Approximation algorithms for NP-hard optimization problems

Philip N. Klein
Department of Computer Science
Brown University

Neal E. Young
Department of Computer Science
Dartmouth College

Chapter 34, Algorithms and Theory of Computation Handbook
©2010 Chapman & Hall/CRC

1 Introduction

In this chapter, we discuss approximation algorithms for optimization problems. An optimization problem consists in finding the best (cheapest, heaviest, etc.) element in a large set $\mathcal{P}$, called the feasible region and usually specified implicitly, where the quality of elements of the set are evaluated using a function $f(x)$, the objective function, usually something fairly simple. The element that minimizes (or maximizes) this function is said to be an optimal solution of the objective function at this element is the optimal value.

$$\text{optimal value} = \min\{f(x) \mid x \in \mathcal{P}\} \tag{1}$$

A example of an optimization problem familiar to computer scientists is that of finding a minimum-cost spanning tree of a graph with edge costs. For this problem, the feasible region $\mathcal{P}$, the set over which we optimize, consists of *spanning trees*; recall that a spanning tree is a set of edges that connect all the vertices but forms no cycles. The value $f(T)$ of the objective function applied to a spanning tree T is the sum of the costs of the edges in the spanning tree.

The minimum-cost spanning tree problem is familiar to computer scientists because there are several good algorithms for solving it — procedures that, for a given graph, quickly determine the minimum-cost spanning tree. No matter what graph is provided as input, the time required for each of these algorithms is guaranteed to be no more than a slowly growing function of the number of vertices n and edges m (e.g. $O(m \log n)$).

For most optimization problems, in contrast to the minimum-cost spanning tree problem, there is no known algorithm that solves all instances quickly in this sense. Furthermore, there is not likely to be such an algorithm ever discovered, for many of these problems are NP-hard, and such an algorithm would imply that every problem in NP could be solved quickly (i.e. P=NP), which is considered unlikely.[1] One option in such a case is to seek an approximation algorithm — an algorithm that is guaranteed to run quickly (in time polynomial in the input size) and to produce a solution for which the value of the objective function is quantifiably close to the optimal value.

Considerable progress has been made towards understanding which combinatorial-optimization problems can be approximately solved, and to what accuracy. The theory of NP-completeness

[1]For those unfamiliar with the theory of NP-completeness, see Chapters 33 and 34 or (**?**).

can provide evidence not only that a problem is hard to solve precisely but also that it is hard to approximate to within a particular accuracy. Furthermore, for many natural NP-hard optimization problems, approximation algorithms have been developed whose accuracy nearly matches the best achievable according to the theory of NP-completeness. Thus optimization problems can be categorized according to the best accuracy achievable by a polynomial-time approximation algorithm for each problem.

This chapter, which focuses on discrete (rather than continuous) NP-hard optimization problems, is organized according to these categories; for each category, we describe a representative problem, an algorithm for the problem, and the analysis of the algorithm. Along the way we demonstrate some of the ideas and methods common to many approximation algorithms. Also, to illustrate the diversity of the problems that have been studied, we briefly mention a few additional problems as we go. We provide a sampling, rather than a compendium, of the field — many important results, and even areas, are not presented. In Section 12, we mention some of the areas that we do not cover, and we direct the interested reader to more comprehensive and technically detailed sources, such as the excellent recent book (**?**). Because of limits on space for references, we do not cite the original sources for algorithms covered in (**?**).

2 Underlying principles

Our focus is on *combinatorial* optimization problems, problems where the feasible region $\mathcal{P}$ is finite (though typically huge). Furthermore, we focus primarily on optimization problems that are NP-hard. As our main organizing principle, we restrict our attention to algorithms that are provably good in the following sense: for *any* input, the algorithm runs in time polynomial in the length of the input and returns a solution (i.e., a member of the feasible region) whose value (i.e., objective function value) is guaranteed to be near-optimal in some well-defined sense.[2] Such a guarantee is called the performance guarantee. Performance guarantees may be absolute, meaning that the additive difference between the optimal value and the value found by the algorithm is bounded. More commonly, performance guarantees are relative, meaning that the value found by the algorithm is within a multiplicative factor of the optimal value.

When an algorithm with a performance guarantee returns a solution, it has implicitly discovered a bound on the exact optimal value for the problem. Obtaining such bounds is perhaps the most basic challenge in designing approximation algorithms. If one can't compute the optimal value, how can one expect to prove that the output of an algorithm is near it? Three common techniques are what which we shall call witnesses, relaxation, and coarsening.

Intuitively, a *witness* encodes a short, easily verified proof that the optimal value is at least, or at most, a certain value. Witnesses provide a dual role to feasible solutions to a problem. For example, for a maximization problem, where any feasible solution provides a *lower* bound to the optimal value, a witness would provide an *upper* bound on the optimal value. Typically, an approximation algorithm will produce not only a feasible solution, but also a witness. The performance guarantee is typically proven with respect to the two bounds — the upper bound provided by the witness and the lower bound provided by the feasible solution. Since the optimal

[2]An alternative to this *worst-case* analysis is *average-case* analysis. See Chapter 2.

value is between the two bounds, the performance guarantee also holds with respect to the optimal value.

Relaxation is another way to obtain a lower bound on the minimum value (or an upper bound in the case of a maximization problem). One formulates a new optimization problem, called a relaxation of the original problem, using the same objective function but a larger feasible region $\mathcal{P}'$ that includes $\mathcal{P}$ as a subset. Because $\mathcal{P}'$ contains $\mathcal{P}$, any $x \in \mathcal{P}$ (including the optimal element x) belongs to $\mathcal{P}'$ as well. Hence the optimal value of the relaxation, $\min\{f(x) \mid x \in \mathcal{P}'\}$, is less than or equal to the optimal value of the original optimization problem. The intent is that the optimal value of the relaxation should be easy to calculate and should be reasonably close to the optimal value of the original problem.

Linear programming can provide both witnesses and relaxations, and is therefore an important technique in the design and analysis of approximation algorithms. Randomized rounding is a general approach, based on the probabilistic method, for converting a solution to a relaxed problem into an approximate solution to the original problem.

To *coarsen* a problem instance is to alter it, typically restricting to a less complex feasible region or objective function, so that the result problem can be efficiently solved, typically by dynamic programming. For coarsening to be useful, the coarsened problem must approximate the original problem, in that there is a rough correspondence between feasible solutions of the two problems, a correspondence that approximately preserves cost. We use the term *coarsening* rather loosely to describe a wide variety of algorithms that work in this spirit.

3 Approximation algorithms with small additive error

3.1 Minimum-degree spanning tree

For our first example, consider a slight variant on the minimum-cost spanning tree problem, the *minimum-degree spanning tree problem.* As before, the feasible region $\mathcal{P}$ consists of spanning trees of the input graph, but this time the objective is to find a spanning tree whose *degree* is minimum. The degree of a vertex of a spanning tree (or, indeed, of any graph), is the number of edges incident to that vertex, and the degree of the spanning tree is the maximum of the degrees of its vertices. Thus minimizing the degree of a spanning tree amounts to finding a smallest integer k for which there exists a spanning tree in which each vertex has at most k incident edges.

Any procedure for finding a minimum-degree spanning tree in a graph could be used to find a Hamiltonian path in any graph that has one, for a Hamiltonian path is a degree-two spanning tree. (A Hamiltonian path of a graph is a path through that graph that visits each vertex of the graph exactly once.) Since it is NP-hard even to determine whether a graph has a Hamiltonian path, even determining whether the minimum-degree spanning tree has degree two is presumed to be computationally difficult.

