to use the algorithms and techniques for solving $\Pi$ to obtain an approximation algorithm for your problem.
4. Try to prove that your problem cannot be well-approximated, 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* (respectively, *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.

## 21.6 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)

Books on linear programming and duality:

- 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 2004)
- The Design of Approximation Algorithms (Williamson and Shmoys 2001), available at http://www.designofapproxalgs.com/

One of the most intriguing lines of work in approximation algorithms over the past few years has been the formulation and development of the Unique Games Conjecture of Subhash Khot (2002). The conjecture asserts that a certain combinatorial problem is hard to approximate. If the conjecture is true, then many simple approximation algorithms (like the 2-approximation for Vertex Cover) are optimal. However, the status of the conjecture is unclear. Recently, there is some interesting evidence (in the form of subexponential-time approximation algorithms) that the Unique Games Conjecture may be false (Arora et al. 2010).

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 1999)

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, John 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 1998 (David Williamson) http://www.almaden.ibm.com/cs/people/dpw/
- Approximation Algorithms, Tel Aviv University, Fall 2001 (Uri Zwick) http://www.cs.tau.ac.il/
- Approximation Algorithms for Network Problems, Lecture Notes (J.Cheriyan and R.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 .ows, linear programming, and approximation algorithms (Michel Goemans) http://www-math.mit.edu/goemans/
- Lecture notes, www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/ (hosted by CMU)

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, Saxena N (2004) PRIMES is in P. Ann Math 160:781–793

Aho AV, Hopcroft JE, Ullman JD (1979) Computers and intractability: a guide to NP-completeness. Freeman, San Francisco

Aho AV, Hopcroft JE, Ullman JD (1983) Data structures and algorithms. Computer science and information processing series. Addison-Wesley, Reading

Alon N, 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, Szegedy M (1998) Proof verification and the hardness of approximation problems. J ACM 45:501–555

Arora S, Barak B, Steurer D (2010) Subexponential algorithms for unique games and related problems. In: Proceedings of the IEEE symposium on foundations of computer science, Las Vegas, pp 563–572

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

Beier R, Vocking B (2003) Random knapsack in expected polynomial time. J Comput Syst Sci 69:306–329

Chazelle B (2001) The discrepancy method. Cambridge University Press, Cambridge/New York

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. Discret Appl Math 8:25–30

Cook W, Cunningham W, Pulleyblank W, Schrijver A (1988) Combinatorial optimization. Wiley, New York

Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms. MIT, Cambridge

Dantzig G (1998) Linear programming and extensions. Princeton University Press, Princeton

Dantzig GB, Ford LR, Fulkerson DR (1956) A primal-dual algorithm for linear programs. In: Kuhn HW, Tucker AW (eds) Linear inequalities and related systems, Annals of Mathematics Study No. 38, Princeton University Press, Princeton, New Jersey, pp 171–181

Egerváry E (1931) Matrixok kombinatorius tujajdonsagairol. Matematikai es Fizikai Lapok 38:16–28

Feige U (2002) Relations between average case complexity and approximation complexity. In: Proceedings of the ACM symposium on theory of computing, Montreal

Feller W (1971) An introduction to probability theory and its applications. Wiley, New York

Garey MR, Graham RL, Ulman JD (1972) Worst case analysis of memory allocation algorithms. In: Proceedings of the 4th ACM symposium on theory of computing, Denver, pp 143–150

Goemans MX, Williamson DP (1995) Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. J ACM 42:1115–1145

Goemans M, Williamson DP (1997) The primal-dual method for approximation algorithms and its application to network design problems. In: Hochbaum DS (ed) Approximation algorithms for NP-hard problems. PWS, Boston

Goldwasser S, Kilian J (1986) Almost all primes can be quickly certified. In: Proceedings of the annual IEEE symposium on foundations of computer science, Toronto, pp 316–329

Gomes CP, Shmoys D (2002) The promise of LP to boost CSP techniques for combinatorial problems. In: Jussien N, Laburthe F (eds) Proceedings of the CP-AI-OR 2002, Le Croisic, France, pp 291–305. http://www.emn.fr/z-info/cpaior/

Gomes CP, Selman B, Kautz H (1998) Boosting combinatorial search through randomization. In: Proceedings of the AAAI 1998, Madison, Wisconsin, pp 431–437. http://www.aaai.org/Conferences/AAAI/aaai98.php

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

Graham RL (1966) Bounds for certain multiprocessing anomalies. Bell Syst Tech J 45:1563–1581

Hochbaum DS (1982) Approximation algorithms for the set covering and vertex cover problem. SIAM J Comput 11:555–556

Hochbaum DS (1983) Efficient bounds for the stable set, vertex cover and the set packing problems. Discret Appl Math 6:243–254

Hochbaum DS (ed) (1997) Approximation algorithms for NP-hard problems. PWS, Boston

Johnson DS (1974) Approximation algorithms for combinatorial problems. J Comput Syst Sci 9:256–278

Khot S (2002) On the power of unique 2-prover 1-round games. In: Proceedings of the 34th annual ACM symposium on theory of computing, Montreal, pp 767–775

Khot S, Regev O (2003) Vertex cover might be hard to approximate within $2 - \varepsilon$. In: Proceedings of the IEEE conference on computational complexity, Aarhus. J Comput Syst Sci 74:335–349

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

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

Kuhn HW (1955) The Hungarian method for the assignment problem. Nav Res Q 2:83–97

Kumar SR, Russell A, Sundaram R (1999) Approximating latin square extensions. Algorithmica 24:128–138

Laywine C, Mullen G (1998) Discrete mathematics using latin squares. Discrete mathematics and optimization series. Wiley-Interscience, New York

Lovasz L (1975) On the ratio of optimal integral and fractional covers. Discret Math 13:383–390

Motwani R, Raghavan P (1995) Randomized algorithms. Cambridge University Press, Cambridge/New York

Nemhauser G, Wolsey L (1988) Integer and combinatorial optimization. Wiley, New York

Papadimitriou C, Steiglitz K (1982) Combinatorial optimization: algorithms and complexity. Prentice-Hall, Englewood Cliffs

Rabin M (1980) Probabilistic algorithm for testing primality. J Number Theor 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: Cook W, Lovasz L, Seymour P (eds) Combinatorial optimization. DIMACS series. AMS, Providence, pp 355–396

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

Srinivasan A (1999) Approximation algorithms via randomized rounding: a survey. In: Karonskin M, Promel HJ (eds) Lectures on approximation and randomized algorithms. Series in advanced topics in mathematics. Polish Scientific Publishers PWN, Warsaw, pp 9–71

Vazirani V (2004) Approximation algorithms. Springer, Berlin

Williams R, Gomes CP, Selman B (2003) Backdoors to typical case complexity. In: Proceedings of the IJCAI, Acapulco, pp 173–1178

Williamson DP, Shmoys DB (2011) The design of approximation algorithms. Cambridge University Press, New York

Wolsey LA (1980) Heuristic analysis, linear programming and branch and bound. Math Programm 28:271–287