I

Basic Methodologies

1

Introduction, Overview, and Notation

Teofilo F. Gonzalez
University of California, Santa Barbara

1.1 Introduction

Approximation algorithms, as we know them now, were formally introduced in the 1960s to generate near-optimal solutions to optimization problems that could not be solved efficiently by the computational techniques available at that time. With the advent of the theory of NP-completeness in the early 1970s, the area became more prominent as the need to generate near-optimal solutions for NP-hard optimization problems became the most important avenue for dealing with computational intractability. As established in the 1970s, for some problems one can generate near-optimal solutions quickly, while for other problems generating provably good suboptimal solutions is as difficult as generating optimal ones. Other approaches based on probabilistic analysis and randomized algorithms became popular in the 1980s. The introduction of new techniques to solve linear programming problems started a new wave for developing approximation algorithms that matured and saw tremendous growth in the 1990s. To deal, in a practical sense, with the inapproximable problems, there were a few techniques introduced in the 1980s and 1990s. These methodologies have been referred to as metaheuristics and include simulated annealing (SA), ant colony optimization (ACO), evolutionary computation (EC), tabu search (TS), and memetic algorithms (MA). Other previously established methodologies such as local search, backtracking, and branch-and-bound were also explored at that time. There has been a tremendous amount of research in metaheuristics during the past two decades. These techniques have been evaluated experimentally and have demonstrated their usefulness for solving practical problems. During the past 15 years or so, approximation algorithms have attracted considerably more attention. This was a result of a stronger inapproximability methodology that could be applied to a wider range of problems and the development of new approximation algorithms for problems arising in established and emerging application areas. Polynomial time approximation schemes (PTAS) were introduced in the 1960s and the more powerful fully polynomial time approximation schemes (FPTAS) were introduced in the 1970s. Asymptotic PTAS and FPTAS, and fully randomized approximation schemes were introduced later on.

Today, approximation algorithms enjoy a stature comparable to that of algorithms in general and the area of metaheuristics has established itself as an important research area. The new stature is a by-product of a natural expansion of research into more practical areas where solutions to real-world problems

are expected, as well as by the higher level of sophistication required to design and analyze these new procedures. The goal of approximation algorithms and metaheuristics is to provide the best possible solutions and to guarantee that such solutions satisfy certain important properties. This volume houses these two approaches and thus covers all the aspects of approximations. We hope it will serve as a valuable reference for approximation methodologies and applications.

Approximation algorithms and metaheuristics have been developed to solve a wide variety of problems. A good portion of these results have only theoretical value due to the fact that their time complexity is a high-order polynomial or have huge constants associated with their time complexity bounds. However, these results are important because they establish what is possible, and it may be that in the near future these algorithms will be transformed into practical ones. Other approximation algorithms do not suffer from this pitfall, but some were designed for problems with limited applicability. However, the remaining approximation algorithms have real-world applications. Given this, there is a huge number of important application areas, including new emerging ones, where approximation algorithms and metaheuristics have barely penetrated and where we believe there is an enormous potential for their use. Our goal is to collect a wide portion of the approximation algorithms and metaheuristics in as many areas as possible, as well as to introduce and explain in detail the different methodologies used to design these algorithms.

1.2 Overview

Our overview in this section is devoted mainly to the earlier years. The individual chapters discuss in detail recent research accomplishments in different subareas. This section will also serve as an overview of Parts I, II, and III of this handbook. Chapter 2 discusses some of the basic methodologies and applies them to simple problems. This prepares the reader for the overview of Parts IV, V, and VI presented in Chapter 2.

Even before the 1960s, research in applied mathematics and graph theory had established upper and lower bounds for certain properties of graphs. For example, bounds had been established for the chromatic number, achromatic number, chromatic index, maximum clique, maximum independent set, etc. Some of these results could be seen as the precursors of approximation algorithms. By the 1960s, it was understood that there were problems that could be solved efficiently, whereas for other problems all their known algorithms required exponential time. Heuristics were being developed to find quick solutions to problems that appeared to be computationally difficult to solve. Researchers were experimenting with heuristics, branch-and-bound procedures, and iterative improvement frameworks and were evaluating their performance when solving actual problem instances. There were many claims being made, not all of which could be substantiated, about the performance of the procedures being developed to generate optimal and suboptimal solutions to combinatorial optimization problems.

1.2.1 Approximation Algorithms

Forty years ago (1966), Ronald L. Graham [1] formally introduced approximation algorithms. He analyzed the performance of list schedules for scheduling tasks on identical machines, a fundamental problem in scheduling theory.

Problem: Scheduling tasks on identical machines.