3.2 An approximation algorithm for minimum-degree spanning tree

Nonetheless, the minimum-degree spanning-tree problem has a remarkably good approximation algorithm (**?**, Ch. 7). For an input graph with m edges and n vertices, the algorithm requires time slightly more than the product of m and n. The output is a spanning tree whose degree is

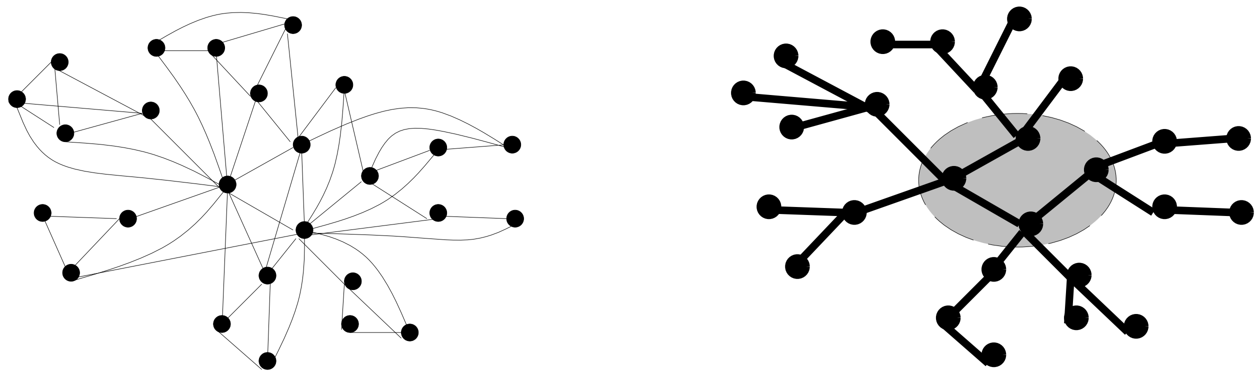

Figure 1: On the left is an example input graph G. On the right is a spanning tree T that might be found by the approximation algorithm. The shaded circle indicates the nodes in the witness set S.

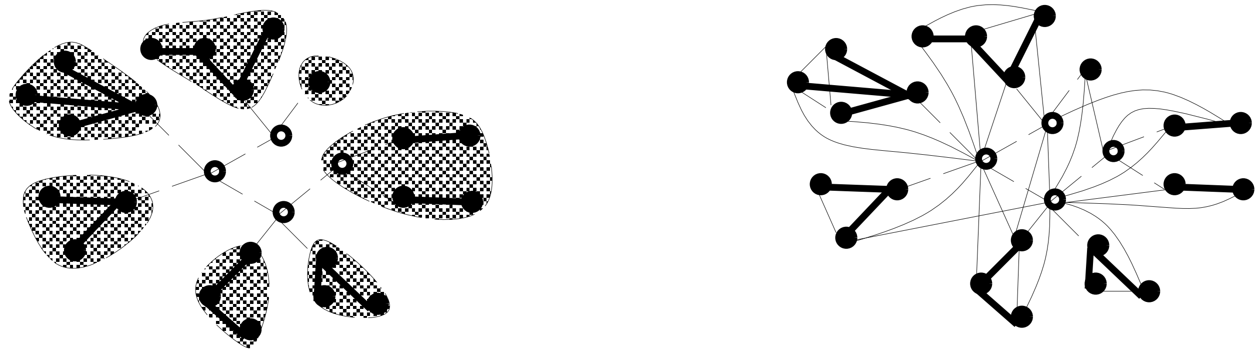

Figure 2: The figure on the left shows the r trees $T_1, \ldots, T_r$ obtained from T by deleting the nodes of S. Each tree is indicated by a shaded region. The figure on the right shows that no edges of the input graph G connect different trees T_i.

guaranteed to be at most *one more* than the minimum degree. For example, if the graph has a Hamiltonian path, the output is either such a path or a spanning tree of degree three.

Given a graph G, the algorithm naturally finds the desired spanning tree T of G. The algorithm also finds a witness — in this case, a set S of vertices proving that T's degree is nearly optimal. Namely, let k denote the degree of T, and let $T_1, T_2, \ldots, T_r$ be the subtrees that would result from T if the vertices of S were deleted. The following two properties are enough to show that T's degree is nearly optimal.

1. There are no edges of the graph G between distinct trees T_i, and

2. the number r of trees T_i is at least $|S|(k-1) - 2(|S|-1)$.

To show that T's degree is nearly optimal, let V_i denote the set of vertices comprising subtree T_i ($i = 1, \ldots, r$). Any spanning tree T^* at all must connect up the sets $V_1, V_2, \ldots, V_r$ and the vertices $y_1, y_2, \ldots, y_{|S|} \in S$, and must use at least $r + |S| - 1$ edges to do so. Furthermore, since no edges go between distinct sets V_i, all these edges must be incident to the vertices of S.

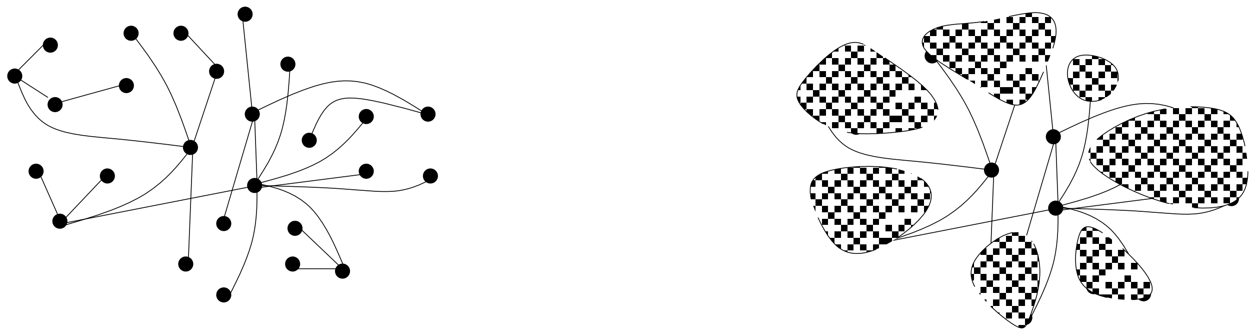
Figure 3: The figure on the left shows an arbitrary spanning tree T^* for the same input graph G. The figure on the right has r shaded regions, one for each subset V_i of nodes corresponding to a tree T_i in Figure 3.2. The proof of the algorithm's performance guarantee is based on the observation that at least $r + |S| - 1$ edges are needed to connect up the V_i's and the nodes in S.

Hence we obtain

$$\begin{aligned}\sum\{\deg_{T^*}(y) \mid y \in S\} &\geq r + |S| - 1 \\ &\geq |S|(k-1) - 2(|S|-1) + |S| - 1 \\ &= |S|(k-1) - (|S|-1) \qquad (2)\end{aligned}$$

where $\deg_{T^*}(y)$ denotes the degree of y in the tree T^*. Thus the average of the degrees of vertices in S is at least $\frac{|S|(k-1)-(|S|-1)}{|S|}$, which is strictly greater than $k-2$. Since the average of the degrees of vertices in S is greater than $k-2$, it follows that at least one vertex has degree at least $k-1$.

We have shown that for every spanning tree T^*, there is at least one vertex with degree at least $k-1$. Hence the minimum degree is at least $k-1$.

We have not explained how the algorithm obtains both the spanning tree T and the set S of vertices, only how the set S shows that the spanning tree is nearly optimal. The basic idea is as follows. Start with any spanning tree T, and let d denote its degree. Let S be the set of vertices having degree d or $d-1$ in the current spanning tree. Let $T_1, \ldots, T_r$ be the subtrees comprising $T-S$. If there are no edges between these subtrees, the set S satisfies property 1 and one can show it also satisfies property 2; in this case the algorithm terminates. If on the other hand there is an edge between two distinct subtrees T_i and T_j, inserting this edge in T and removing another edge from T results in a spanning tree with fewer vertices having degree at least $d-1$. Repeat this process on the new spanning tree; in subsequent iterations the improvement steps are somewhat more complicated but follow the same lines. One can prove that the number of iterations is $O(n \log n)$.

We summarize our brief sketch of the algorithm as follows: either the current set S is a witness to the near-optimality of the current spanning tree T, or there is a slight modification to the set and the spanning tree that improve them. The algorithm terminates after a relatively small number of improvements.

This algorithm is remarkable not only for its simplicity and elegance but also for the quality of the approximation achieved. As we shall see, for most NP-hard optimization problems, we must settle for approximation algorithms that have much weaker guarantees.

3.3 Other problems having small-additive-error algorithms

There are a few other natural combinatorial-optimization problems for which approximation algorithms with similar performance guarantees are known. Here are two examples:

Edge Coloring: Given a graph, color its edges with a minimum number of colors so that, for each vertex, the edges incident to that vertex are all different colors. For this problem, it is easy to find a witness. For any graph G, let v be the vertex of highest degree in G. Clearly one needs to assign at least $\deg_G(v)$ colors to the edges of G, for otherwise there would be two edges with the same color incident to v. For any graph G, there is an edge coloring using a number of colors equal to one plus the degree of G. The proof of this fact translates into a polynomial-time algorithm that approximates the minimum edge-coloring to within an additive error of 1.

Bin Packing: The input consists of a set of positive numbers less than 1. A solution is a partition of the numbers into sets summing to no more than 1. The goal is to minimize the number of blocks of the partition. There are approximation algorithms for bin packing that have very good performance guarantees. For example, the performance guarantee for one such algorithm is as follows: for any input set I of item weights, it finds a packing that uses at most $\mathrm{OPT}(I) + O(\log^2 \mathrm{OPT}(I))$ bins, where $\mathrm{OPT}(I)$ is the number of bins used by the best packing, i.e. the optimal value.

4 Randomized rounding and linear programming

A *linear programming* problem is any optimization problem in which the feasible region corresponds to assignments of values to variables meeting a set of linear inequalities and in which the objective function is a linear function. An instance is determined by specifying the set of variables, the objective function, and the set of inequalities. Linear programs are capable of representing a large variety of problems and have been studied for decades in combinatorial optimization and have a tremendous literature (see e.g., Chapters 24 and 25 of this book). Any linear program can be solved — that is, a point in the feasible region maximizing or minimizing the objective function can be found — in time bounded by a polynomial in the size of the input.

A <u>(mixed) integer linear programming problem</u> is a linear programming problem augmented with additional constraints specifying that (some of) the variables must take on integer values. Such constraints make integer linear programming even more general than linear programming — in general, solving integer linear programs is NP-hard.

For example, consider the following *balanced matching problem*: The input is a bipartite graph $G = (V, W, E)$. The goal is to choose an edge incident to each vertex in V ($|V|$ edges in total), while minimizing the maximum *load* of (number of chosen edges adjacent to) any vertex in W. The vertices in V might represent tasks, the vertices in W might represent people, while the presence of edge $\{v, w\}$ indicates that person w is competent to perform task v. The problem is then to assign each task to a person competent to perform it, while minimizing the maximum number of tasks assigned to any person.[3]

[3]Typically, randomized rounding is applied to NP-hard problems, whereas the balanced matching problem here is actually solvable in polynomial time. We use it as an example for simplicity — the analysis captures the essential

This balanced matching problem can be formulated as the following integer linear program:

$$\text{minimize } \Delta$$
$$\text{subject to } \begin{cases} \sum_{u\in N(v)} x(u,v) & = & 1 & \forall v \in V \\ \sum_{v\in N(u)} x(u,v) & \le & \Delta & \forall w \in W \\ x(u,v) & \in & \{0,1\} & \forall (u,v) \in E. \end{cases}$$

Here $N(x)$ denotes the set of neighbors of vertex x in the graph. For each edge (u, v) the variable $x(u, v)$ determines whether the edge (u, v) is chosen. The variable Δ measures the maximum load.

Relaxing the integrality constraints (i.e., replacing them as well as we can by linear inequalities) yields the linear program:

$$\text{minimize } \Delta$$
$$\text{subject to } \begin{cases} \sum_{u\in N(v)} x(u,v) & = & 1 & \forall v \in V \\ \sum_{v\in N(u)} x(u,v) & \le & \Delta & \forall w \in W \\ x(u,v) & \ge & 0 & \forall (u,v) \in E. \end{cases}$$

Rounding a fractional solution to a true solution. This relaxed problem can be solved in polynomial time simply because it is a linear program. Suppose we have an optimal solution x^*, where each $x^*(e)$ is a fraction between 0 and 1. How can we convert such an optimal <u>fractional</u> solution into an *approximately optimal* integer solution? *Randomized rounding* is a *general* approach for doing just this (**?**, Ch. 5).

Consider the following polynomial-time randomized algorithm to find an integer solution $\hat{x}$ from the optimal solution x^* to the linear program:

1. Solve the linear program to obtain a fractional solution x^* of load Δ^*.
2. For each vertex $v \in V$:
   (a) Choose a single edge incident to v *at random*, so that the probability that a given edge (u, v) is chosen is $x^*(u, v)$. (Note that $\sum_{u\in N(v)} x^*(u,v) = 1$.)
   (b) Let $\hat{x}(u, v) \leftarrow 1$.
   (c) For all other edges (u', v) incident to v, let $\hat{x}(u', v) \leftarrow 0$.

The algorithm will always choose one edge adjacent to each vertex in V. Thus, $\hat{x}$ is a feasible solution to the original integer program. What can we say about the load? For any particular vertex $w \in W$, the load on w is $\sum_{u\in N(v)} \hat{x}(u,v)$. For any particular edge $(u, v) \in E$, the probability that $\hat{x}(u, v) = 1$ is $x^*(u, v)$. Thus the *expected* value of the load on a vertex $u \in U$ is $\sum_{vinN(u)} x^*(u,v)$, which is at most Δ^*. This is a good start. Of course, the *maximum* load over all $u \in U$ is likely to be larger. How much larger?

spirit of a similar analysis for the well-studied integer multicommodity flow problem. (A simple version of that problem is: "Given a network and a set of commodities (each a pair of vertices), choose a path for each commodity minimizing the maximum congestion on any edge.")