Instance: Set of n tasks $(T_1, T_2, \ldots, T_n)$ with processing time requirements $t_1, t_2, \ldots, t_n$, partial order C defined over the set of tasks to enforce task dependencies, and a set of m identical machines.

Objective: Construct a schedule with minimum makespan. A *schedule* is an assignment of tasks to time intervals on the machines in such a way that (1) each task T_i is processed continuously for t_i units of time by one of the machines; (2) each machine processes at most one task at a time; and (3) the precedence constraints are satisfied (i.e., machines cannot commence the processing of a task until all its predecessors have been completed). The *makespan* of a schedule is the time at which all the machines have completed processing the tasks.

The *list scheduling* procedure is given an ordering of the tasks specified by a list L. The procedure finds the earliest time t when a machine is idle and an unassigned task is *available* (i.e., all its predecessors have

been completed). It assigns the leftmost available task in the list L to an idle machine at time t and this step is repeated until all the tasks have been scheduled.

The main result in Ref. [1] is proving that for every problem instance I, the schedule generated by this policy has a makespan that is bounded above by $(2 - 1/m)$ times the optimal makespan for the instance. This is called the *approximation ratio* or *approximation factor* for the algorithm. We also say that the algorithm is a $(2 - 1/m)$-approximation algorithm. This criterion for measuring the quality of the solutions generated by an algorithm remains one of the most important ones in use today. The second contribution in Ref. [1] is establishing that the approximation ratio $(2 - 1/m)$ is the best possible for list schedules, i.e., the analysis of the approximation ratio for this algorithm cannot be improved. This was established by presenting problem instances (for all m and $n \geq 2m - 1$) and lists for which the schedule generated by the procedure has a makespan equal to $2 - 1/m$ times the optimal makespan for the instance. A restricted version of the list scheduling algorithm is analyzed in detail in Chapter 2.

The third important result in Ref. [1] is showing that list scheduling procedures schedules may have anomalies. To explain this, we need to define some terms. The makespan of the list schedule, for instance, I, using list L is denoted by $f_L(I)$. Suppose that instance I' is a slightly modified version of instance I. The modification is such that we intuitively expect that $f_L(I') \leq f_L(I)$. But that is not always true, so there is an *anomaly*. For example, suppose that I' is I, except that I' has an additional machine. Intuitively, $f_L(I') \leq f_L(I)$ because with one additional machine tasks should be completed earlier or at the same time as when there is one fewer machine. But this is not always the case for list schedules, there are problem instances and lists for which $f_L(I') > f_L(I)$. This is called an *anomaly*. Our expectation would be valid if list scheduling would generate minimum makespan schedules. But we have a procedure that generates suboptimal solutions. Such guarantees are not always possible in this environment. List schedules suffer from other anomalies. For example, relaxing the precedence constraints or decreasing the execution time of the tasks. In both these cases, one would expect schedules with smaller or the same makespan. But, that is not always the case. Chapter 2 presents problem instances where anomalies occur. The main reason for discussing anomalies now is that even today numerous papers are being published and systems are being deployed where "common sense"-based procedures are being introduced without any analytical justification or thorough experimental validation. Anomalies show that since we live for the most part in a "suboptimal world," the effect of our decisions is not always the intended one.

Other classical problems with numerous applications are the traveling salesperson, Steiner tree, and spanning tree problems, which will be defined later on. Even before the 1960s, there were several well-known polynomial time algorithms to construct minimum-weight spanning trees for edge-weighted graphs [2]. These simple greedy algorithms have low-order polynomial time complexity bounds. It was well known at that time that the same type of procedures do not always generate an optimal tour for the traveling salesperson problem (TSP), and do not always construct optimal Steiner trees. However, in 1968 E. F. Moore (see Ref. [3]) showed that for any set of points P in metric space $L_M < L_T \leq 2L_S$, where L_M, L_T, and L_S are the weights of a minimum-weight spanning tree, a minimum-weight tour (solution) for the TSP and minimum-weight Steiner tree for P, respectively. Since every spanning tree is a Steiner tree, the above bounds show that when using a minimum-weight spanning tree to approximate a minimum weight Steiner tree we have a solution (tree) whose weight is at most twice the weight of an optimal Steiner tree. In other words, any algorithm that generates a minimum-weight spanning tree is a 2-approximation algorithm for the Steiner tree problem. Furthermore, this approximation algorithm takes the same time as an algorithm that constructs a minimum-weight spanning tree for edge-weighted graphs [2], since such an algorithm can be used to construct an optimal spanning tree for a set of points in metric space. The above bound is established by defining a transformation from any minimum-weight Steiner tree into a TSP tour with weight at most $2L_S$. Therefore, $L_T \leq 2L_S$ [3]. Then by observing that the deletion of an edge in an optimum tour for the TSP results in a spanning tree, it follows that $L_M < L_T$. Chapter 3 discusses this approximation algorithm in detail. The Steiner ratio is defined as L_S/L_M. The above arguments show that the Steiner ratio is at least $\frac{1}{2}$. Gilbert and Pollak [3] conjectured that the Steiner ratio in the Euclidean plane equals $\frac{\sqrt{3}}{2}$ (the 0.86603 . . . conjecture). The proof of this conjecture and improved approximation algorithms for different versions of the Steiner tree problem are discussed in Chapters 42.