To answer this, we need to know more about the distribution of the load on v than just the expected value. The key fact that we need to observe is that the load on any $v \in V$ is a sum of *independent* $\{0,1\}$*-random variables.* This means it is not likely to deviate much from its expected value. Precise estimates come from standard bounds, called "Chernoff"- or "Hoeffding" bounds, such as the following:

Theorem *Let X be the sum of independent $\{0,1\}$ random variables. Let $\mu > 0$ be the expected value of X. Then for any $\epsilon > 0$,*

$$\Pr[X \geq (1+\epsilon)\mu] < \exp(-\mu \min\{\epsilon, \epsilon^2\}/3).$$

(See e.g. (**?**, Ch. 4.1).) This is enough to analyze the performance guarantee of the algorithm. It is slightly complicated, but not too bad:

Claim *With probability at least 1/2, the maximum load induced by $\hat{x}$ exceeds the optimal by at most an additive error of*

$$\max\left\{3\ln(2m), \sqrt{3\ln(2m)\Delta^*}\right\},$$

where $m = |W|$.

proof sketch: As observed previously, for any particular v, the load on v is a sum (of independent random $\{0,1\}$-variables) with expectation bounded by Δ^*. Let ϵ be just large enough so that $\exp(-\Delta^* \min\{\epsilon, \epsilon^2\}/3) = 1/(2m)$. By the Chernoff-type bound above, the probability that the load exceeds $(1+\epsilon)\Delta^*$ is then less than $1/(2m)$. Thus, by the naive union bound[4], the probability that the *maximum* load on any $v \in V$ is more than $\Delta^*(1+\epsilon) = \Delta^* + \epsilon\Delta^*$ is less then 1/2. We leave it to the reader to verify that the choice of ϵ makes $\epsilon\Delta^*$ equal the expression in the statement of the claim. $\diamond$

Summary. This is the general randomized-rounding recipe:

1. Formulate the original NP-hard problem as an integer linear programming problem (IP).
2. Relax the program IP to obtain a linear program (LP).
3. Solve the linear program, obtaining a fractional solution.
4. Randomly round the fractional solution to obtain an approximately optimal integer solution.

5 Performance ratios and ρ-approximation

Relative (multiplicative) performance guarantees are more common than absolute (additive) performance guarantees. One reason is that many NP-hard optimization problems are <u>rescalable</u>: given an instance of the problem, one can construct a new, equivalent instance by scaling the objective function. For instance, the traveling salesman problem is rescalable — given an instance, multiplying the edge weights by any $\lambda > 0$ yields an equivalent problem with the objective function scaled by λ. For rescalable problems, the best one can hope for is a relative performance guarantee (**?**).

[4]The probability that any of several events happens is at most the sum of the probabilities of the individual events.

A ρ-approximation algorithm is an algorithm that returns a feasible solution whose objective function value is at most ρ times the minimum (or, in the case of a maximization problem, the objective function value is at least ρ times the maximum). We say that the performance ratio of the algorithm is ρ.[5]

6 Polynomial approximation schemes

The *knapsack problem* is an example of a rescalable NP-hard problem. An instance consists of a set of pairs of numbers $(\text{weight}_i, \text{profit}_i)$, and the goal is to select a subset of pairs for which the sum of weights is at most 1 so as to maximize the sum of profits. (Which items should one put in a knapsack of capacity 1 so as to maximize profit?)

Since the knapsack problem is rescalable and NP-hard, we assume that there is no approximation algorithm achieving, say, a fixed absolute error. One is therefore led to ask: what is the best performance ratio achievable by a polynomial-time approximation algorithm? In fact (assuming P$\neq$NP), there is no such *best* performance ratio: for any given $\epsilon > 0$, there is a polynomial approximation algorithm whose performance ratio is $1 + \epsilon$. The smaller the value of ϵ, however, the greater the running time of the corresponding approximation algorithm. Such a collection of approximation algorithms, one for each $\epsilon > 0$, is called a (polynomial) approximation scheme.

Think of an approximation scheme as an algorithm that takes an additional parameter, the value of ϵ, in addition to the input specifying the instance of some optimization problem. The running time of this algorithm is bounded in terms of the size of the input and in terms of ϵ. For example, there is an approximation scheme for the knapsack problem that requires time $O(n \log(1/\epsilon) + 1/\epsilon^4)$ for instances with n items. Below we sketch a much simplified version of this algorithm that requires time $O(n^3/\epsilon)$. The algorithm works by coarsening.

The algorithm is given the pairs $(\text{weight}_1, \text{profit}_1), \ldots, (\text{weight}_n, \text{profit}_n)$, and the parameter ϵ. We assume without loss of generality that each weight is less than or equal to 1. Let $\text{profit}_{\max} = \max_i \text{profit}_i$. Let OPT denote the (unknown) optimal value. Since the item of greatest profit itself constitutes a solution, albeit not usually a very good one, we have $\text{profit}_{\max} \leq \text{OPT}$. In order to achieve a relative error of at most ϵ, therefore, it suffices to achieve an absolute error of at most $\epsilon\ \text{profit}_{\max}$.

We transform the given instance into a coarsened instance by rounding each profit down to a multiple of $K = \epsilon\ \text{profit}_{\max}/n$. In so doing, we reduce each profit by less than $\epsilon\ \text{profit}_{\max}/n$. Consequently, since the optimal solution consists of no more than n items, the profit of this optimal solution is reduced by less than $\epsilon\ \text{profit}_{\max}$ in total. Thus, the optimal value for the coarsened instance is at least $\text{OPT} - \epsilon\ \text{profit}_{\max}$, which is in turn at least $(1 - \epsilon)\,\text{OPT}$. The corresponding solution, when measured according to the original profits, has value at least this much. Thus we need only solve the coarsened instance optimally in order to get a performance guarantee of $1 - \epsilon$.

Before addressing the solution of the coarsened instance, note that the optimal value is the sum of at most n profits, each at most $\text{profit}_{\max}$. Thus $\text{OPT} \leq n^2 K/\epsilon$. The optimal value for the coarsened instance is therefore also at most $n^2 K/\epsilon$.

[5]This terminology is the most frequently used, but one also finds alternative terminology in the literature. Confusingly, some authors have used the term $1/\rho$-approximation algorithm or $(1-\rho)$-approximation algorithm to refer to what we call a ρ-approximation algorithm.

To solve the coarsened instance optimally, we use dynamic programming. Note that for the coarsened instance, each achievable total profit can be written as $i \cdot K$ for some integer $i \leq n^2/\epsilon$. The dynamic-programming algorithm constructs an $\lceil n^2/\epsilon \rceil \times (n+1)$ table $T[i,j]$ whose i,j entry is the minimum weight required to achieve profit $i \cdot K$ using a subset of the items 1 through j. The entry is infinity if there is no such way to achieve that profit.

To fill in the table, the algorithm initializes the entries $T[i,0]$ to infinity, then executes the following step for $j = 1, 2, \ldots, n$:

$$\text{For each } i, \text{ set } T[i,j] := \min\{T[i,j-1], \text{weight}_j + T[i - (\widehat{\text{profit}}_j/K), j-1]\}$$

where $\widehat{\text{profit}}_j$ is the profit of item j in the rounded-down instance. A simple induction on j shows that the calculated values are correct. The optimal value for the coarsened instance is

$$\widehat{\text{OPT}} = \max\{iK \mid T[i,n] \leq 1\}.$$

The above calculates the optimal *value* for the coarsened instance; as usual in dynamic programming, a corresponding feasible solution can easily be computed if desired.

6.1 Other problems having polynomial approximation schemes

The running time of the knapsack approximation scheme depends polynomially on $1/\epsilon$. Such a scheme is called a fully polynomial approximation scheme. Most natural NP-complete optimization problems are strongly NP-hard, meaning essentially that the problems are NP-hard even when the numbers appearing in the input are restricted to be no larger in magnitude than the size of the input. For such a problem, we cannot expect a fully polynomial approximation scheme to exist (**?**, §4.2). On the other hand, a variety of NP-hard problems in fixed-dimensional Euclidean space have approximation schemes. For instance, given a set of points in the plane:

Covering with Disks: Find a minimum set of area-1 disks (or squares, etc.) covering all the points (**?**, §9.3.3).

Euclidean Traveling Salesman: Find a closed loop passing through each of the points and having minimum total arc length (**?**).

Euclidean Steiner Tree: Find a minimum-length set of segments connecting up all the points (**?**).

Similarly, many problems in planar graphs or graphs of fixed genus can be have polynomial approximation schemes (**?**, §9.3.3), For instance, **given a planar graph with weights assigned to its vertices:**

Maximum-Weight Independent Set: Find a maximum-weight set of vertices, no two of which are adjacent.

Minimum-Weight Vertex Cover: Find a minimum-weight set of vertices such that every edge is incident to at least one of the vertices in the set.

The above algorithms use relatively more sophisticated and varied coarsening techniques.

7 Constant-factor performance guarantees

We have seen that, assuming P$\neq$NP, rescalable NP-hard problems do not have polynomial-time approximation algorithms with small absolute errors but may have fully polynomial approximation schemes, while strongly NP-hard problems do not have fully polynomial approximation schemes but may have polynomial approximation schemes. Further, there is a class of problems that do not have approximation schemes: for each such problem there is a constant c such that any polynomial-time approximation algorithm for the problem has relative error at least c (assuming P$\neq$ NP). For such a problem, the best one can hope for is an approximation algorithm with constant performance ratio.

Our example of such a problem is the *vertex cover* problem: given a graph G, find a minimum-size set C (a *vertex cover*) of vertices such that every edge in the graph is incident to some vertex in C. Here the feasible region $\mathcal{P}$ consists of the vertex covers in G, while the objective function is the size of the cover. Here is a simple approximation algorithm (**?**):

1. Find a maximal independent set S of edges in G.
2. Let C be the vertices incident to edges in S.

(A set S of edges is *independent* if no two edges in S share an endpoint. The set S is *maximal* if no larger independent set contains S.) The reader may wish to verify that the set S can be found in linear time, and that because S is maximal, C is necessarily a cover.

What performance guarantee can we show? Since the edges in S are independent, *any* cover must have at least one vertex for each edge in S. Thus S is a witness proving that any cover has at least $|S|$ vertices. On the other hand, the cover C has $2|S|$ vertices. Thus the cover returned by the algorithm is at most *twice* the size of the optimal vertex cover.

The weighted vertex cover problem. The *weighted* vertex cover problem is a generalization of the vertex cover problem. An instance is specified by giving a graph $G = (V, E)$ and, for each vertex v in the graph, a number $\mathrm{wt}(v)$ called its *weight.* The goal is to find a vertex cover minimizing the total weight of the vertices in the cover. Here is one way to represent the problem as an integer linear program:

$$\begin{aligned} &\text{minimize } \sum_{v \in V} \mathrm{wt}(v)x(v) \\ &\text{subject to } \left\{ \begin{array}{rcll} x(u)+x(v) & \geq & 1 & \forall \{u,v\} \in E \\ x(v) & \in & \{0,1\} & \forall v \in V. \end{array} \right. \end{aligned}$$

There is one $\{0,1\}$-variable $x(v)$ for each vertex v representing whether v is in the cover or not, and there are constraints for the edges that model the covering requirement. The feasible region of this program corresponds to the set of vertex covers. The objective function corresponds to the total weight of the vertices in the cover. Relaxing the integrality constraints yields

$$\text{minimize} \sum_{v\in V} \text{wt}(v)x(v)$$
$$\text{subject to} \begin{cases} x(u)+x(v) & \geq & 1 & \forall\{u,v\}\in E \\ x(v) & \geq & 0 & \forall v \in V. \end{cases}$$

This relaxed problem is called the *fractional* weighted vertex cover problem; feasible solutions to it are called *fractional* vertex covers.[6]

Rounding a fractional solution to a true solution. By solving this linear program, an optimal fractional cover can be found in polynomial time. For this problem, it is possible to convert a fractional cover into an approximately optimal true cover by rounding the fractional cover in a simple way:

1. Solve the linear program to obtain an optimal fractional cover x^*.
2. Let $C = \left\{v \in V : x^*(v) \geq \frac{1}{2}\right\}$.

The set C is a cover because for any edge, at least one of the endpoints must have fractional weight at least 1/2. The reader can verify that the total weight of vertices in C is at most twice the total weight of the fractional cover x^*. Since the fractional solution was an optimal solution to a relaxation of the original problem, this is a 2-approximation algorithm (**?**).

For most problems, this simple kind of rounding is not sufficient. The previously discussed technique called *randomized rounding* is more generally useful.

Primal-dual algorithms — witnesses via duality. For the purposes of approximation, solving a linear program *exactly* is often unnecessary. One can often design a faster algorithm based on the witness technique, using the fact that *every linear program has a well-defined notion of "witness"*. The witnesses for a linear program P are the feasible solutions to another related linear program called the dual of P.

Suppose our original problem is a minimization problem. Then for each point y in the feasible region of the dual problem, the value of the objective function at y is a lower bound on the value of the optimal value of the original linear program. That is, any feasible solution to the dual problem is a possible witness — both for the original integer linear program and its relaxation. For the weighted vertex cover problem, the dual is the following:

$$\text{maximize} \sum_{e\in E} y(e)$$
$$\text{subject to} \begin{cases} \sum_{e\ni v} y(e) & \leq & \text{wt}(v) & \forall v \in V \\ y(e) & \geq & 0 & \forall e \in E. \end{cases}$$

A feasible solution to this linear program is called an *edge packing*. The constraints for the vertices are called *packing constraints*.

[6]The reader may wonder whether additional constraints of the form $x(v) \leq 1$ are necessary. In fact, assuming the vertex weights are non-negative, there is no incentive to make any $x(v)$ larger than 1, so such constraints would be redundant.

Recall the original approximation algorithm for the unweighted vertex cover problem: find a maximal independent set of edges S; let C be the vertices incident to edges in S. In the analysis, the set S was the witness.

Edge packings generalize independent sets of edges. This observation allows us to generalize the algorithm for the unweighted problem. Say an edge packing is *maximal* if, for every edge, one of the edge's vertices has its packing constraint met. Here is the algorithm:

1. Find a maximal edge packing y.
2. Let C be the vertices whose packing constraints are tight for y.

The reader may wish to verify that a maximal edge packing can easily be found in linear time and that the set C is a cover because y is maximal.

What about the performance guarantee? Since only vertices whose packing constraints are tight are in C, and each edge has only two vertices, we have

$$\sum_{v\in C}\mathrm{wt}(v)=\sum_{v\in C}\sum_{e\ni v}y(e)\le 2\sum_{e\in E}y(e).$$

Since y is a solution to the dual, $\sum_e y(e)$ is a lower bound on the weight of any vertex cover, fractional or otherwise. Thus, the algorithm is a 2-approximation algorithm.

Summary. This is the general primal-dual recipe:

1. Formulate the original NP-hard problem as an integer linear programming problem (IP).
2. Relax the program IP to obtain a linear program (LP).
3. Use the dual (DLP) of LP as a source of witnesses.

Beyond these general guidelines, the algorithm designer is still left with the task of figuring out *how* to find a good solution and witness. See (**?**, Ch. 4) for an approach that works for a wide class of problems.