The above constructive proof can be applied to a minimum-weight spanning tree to generate a tour for the TSP. The construction takes polynomial time and results in a 2-approximation algorithm for the TSP. This approximation algorithm for the TSP is also referred to as the *double spanning tree algorithm* and is discussed in Chapters 3 and 31. Improved approximation algorithms for the TSP as well as algorithms for its generalizations are discussed in Chapters 3, 31, 40, 41, and 51. The approximation algorithm for the Steiner tree problem just discussed is explained in Chapter 3 and improved approximation algorithms and applications are discussed in Chapters 42, 43, and 51. Chapter 59 discusses approximation algorithms for variations of the spanning tree problem.

In 1969, Graham [4] studied the problem of scheduling tasks on identical machines, but restricted to independent tasks, i.e., the set of precedence constraints is empty. He analyzes the longest processing time (LPT) scheduling rule; this is list scheduling where the list of tasks L is arranged in nonincreasing order of their processing requirements. His elegant proof established that the LPT procedure generates a schedule with makespan at most $\frac{4}{3}-\frac{1}{3m}$ times the makespan of an optimal schedule, i.e., the LPT scheduling algorithm has a $\frac{4}{3}-\frac{1}{3m}$ approximation ratio. He also showed that the analysis is best possible for all m and $n \geq 2m+1$. For $n \leq 2m$ tasks, the approximation ratio is smaller and under some conditions LPT generates an optimal makespan schedule. Graham [4], following a suggestion by D. Kleitman and D. Knuth, considered list schedules where the first portion of the list L consists of k tasks with the longest processing times arranged by their starting times in an optimal schedule for these k tasks (only). Then the list L has the remaining $n-k$ tasks in any order. The approximation ratio for this list schedule using list L is $1+\frac{1-1/m}{1+\lfloor k/m \rfloor}$. An optimal schedule for the longest k tasks can be constructed in $O(km^k)$ time by a straightforward branch-and-bound algorithm. In other words, this algorithm has approximation ratio $1+\epsilon$ and time complexity $O(n \log m + m^{(m-1-\epsilon m)/\epsilon})$. For any fixed constants m and ϵ, the algorithm constructs in polynomial (linear) time with respect to n a schedule with makespan at most $1+\epsilon$ times the optimal makespan. Note that for a fixed constant m, the time complexity is polynomial with respect to n, but it is not polynomial with respect to $1/\epsilon$. This was the first algorithm of its kind and later on it was called a *polynomial time approximation scheme*. Chapter 9 discusses different PTASs. Additional PTASs appear in Chapters 42, 45, and 51. The proof techniques presented in Refs. [1,4] are outlined in Chapter 2, and have been extended to apply to other problems. There is an extensive body of literature for approximation algorithms and metaheuristics for scheduling problems. Chapters 44, 45, 46, 47, 73, and 81 discuss interesting approximation algorithms and heuristics for scheduling problems. The recent scheduling handbook [5] is an excellent source for scheduling algorithms, models, and performance analysis.

The development of NP-completeness theory in the early 1970s by Cook [6] and Karp [7] formally introduced the notion that there is a large class of decision problems (the answer to these problems is a simple yes or no) that are computationally equivalent. By this, it is meant that either every problem in this class has a polynomial time algorithm that solves it, or none of them do. Furthermore, this question is the same as the $P = NP$ question, an open problem in computational complexity. This question is to determine whether or not the set of languages recognized in polynomial time by deterministic Turing machines is the same as the set of languages recognized in polynomial time by nondeterministic Turing machines. The conjecture has been that $P \neq NP$, and thus the hardest problems in NP cannot be solved in polynomial time. These computationally equivalent problems are called *NP-complete* problems. The scheduling on identical machines problem discussed earlier is an optimization problem. Its corresponding decision problem has its input augmented by an integer value B and the yes-no question is to determine whether or not there is a schedule with makespan at most B. An optimization problem whose corresponding decision problem is NP-complete is called an *NP-hard* problem. Therefore, scheduling tasks on identical machines is an NP-hard problem. The TSP and the Steiner tree problem are also NP-hard problems. The minimum-weight spanning tree problem can be solved in polynomial time and is not an NP-hard problem under the assumption that $P \neq NP$. The next section discusses NP-completeness in more detail. There is a long list of practical problems arising in many different fields of study that are known to be NP-hard problems [8]. Because of this, the need to cope with these computationally intractable problems was recognized earlier on. This is when approximation algorithms became a central area of research activity. Approximation algorithms offered a way to circumvent computational intractability by paying a price when it comes to the quality of the solution generated. But a solution can be generated quickly. In other