7.1 Other optimization problems with constant-factor approximations

Constant-factor approximation algorithms are known for problems from many areas. In this section, we describe a sampling of these problems. For each of the problems described here, there is no polynomial approximation scheme (unless P=NP); thus constant-factor approximation algorithms are the best we can hope for. For a typical problem, there will be a simple algorithm achieving a small constant factor while there may be more involved algorithms achieving better factors. The factors known to be achievable typically come close to, but do not meet, the best lower bounds known (assuming P$\neq$NP).

For the problems below, we omit discussion of the techniques used; many of the problems are solved using a relaxation of some form, and (possibly implicitly) the primal-dual recipe. Many of these problems have polynomial approximation schemes if restricted to graphs induced by points in the plane or constant-dimensional Euclidean space (see Section 6.1).

MAX-SAT: Given a propositional formula in conjunctive normal form (an “and” of “or”’s of possibly negated Boolean variables), find a truth assignment to the variables that maximizes the number of clauses (groups of “or”’ed variables in the formula) that are true under the assignment. A variant called MAX-3SAT restricts to the formula to have three variables per clause. MAX-3SAT is a canonical example of a problem in the complexity class MAX-SNP (**?**, §10.3).

MAX-CUT: Given a graph, partition the vertices of the input graph into two sets so as to maximize the number of edges with endpoints in distinct sets. For MAX-CUT and MAX-SAT problems, the best approximation algorithms currently known rely on randomized rounding and a generalization of linear programming called semidefinite programming (**?**, §11.3).

Shortest Superstring: Given a set of strings $\sigma_1, \ldots, \sigma_k$, find a minimum-length string containing all σ_i’s. This problem has applications in computational biology (**?**; **?**).

K**-Cluster:** Given a graph with weighted edges and given a parameter k, partition the vertices into k clusters so as to minimize the maximum distance between any two vertices in the same cluster. For this and related problems see (**?**, §9.4).

Traveling Salesman: Given a complete graph with edge weights satisfying the triangle inequality, find a minimum-length path that visits every vertex of the graph (**?**, Ch. 8).

Edge and Vertex Connectivity: Given a weighted graph $G = (V, E)$ and an integer k, find a minimum-weight edge set $E' \subseteq E$ such that between any pair of vertices, there are k edge-disjoint paths in the graph $G' = (V, E')$. Similar algorithms handle the goal of k *vertex*-disjoint paths and the goal of *augmenting* a given graph to achieve a given connectivity (**?**, Ch. 6)

Steiner Tree: Given an undirected graph with positive edge-weights and a subset of the vertices called *terminals*, find a minimum-weight set of edges through which all the terminals (and possibly other vertices) are connected (**?**, Ch. 8). The Euclidean version of the problem is “Given a set of points in $\mathbb{R}^n$, find a minimum-total-length union of line segments (with arbitrary endpoints) that is connected and contains all the given points.”

Steiner Forest: Given a weighted graph and a collection of *groups* of terminals, find a minimum-weight set of edges through which every pair of terminals within each group are connected (**?**, Ch. 4). The algorithm for this problem is based on a primal-dual framework that has been adapted to a wide variety of network design problems. See Section 8.1.

8 Logarithmic performance guarantees

When a constant-ratio performance guarantee is not possible, a slowly-growing ratio is the next best thing. The canonical example of this is the *set cover* problem: Given a family of sets $\mathcal{F}$ over a universe $\mathcal{U}$, find a minimum-cardinality *set cover* C — a collection of the sets that collectively contain all elements in $\mathcal{U}$. In the weighted version of the problem, each set also has a weight and the goal is to find a set cover of minimum total weight. This problem is important due to its generality. For instance, it generalizes the vertex cover problem.

Here is a simple greedy algorithm:

1. Let $C \leftarrow \emptyset$.

2. Repeat until all elements are covered: add a set S to C maximizing

$$\frac{\text{the number of elements in } S \text{ not in any set in } C}{\text{wt}(S)}.$$

3. Return C.

The algorithm has the following performance guarantee (**?**, §3.2):
Theorem *The greedy algorithm for the weighted set cover problem is an H_s-approximation algorithm, where s is the maximum size of any set in $\mathcal{F}$.*
By definition $H_s = 1 + \frac{1}{2} + \frac{1}{3} + \cdot + \frac{1}{s}$; also, $H_s \le 1 + \ln s$.

We will give a direct argument for the performance guarantee and then relate it to the general primal-dual recipe. Imagine that as the algorithm proceeds, it assigns *charges* to the elements as they are covered. Specifically, when a set S is added to the cover C, if there are k elements in S not previously covered, assign each such elements a charge of $\text{wt}(S)/k$. Note that *the total charge assigned over the course of the algorithm equals the weight of the final cover C.*

Next we argue that *the total charge assigned over the course of the algorithm is a lower bound on H_s times the weight of the optimal vertex cover.* These two facts together prove the theorem.

Suppose we could prove that for any set T in the optimal cover C^*, the elements in T are assigned a total charge of at most $\text{wt}(T)H_s$. Then we would be done, because every element is in at least one set in the optimal cover:

$$\sum_{i \in U} charge(i) \le \sum_{T \in C^*} \sum_{i \in T} charge(i) \le \sum_{T \in C^*} \text{wt}(T) H_s.$$

So, consider, for example, a set $T = \{a, b, c, d, e, f\}$ with $\text{wt}(T) = 3$. For convenience, assume that the greedy algorithm covers elements in T in alphabetical order. What can we say about the charge assigned to a? Consider the iteration when a was first covered and assigned a charge. At the beginning of that iteration, T was not yet chosen and none of the 6 elements in T were yet covered. Since the greedy algorithm had the option of choosing T, whatever set it did choose resulted in a charge to a of at most $\text{wt}(T)/|T| = 3/6$.

What about the element b? When b was first covered, T was not yet chosen, and at least 5 elements in T remained uncovered. Consequently, the charge assigned to b was at most 3/5. Reasoning similarly, the elements c, d, e, and f were assigned charges of at most 3/4, 3/3, 3/2, and 3/1, respectively. The total charge to elements in T is at most

$$3 \times (1/6 + 1/5 + 1/4 + 1/3 + 1/2 + 1/1) = \text{wt}(T) H_{|T|} \le \text{wt}(T) H_s.$$

This line of reasoning easily generalizes to show that for any set T, the elements in T are assigned a total charge of at most $\text{wt}(T)H_s$. ◇

Underlying duality. What role does duality and the primal-dual recipe play in the above analysis? A natural integer linear program for the weighted set cover problem is

$$\text{minimize} \sum_{S\in\mathcal{F}} \text{wt}(S)x(S)$$
$$\text{subject to} \begin{cases} \sum_{S\ni i} x(S) & \geq & 1 & \forall i \in \mathcal{U} \\ x(S) & \in & \{0,1\} & \forall S \in \mathcal{F}. \end{cases}$$

Relaxing this integer linear program yields the linear program

$$\text{minimize} \sum_{S\in\mathcal{F}} \text{wt}(S)x(S)$$
$$\text{subject to} \begin{cases} \sum_{S\ni i} x(S) & \geq & 1 & \forall i \in \mathcal{U} \\ x(S) & \geq & 0 & \forall S \in \mathcal{F}. \end{cases}$$

A solution to this linear program is called a *fractional* set cover. The dual is

$$\text{maximize} \sum_{i\in\mathcal{U}} y(i)$$
$$\text{subject to} \begin{cases} \sum_{i\in S} y(i) & \leq & \text{wt}(S) & \forall S \in \mathcal{F} \\ y(i) & \geq & 0 & \forall i \in \mathcal{U}. \end{cases}$$

The inequalities for the sets are called *packing constraints.* A solution to this dual linear program is called an *element packing.* In fact, the "charging" scheme in the analysis is just an element packing y, where $y(i)$ is the charge assigned to i divided by H_s. In this light, the previous analysis is simply constructing a dual solution and using it as a witness to show the performance guarantee.