words and another language, "no te fijes en lo bien, fijate en lo rápido." Words that my mother used to describe my ability to play golf when I was growing up.

In the early 1970s Garey et al. [9] as well as Johnson [10,11] developed the first set of polynomial time approximation algorithms for the bin packing problem. The analysis of the approximation ratio for these algorithms is asymptotic, which is different from those for the scheduling problems discussed earlier. We will define this notion precisely in the next section, but the idea is that the ratio holds when the optimal solution value is greater than some constant. Research on the bin packing problem and its variants has attracted very talented investigators who have generated more than 650 papers, most of which deal with approximations. This work has been driven by numerous applications in engineering and information sciences (see Chapters 32–35).

Johnson [12] developed polynomial time algorithms for the sum of subsets, max satisfiability, set cover, graph coloring, and max clique problems. The algorithms for the first two problems have a constant ratio approximation, but for the other problems the approximation ratio is $\ln n$ and n^{ϵ}. Sahni [13,14] developed a PTAS for the knapsack problem. Rosenkrantz et al. [15] developed several constant ratio approximation algorithms for the TSP. This version of the problem is defined over edge-weighted complete graphs that satisfy the triangle inequality (or simply metric graphs), rather than for points in metric space as in Ref. [3]. These algorithms have an approximation ratio of 2.

Sahni and Gonzalez [16] showed that there were a few NP-hard optimization problems for which the existence of a constant ratio polynomial time approximation algorithm implies the existence of a polynomial time algorithm to generate an optimal solution. In other words, for these problems the complexity of generating a constant ratio approximation and an optimal solution are computationally equivalent problems. For these problems, the approximation problem is NP-hard or simply inapproximable (under the assumption that $P \neq NP$). Later on, this notion was extended to mean that there is no polynomial time algorithm with approximation ratio r for a problem under some complexity theoretic hypothesis. The approximation ratio r is called the *in-approximability ratio*, and r may be a function of the input size (see Chapter 17).

The k-min-cluster problem is one of these inapproximable problems. Given an edge-weighted undirected graph, the k-min-cluster problem is to partition the set of vertices into k sets so as to minimize the sum of the weight of the edges with endpoints in the same set. The k-maxcut problem is defined as the k-min-cluster problem, except that the objective is to maximize the sum of the weight of the edges with endpoints in different sets. Even though these two problems have exactly the same set of feasible and optimal solutions, there is a linear time algorithm for the k-maxcut problem that generates k-cuts with weight at least $\frac{k-1}{k}$ times the weight of an optimal k-cut [16], whereas approximating the k-min-cluster problem is a computationally intractable problem. The former problem has the property that a near-optimal solution may be obtained as long as partial decisions are made optimally, whereas for the k-min-cluster an optimal partial decision may turn out to force a terrible overall solution.

Another interesting problem whose approximation problem is NP-hard is the TSP [16]. This is not exactly the same version of the TSP discussed above, which we said has several constant ratio polynomial time approximation algorithms. Given an edge-weighted undirected graph, the TSP is to find a least weight tour, i.e., find a least weight (simple) path that starts at vertex 1, visits each vertex in the graph *exactly* once, and ends at vertex 1. The weight of a path is the sum of the weight of its edges. The version of the TSP studied in Ref. [15] is limited to metric graphs, i.e., the graph is complete (all the edges are present) and the set of edge weights satisfies the triangle inequality (which means that the weight of the edge joining vertex i and j is less than or equal to the weight of any path from vertex i to vertex j). This version of the TSP is equivalent to the one studied by E. F. Moore [3]. The approximation algorithms given in Refs. [3,15] can be adapted easily to provide a constant-ratio approximation to the version of the TSP where the tour is defined as visiting each vertex in the graph *at least* once. Since Moore's approximation algorithms for the metric Steiner tree and metric TSP are based on the same idea, one would expect that the Steiner tree problem defined over arbitrarily weighted graphs is NP-hard to approximate. However, this is not the case. Moore's algorithm [3] can be modified to be a 2-approximation algorithm for this more general Steiner tree problem.