8.1 Other problems having poly-logarithmic performance guarantees

Minimizing a Linear Function subject to a Submodular Constraint: This is a natural generalization of the weighted set cover problem. Rather than state the general problem, we give the following special case as an example: Given a family $\mathcal{F}$ of sets of n-vectors, with each set in $\mathcal{F}$ having a cost, find a subfamily of sets of minimum total cost whose union has rank n. A natural generalization of the greedy set cover algorithm gives a logarithmic performance guarantee (**?**).

Vertex-Weighted Network Steiner Tree: Like the network Steiner tree problem described in Section 7.1, an instance consists of a graph and a set of terminals; in this case, however, the graph can have vertex weights in addition to edge weights. An adaptation of the greedy algorithm achieves a logarithmic performance ratio.

Network Design Problems: This is a large class of problems generalizing the Steiner forest problem (see Section 7.1). An example of a problem in this class is *survivable network design*: given a weighted graph $G = (V, E)$ and a non-negative integer r_{uv} for each pair of vertices, find a minimum-cost set of edges $E' \subseteq E$ such that for every pair of vertices u and v, there are at least r_{uv} edge-disjoint paths connecting u and v in the graph $G = (V, E')$. A primal-dual approach, generalized from an algorithm for the Steiner forest problem, yields good performance guarantees for problems in this class. The performance guarantee depends on the particular problem; in some cases it is known to be bounded only logarithmically (**?**, Ch. 4). For a commercial application of this work see (**?**).

Graph Bisection: Given a graph, partition the nodes into two sets of equal size so as to minimize the number of edges with endpoints in different sets. An algorithm to find an approximately minimum-weight bisector would be remarkably useful, since it would provide the basis for a divide-and-conquer approach to many other graph optimization problems. In fact, a solution to a related but easier problem suffices.

Define a $\frac{1}{3}$-*balanced cut* to be a partition of the vertices of a graph into two sets each containing at least one-third of the vertices; its weight is the total weight of edges connecting the two sets. There is an algorithm to find a $\frac{1}{3}$-balanced cut whose weight is $O(\log n)$ times the minimum weight of a bisector. Note that this algorithm is not, strictly speaking, an approximation algorithm for any one optimization problem: the output of the algorithm is a solution to one problem while the quality of the output is measured against the optimal value for another. (We call this kind of performance guarantee a "bait-and-switch" guarantee.) Nevertheless, the algorithm is nearly as useful as a true approximation algorithm would be because in many divide-and-conquer algorithms the precise balance is not critical. One can make use of the balanced-cut algorithm to obtain approximation algorithms for many problems, including the following.

Optimal Linear Arrangement: Assign vertices of a graph to distinct integral points on the real number line so as to minimize the total length of edges.

Minimizing Time and Space for Sparse Gaussian Elimination: Given a sparse, positive-semidefinite linear system, the order in which variables are eliminated affects the time and storage space required for solving the system; choose an ordering to simultaneously minimize both time and storage space required.

Crossing Number: embed a graph in the plane so as to minimize the number of edge-crossings.

The approximation algorithms for the above three problems have performance guarantees that depend on the performance guarantee of the balanced-separator algorithm. It is not known whether the latter performance guarantee can be improved: there might be an algorithm for balanced separators that has a constant performance ratio.

There are several other graph-separation problems for which approximation algorithms are known, e.g. problems involving directed graphs. All these approximation algorithms for cut problems make use of linear-programming relaxation. See (**?**, Ch. 5).

9 Multi-criteria problems

In many applications, there are two or more objective functions to be considered. There have been some approximation algorithms developed for such *multi-criteria* optimization problems (though much work remains to be done). Several problems in previous sections, such as the k-cluster problem described in Section 7.1, can be viewed as a bi-criteria problem: there is a budget imposed on one resource (the number of clusters), and the algorithm is required to approximately optimize use of another resource (cluster diameter) subject to that budget constraint. Another example is scheduling unrelated parallel machines with costs: for a given budget on cost, jobs are assigned to machines in such a way that the cost of the assignment is under budget and the makespan of the schedule is nearly minimum.

Other approximation algorithms for bi-criteria problems use the bait-and-switch idea mentioned in Section 8.1. For example, there is a polynomial approximation scheme for variant of the minimum-spanning-tree problem in which there are two unrelated costs per edge, say weight and length: given a budget L on length, the algorithm finds a spanning tree whose length is at most $(1+\epsilon)L$ and whose weight is no more than the minimum weight of a spanning tree having length at most L (**?**).

10 Hard-to-approximate problems

For some optimization problems, worst-case performance guarantees are unlikely to be possible: it is NP-hard to approximate these problems even if one is willing to accept very poor performance guarantees. Following are some examples (**?**, §10.5,10.6).

Maximum Clique: Given a graph, find a largest set of vertices that are pairwise adjacent (see also (**?**)).

Minimum Vertex Coloring: Given a graph, color the vertices with a minimum number of colors so that adjacent vertices receive distinct colors.

Longest Path: Given a graph, find a longest simple path.

Max Linear Satisfy: Given a set of linear equations, find a largest possible subset that are simultaneously satisfiable.

Nearest Codeword: Given a linear error-correcting code specified by a matrix, and given a vector, find the codeword closest in Hamming distance to the vector.

Nearest Lattice Vector: Given a set of vectors $v_1, \ldots, v_n$ and a vector v, find an integer linear combination of the v_i that is nearest in Euclidean distance to v.

11 Research Issues and Summary

We have given examples for the techniques most frequently used to obtain approximation algorithms with provable performance guarantees, the use of witnesses, relaxation, and coarsening. We have categorized NP-hard optimization problems according to the performance guarantees achievable in polynomial time:

1. a small additive error,

2. a relative error of ϵ for any fixed positive ϵ,

3. a constant-factor performance guarantee,
4. a logarithmic- or polylogarithmic-factor performance guarantee,
5. no significant performance guarantee.

The ability to categorize problems in this way has been greatly aided by recent research developments in complexity theory. Novel techniques have been developed for proving the hardness of approximation of optimization problems. For many fundamental problems, we can state with considerable precision how good a performance guarantee can be achieved in polynomial time: known lower and upper bounds match or nearly match. Research towards proving matching bounds continues. In particular, for several problems for which there are logarithmic-factor performance guarantees (e.g. balanced cuts in graphs), researchers have so far not ruled out the existence of constant-factor performance guarantees.

Another challenge in research is methodological in nature. This chapter has presented methods of *worst-case analysis*: ways of universally bounding the error (relative or absolute) of an approximation algorithm. This theory has led to the development of many interesting and useful algorithms, and has proved useful in making distinctions between algorithms and between optimization problems. However, worst-case bounds are clearly not the whole story. Another approach is to develop algorithms tuned for a particular probability distribution of inputs, e.g. the uniform distribution. This approach is of limited usefulness because the distribution of inputs arising in a particular application rarely matches that for which the algorithm was tuned. Perhaps the most promising approach would address a hybrid of the worst-case and probabilistic models. The performance of an approximation algorithm would be defined as the probabilistic performance on a probability distribution selected by an adversary from among a large class of distributions. Blum (**?**) has has presented an analysis of this kind in the context of graph coloring, and others (see (**?**, 13.7) and (**?**)) have addressed similar issues in the context of on-line algorithms.