As pointed out in Ref. [17], Levner and Gens [18] added a couple of problems to the list of problems that are NP-hard to approximate. Garey and Johnson [19] showed that the max clique problem has the

property that if for some constant r there is a polynomial time r-approximation algorithm, then there is a polynomial time r'-approximation algorithm for any constant r' such that $0 < r' < 1$. Since at that time researchers had considered many different polynomial time algorithms for the clique problem and none had a constant ratio approximation, it was conjectured that none existed, under the assumption that $P \neq NP$. This conjecture has been proved (see Chapter 17).

A PTAS is said to be an FPTAS if its time complexity is polynomial with respect to n (the problem size) and $1/\epsilon$. The first FPTAS was developed by Ibarra and Kim [20] for the knapsack problem. Sahni [21] developed three different techniques based on rounding, interval partitioning, and separation to construct FPTAS for sequencing and scheduling problems. These techniques have been extended to other problems and are discussed in Chapter 10. Horowitz and Sahni [22] developed FPTAS for scheduling on processors with different processing speed. Reference [17] discusses a simple $O(n^3/\epsilon)$ FPTAS for the knapsack problem developed by Babat [23,24]. Lawler [25] developed techniques to speed up FPTAS for the knapsack and related problems. Chapter 10 presents different methodologies to design FPTAS. Garey and Johnson [26] showed that if any problem in a class of NP-hard optimization problems that satisfy certain properties has a FPTAS, then $P = NP$. The properties are that the objective function value of every feasible solution is a positive integer, and the problem is *strongly* NP-hard. Strongly *NP*-hard means that the problem is NP-hard even when the magnitude of the maximum number in the input is bounded by a polynomial on the input length. For example, the TSP is strongly NP-hard, whereas the knapsack problem is not, under the assumption that $P \neq NP$ (see Chapter 10).

Lin and Kernighan [27] developed elaborate heuristics that established experimentally that instances of the TSP with up to 110 cities can be solved to optimality with 95% confidence in $O(n^2)$ time. This was an iterative improvement procedure applied to a set of randomly selected feasible solutions. The process was to perform k pairs of link (edge) interchanges that improved the length of the tour. However, Papadimitriou and Steiglitz [28] showed that for the TSP no local optimum of an efficiently searchable neighborhood can be within a constant factor of the optimum value unless $P = NP$. Since then, there has been quite a bit of research activity in this area. Deterministic and stochastic local search in efficiently searchable as well as in very large neighborhoods are discussed in Chapters 18–21. Chapter 14 discusses issues relating to the empirical evaluation of approximation algorithms and metaheuristics.

Perhaps the best known approximation algorithm is the one by Christofides [29] for the TSP defined over metric graphs. The approximation ratio for this algorithm is $\frac{3}{2}$, which is smaller than the approximation ratio of 2 for the algorithms reported in Refs. [3,15]. However, looking at the bigger picture that includes the time complexity of the approximation algorithms, Christofides algorithm is not of the same order as the ones given in Refs. [3,15]. Therefore, neither set of approximation algorithms dominates the other as one set has a smaller time complexity bound, whereas the other (Christofides algorithm) has a smaller worst-case approximation ratio.

Ausiello et al. [30] introduced the differential ratio, which is another way of measuring the quality of the solutions generated by approximation algorithms. The differential ratio destroys the artificial dissymmetry between "equivalent" minimization and maximization problems (e.g., the k-max cut and the k-min-cluster discussed above) when it comes to approximation. This ratio uses the difference between the worst possible solution and the solution generated by the algorithm, divided by the difference between the worst solution and the best solution. Cornuejols et al. [31] also discussed a variation of the differential ratio approximations. They wanted the ratio to satisfy the following property: "A modification of the data that adds a constant to the objective function value should also leave the error measure unchanged." That is, the "error" by the approximation algorithm should be the same as before. Differential ratio and its extensions are discussed in Chapter 16, along with other similar notions [30]. Ausiello et al. [30] also introduced *reductions that preserve approximability*. Since then, there have been several new types of approximation preserving reductions. The main advantage of these reductions is that they enable us to define large classes of optimization problems that behave in the same way with respect to approximation. Informally, the class of NP-optimization problems, **NPO**, is the set of all optimization problems Π that can be "recognized" in polynomial time (see Chapter 15 for a formal definition). An **NPO** problem Π is said to be in **APX**, if it has a constant approximation ratio polynomial time algorithm. The class **PTAS** consists of all **NPO**

problems that have PTAS. The class **FPTAS** is defined similarly. Other classes, **Poly-APX**, **Log-APX**, and **Exp-APX**, have also been defined (see Chapter 15).