12 Defining Terms

ρ-approximation algorithm — An approximation algorithm that is guaranteed to find a solution whose value is at most (or at least, as appropriate) ρ times the optimum. The ratio ρ is the *performance ratio* of the algorithm.

absolute performance guarantee — An approximation algorithm with an absolute performance guarantee is guaranteed to return a feasible solution whose value differs additively from the optimal value by a bounded amount.

approximation algorithm — For solving an optimization problem. An algorithm that runs in time polynomial in the length of the input and outputs a feasible solution that is guaranteed to be nearly optimal in some well-defined sense called the *performance guarantee.*

coarsening — To *coarsen* a problem instance is to alter it, typically restricting to a less complex feasible region or objective function, so that the resulting problem can be efficiently solved, typically by dynamic programming. This is not standard terminology.

dual linear program — Every linear program has a corresponding linear program called the dual. For the linear program under **linear program**, the dual is $\max_y \left\{ b \cdot y : A^T y \leq c \text{ and } y \geq \overline{0} \right\}$. For any solution x to the original linear program and any solution y to the dual, we have $c \cdot x \geq (A^T y)^T x = y^T(Ax) \geq y \cdot b$. For optimal x and y, equality holds. For a problem formulated as an integer linear program, feasible solutions to the dual of a relaxation of the program can serve as witnesses.

feasible region — See **optimization problem**.

feasible solution — Any element of the feasible region of an optimization problem.

fractional solution — Typically, a solution to a relaxation of a problem.

fully polynomial approximation scheme — An approximation scheme in which the running time of A_ϵ is bounded by a polynomial in the length of the input and $1/\epsilon$.

integer linear program — A linear program augmented with additional constraints specifying that the variables must take on integer values. Solving such problems is NP-hard.

linear program — A problem expressible in the following form. Given an $n \times m$ real matrix A, m-vector b and n-vector c, determine $\min_x \{c \cdot x : Ax \geq b \text{ and } x \geq \overline{0}\}$ where x ranges over all n-vectors and the inequalities are interpreted component-wise (i.e. $x \geq \overline{0}$ means that the entries of x are non-negative).

MAX-SNP — A complexity class consisting of problems that have constant-factor approximation algorithms, but no approximation schemes unless P=NP.

mixed integer linear program — A linear program augmented with additional constraints specifying that some of the variables must take on integer values. Solving such problems is NP-hard.

objective function — See **optimization problem**.

optimal solution — To an optimization problem. A feasible solution minimizing (or possibly maximizing) the value of the objective function.

optimal value — The minimum (or possibly maximum) value taken on by the objective function over the feasible region of an optimization problem.

optimization problem — An optimization problem consists of a set $\mathcal{P}$, called the *feasible region* and usually specified implicitly, and a function $f : \mathcal{P} \to \mathbb{R}$, the *objective function.*

performance guarantee — See **approximation algorithm**.

performance ratio — See **ρ-approximation algorithm**.

polynomial approximation scheme — A collection of algorithms $\{A_\epsilon : \epsilon > 0\}$, where each A_ϵ is a $(1+\epsilon)$-approximation algorithm running in time polynomial in the length of the input. There is no restriction on the dependence of the running time on ϵ.

randomized rounding — A technique that uses the probabilistic method to convert a solution to a relaxed problem into an approximate solution to the original problem.

relative performance guarantee — An approximation algorithm with a relative performance guarantee is guaranteed to return a feasible solution whose value is bounded by a multiplicative factor times the optimal value.

relaxation — A relaxation of an optimization problem with feasible region $\mathcal{P}$ is another optimization problem with feasible region $\mathcal{P}' \supset \mathcal{P}$ and whose objective function is an extension of the original problem's objective function. The relaxed problem is typically easier to solve. Its value provides a bound on the value of the original problem.

rescalable — An optimization problem is rescalable if, given any instance of the problem and integer $\lambda > 0$, there is an easily computed second instance that is the same except that the objective function for the second instance is (element-wise) λ times the objective function of the first instance. For such problems, the best one can hope for is a multiplicative performance guarantee, not an absolute one.

semidefinite programming — A generalization of linear programming in which any subset of the variables may be constrained to form a semi-definite matrix. Used in recent results obtaining better approximation algorithms for cut, satisfiability, and coloring problems.

strongly NP-hard — A problem is strongly NP-hard if it is NP-hard even when any numbers appearing in the input are bounded by some polynomial in the length of the input.

triangle inequality — A complete weighted graph satisfies the triangle inequality if $\mathrm{wt}(u, v) \leq \mathrm{wt}(u, w) + \mathrm{wt}(w, v)$ for all vertices u, v, and w. This will hold for any graph representing points in a metric space. Many problems involving edge-weighted graphs have better approximation algorithms if the problem is restricted to weights satisfying the triangle inequality.

witness — A structure providing an easily verified bound on the optimal value of an optimization problem. Typically used in the analysis of an approximation algorithm to prove the performance guarantee.

References

Arora, S. (1996). Polynomial time approximation scheme for Euclidean TSP and other geometric problems. In (**?**), pages 2–11.

Blum, A., Jiang, T., Li, M., Tromp, J., and Yannakakis, M. (1994). Linear approximation of shortest superstrings. *Journal of the ACM*, 41(4):630–647.

Crescenzi, P. and Kann, V. (1995). A compendium of NP optimization problems. http://www.nada.kth.se/nada/theory/problemlist.html.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York.

Håstad, J. (1996). Clique is hard to approximate within $n^{1-\epsilon}$. In (?), pages 627–636.

Hochbaum, D. S., editor (1995). *Approximation Algorithms for NP-hard Problems.* PWS Publishing Co.

IEEE (1996). *37th Annual Symposium on Foundations of Computer Science*, Burlington, Vermont.

Johnson, D. S. (1974). Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278.

Li, M. (1990). Towards a DNA sequencing theory (learning a string) (preliminary version). In *31st Annual Symposium on Foundations of Computer Science*, volume I, pages 125–134, St. Louis, Missouri. IEEE.

Mihail, M., Shallcross, D., Dean, N., and Mostrel, M. (1996). A commercial application of survivable network design: ITP/INPLANS CCS network topology analyzer. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 279–287, Atlanta, Georgia.

Motwani, R. and Raghavan, P. (1995). *Randomized Algorithms.* Cambridge University Press.

Nemhauser, G. L. and Wolsey, L. A. (1988). *Integer and Combinatorial Optimization.* John Wiley and Sons, New York.

Ravi, R. and Goemans, M. X. (1996). The constrained minimum spanning tree problem. In *Proc. 5th Scand. Worksh. Algorithm Theory*, number 1097 in Lecture Notes in Computer Science, pages 66–75. Springer-Verlag.

Shmoys, D. B. Computing near-optimal solutions to combinatorial optimization problems. Series in Discrete Mathematics and Computer Science. AMS.

Further Information

For an excellent *survey* of the field of approximation algorithms, focusing on recent results and research issues, see the survey by David Shmoys (?). Further details on almost all of the topics in this chapter, including algorithms and hardness results, can be found in the definitive book edited by Dorit Hochbaum (?). NP-completeness is the subject of the classic book by Michael Garey and David Johnson (?). An article by Johnson anticipated many of the issues and methods subsequently developed (?). Randomized rounding and other probabilistic techniques used in algorithms are the subject of an excellent text by Motwani and Raghavan (?). As of this writing, a searchable compendium of approximation algorithms and hardness results, by Crescenzi and Kann, is available on-line (?).