One of the main accomplishments at the end of the 1970s was the development of a polynomial time algorithm for linear programming problems by Khachiyan [32]. This result had a tremendous impact on approximation algorithms research, and started a new wave of approximation algorithms. Two subsequent research accomplishments were at least as significant as Khachiyan's [32] result. The first one was a faster polynomial time algorithm for solving linear programming problems developed by Karmakar [33]. The other major accomplishment was the work of Grötschel et al. [34,35]. They showed that it is possible to solve a linear programming problem with an exponential number of constraints (with respect to the number of variables) in time which is polynomial in the number of variables and the number of bits used to describe the input, given a *separation oracle* plus a bounding ball and a lower bound on the volume of the feasible solution space. Given a solution, the separation oracle determines in polynomial time whether or not the solution is feasible, and if it is not it finds a constraint that is violated. Chapter 11 gives an example of the use of this approach. Important developments have taken place during the past 20 years. The books [35,36] are excellent references for linear programming theory, algorithms, and applications.

Because of the above results, the approach of formulating the solution to an NP-hard problem as an integer linear programming problem and then solving the corresponding linear programming problem became very popular. This approach is discussed in Chapter 2. Once a fractional solution is obtained, one uses rounding to obtain a solution to the original NP-hard problem. The rounding may be deterministic or randomized, and it may be very complex (metarounding). LP rounding is discussed in Chapters 2, 4, 6–9, 11, 12, 37, 45, 57, 58, and 70.

Independently, Johnson [12] and Lovász [37] developed efficient algorithms for the set cover with approximation ratio of $1 + \ln d$, where d is the maximum number of elements in each set. Chvátal [38] extended this result to the weighted set cover problem. Subsequently, Hochbaum [39] developed an algorithm with approximation ratio f, where f is the maximum number of sets containing any of the elements in the set. This result is normally inferior to the one by Chvátal [38], but is more attractive for the weighted vertex cover problem, which is a restricted version of the weighted set cover. For this subproblem, it is a 2-approximation algorithm. A few months after Hochbaum's initial result,[1] Bar-Yehuda and Even [40] developed a primal-dual algorithm with the same approximation ratio as the one in [39]. The algorithm in [40] does not require the solution of an LP problem, as in the case of the algorithm in [39], and its time complexity is linear. But it uses linear programming theory. This was the first primal-dual approximation algorithm, though some previous algorithms may also be viewed as falling into this category. An application of the primal-dual approach, as well as related ones, is discussed in Chapter 2. Chapters 4, 37, 39, 40, and 71 discuss several primal-dual approximation algorithms. Chapter 13 discusses "distributed" primal-dual algorithms. These algorithms make decisions by using only "local" information.

In the mid 1980s, Bar-Yehuda and Even [41] developed a new framework parallel to the primal-dual methods. They call it *local ratio*; it is simple and requires no prior knowledge of linear programming. In Chapter 2, we explain the basics of this approach, and recent developments are discussed in [42].

Raghavan and Thompson [43] were the first to apply randomized rounding to relaxations of linear programming problems to generate solutions to the problem being approximated. This field has grown tremendously. LP randomized rounding is discussed in Chapters 2, 4, 6–8, 11, 12, 57, 70, and 80 and deterministic rounding is discussed in Chapters 2, 6, 7, 9, 11, 37, 45, 57, 58, and 70. A disadvantage of LP-rounding is that a linear programming problem needs to be solved. This takes polynomial time with

[1] Here, we are referring to the time when these results appeared as technical reports. Note that from the journal publication dates, the order is reversed. You will find similar patterns throughout the chapters. To add to the confusion, a large number of papers have also been published in conference proceedings. Since it would be very complex to include the dates when the initial technical report and conference proceedings were published, we only include the latest publication date. Please keep this in mind when you read the chapters and, in general, the computer science literature.

respect to the input length, but in this case it means the number of bits needed to represent the input. In contrast, algorithms based on the primal-dual approach are for the most part faster, since they take polynomial time with respect to the number of "objects" in the input. However, the LP-rounding approach can be applied to a much larger class of problems and it is more robust since the technique is more likely to be applicable after changing the objective function or constraints for a problem.

The first APTAS (asymptotic PTAS) was developed by Fernandez de la Vega and Lueker [44] for the bin packing problem. The first AFPTAS (Asymptotic FPTAS) for the same problem was developed by Karmakar and Karp [45]. These approaches are discussed in Chapter 16. Fully polynomial randomized approximation schemes (FPRAS) are discussed in Chapter 12.

In the 1980s, new approximation algorithms were developed as well as PTAS and FPTAS based on different approaches. These results are reported throughout the handbook. One difference was the application of approximation algorithms to other areas of research activity (very large-scale integration (VLSI), bioinformatics, network problems) as well as to other problems in established areas.

In the late 1980s, Papadimitriou and Yannakakis [46] defined *MAXSNP* as a subclass of NPO. These problems can be approximated within a constant factor and have a nice logical characterization. They showed that if MAX3SAT, vertex cover, MAXCUT, and some other problems in the class could be approximated in polynomial time with an arbitrary precision, then all MAXSNP problems have the same property. This fact was established by using *approximation preserving* reductions (see Chapters 15 and 17). In the 1990s, Arora et al. [47], using complex arguments (see Chapter 17), showed that MAX3SAT is hard to approximate within a factor of $1 + \epsilon$ for some $\epsilon > 0$ unless $P = NP$. Thus, all problems in MAXSNP do not admit a PTAS unless $P = NP$. This work led to major developments in the area of approximation algorithms, including inapproximability results for other problems, a bloom of approximation preserving reductions, discovery of new inapproximability classes, and construction of approximation algorithms achieving optimal or near optimal approximation ratios.

Feige et al. [48] showed that the clique problem could not be approximated to within some constant value. Applying the previous result in Ref. [26] showed that the clique problem is inapproximable to within any constant. Feige [49] showed that the set cover is inapproximable within $\ln n$. Other inapproximable results appear in Refs. [50,51]. Chapter 17 discusses all of this work in detail.

There are many other very interesting results that have been published in the past 15 years. Goemans and Williamson [52] developed improved approximation algorithms for the maxcut and satisfiability problems using *semidefinite programming* (SDP). This seminal work opened a new venue for the design of approximation algorithms. Chapter 15 discusses this work as well as recent developments in this area. Goemans and Williamson [53] also developed powerful techniques for designing approximation algorithms based on the primal-dual approach. The dual-fitting and factor revealing approach is used in Ref. [54]. Techniques and extensions of these approaches are discussed in Chapters 4, 13, 37, 39, 40, and 71.

In the past couple of decades, we have seen approximation algorithms being applied to traditional combinatorial optimization problems as well as problems arising in other areas of research activity. These areas include VLSI design automation, networks (wired, sensor and wireless), bioinformatics, game theory, computational geometry, and graph problems. In Section 2, we elaborate further on these applications.

1.2.2 Local Search, Artificial Neural Networks, and Metaheuristics

Local search techniques have a long history; they range from simple constructive and iterative improvement algorithms to rather complex methods that require significant fine-tuning, such as evolutionary algorithms (EAs) or SA. Local search is perhaps one of the most natural ways to attempt to find an optimal or suboptimal solution to an optimization problem. The idea of local search is simple: start from a solution and improve it by making local changes until no further progress is possible. Deterministic local search algorithms are discussed in Chapter 18. Chapter 19 covers stochastic local search algorithms. These are local search algorithms that make use of randomized decisions, for example, in the context of generating initial solutions or when determining search steps. When the neighborhood to search for the next solution is very large,

finding the best neighbor to move to is many times an NP-hard problem. Therefore, a suboptimal solution is needed at this step. In Chapter 20, the issues related to very large-scale neighborhood search are discussed from the theoretical, algorithmic, and applications point of view.

Reactive search advocates the use of simple sub symbolic machine learning to automate the parameter tuning process and make it an integral (and fully documented) part of the algorithm. Parameters are normally tuned through a feedback loop that many times depends on the user. Reactive search attempts to mechanize this process. Chapter 21 discusses issues arising during this process.

Artificial neural networks have been proposed as a tool for machine learning and many results have been obtained regarding their application to practical problems in robotics control, vision, pattern recognition, grammatical inferences, and other areas. Once trained, the network will compute an input/output mapping that, if the training data was representative enough, will closely match the unknown rule that produced the original data. Neural networks are discussed in Chapter 22.

The work of Lin and Kernighan [27] as well as that of others sparked the study of modern heuristics, which have evolved and are now called *metaheuristics*. The term metaheuristics was coined by Glover [55] in 1986 and in general means "to find beyond in an upper level." Metaheuristics include Tabu Search (TS), Simulated Annealing (SA), Ant Colony Optimization, Evolutionary Computation (EC), iterated local search (ILC), and Memetic Algorithms (MA). One of the motivations for the study of metaheuristics is that it was recognized early on that constant ratio polynomial time approximation algorithms are not likely to exist for a large class of practical problems [16]. Metaheuristics do not guarantee that near-optimal solutions will be found quickly for all problem instances. However, these complex programs do find near-optimal solutions for many problem instances that arise in practice. These procedures have a wide range of applicability. This is the most appealing aspect of metaheuristics.

The term Tabu Search (TS) was coined by Glover [55]. TS is based on *adaptive memory* and *responsive exploration*. The former allows for the effective and efficient search of the solution space. The latter is used to guide the search process by imposing restraints and inducements based on the information collected. Intensification and diversification are controlled by the information collected, rather than by a random process. Chapter 23 discusses many different aspects of TS as well as problems to which it has been applied.

In the early 1980s Kirkpatrick et al. [56] and, independently, Černý [57] introduced Simulated Annealing (SA) as a randomized local search algorithm to solve combinatorial optimization problems. *SA* is a local search algorithm, which means that it starts with an initial solution and then searches through the solution space by iteratively generating a new solution that is "near" it. Sometimes, the moves are to a worse solution to escape local optimal solutions. This method is based on statistical mechanics (Metropolis algorithm). It was heavily inspired by an analogy between the physical annealing process of solids and the problem of solving large combinatorial optimization problems. Chapter 25 discusses this approach in detail.

Evolutionary Computation (EC) is a metaphor for building, applying, and studying algorithms based on Darwinian principles of natural selection. Algorithms that are based on evolutionary principles are called *evolutionary algorithms* (EA). They are inspired by nature's capability to evolve living beings well adapted to their environment. There has been a variety of slightly different EAs proposed over the years. Three different strands of EAs were developed independently of each other over time. These are *evolutionary programming* (EP) introduced by Fogel [58] and Fogel et al. [59], *evolutionary strategies* (ES) proposed by Rechenberg [60], and *genetic algorithms* (GAs) initiated by Holland [61]. GAs are mainly applied to solve discrete problems. *Genetic programming* (GP) and *scatter search* (SS) are more recent members of the EA family. EAs can be understood from a unified point of view with respect to their main components and the way they explore the search space. EC is discussed in Chapter 24.

Chapter 17 presents an overview of Ant Colony Optimization (ACO)—a metaheuristic inspired by the behavior of real ants. ACO was proposed by Dorigo and colleagues [62] in the early 1990s as a method for solving hard combinatorial optimization problems. ACO algorithms may be considered to be part of *swarm intelligence*, the research field that studies algorithms inspired by the observation of the behavior of *swarms*. Swarm intelligence algorithms are made up of simple individuals that cooperate through self-organization.

Memetic Algorithms (MA) were introduced by Moscato [63] in the late 1980s to denote a family of metaheuristics that can be characterized as the hybridization of different algorithmic approaches for a

given problem. It is a population-based approach in which a set of cooperating and competing agents are engaged in periods of individual improvement of the solutions while they sporadically interact. An important component is *problem and instance-dependent knowledge*, which is used to speed-up the search process. A complete description is given in Chapter 27.

1.2.3 Sensitivity Analysis, Multiobjective Optimization, and Stability

Chapter 30 covers *sensitivity analysis*, which has been around for more than 40 years. The aim is to study how variations affect the optimal solution value. In particular, parametric analysis studies problems whose structure is fixed, but where cost coefficients vary continuously as a function of one or more parameters. This is important when selecting the model parameters in optimization problems. In contrast, Chapter 31 considers a newer area, which is called *stability*. By this we mean how the complexity of a problem depends on a parameter whose variation alters the space of allowable instances.

Chapters 28 and 29 discuss *multiobjective combinatorial optimization*. This is important in practice since quite often a decision is rarely made with only one criterion. There are many examples of such applications in the areas of transportation, communication, biology, finance, and also computer science. Approximation algorithms and a FPTAS for multiobjective optimization problems are discussed in Chapter 28. Chapter 29 covers stochastic local search algorithms for multiobjective optimization problems.

1.3 Definitions and Notation

One can use many different criteria to judge approximation algorithms and heuristics. For example the quality of solution generated, and the time and space complexity needed to generate it. One may measure the criteria in different ways, e.g., we could use the worst case, average case, median case, etc. The evaluation could be analytical or experimental. Additional criteria include characterization of data sets where the algorithm performs very well or very poorly; comparison with other algorithms using benchmarks or data sets arising in practice; tightness of bounds (for quality of solution, time and space complexity); the value of the constants associated with the time complexity bound including the ones for the lower order terms; and so on. For some researchers, the most important aspect of an approximation algorithm is that it is complex to analyze, but for others it is more important that the algorithm be complex and involve the use of sophisticated data structures. For researchers working on problems directly applicable to the "real world," experimental evaluation or evaluation on benchmarks is a more important criterion. Clearly, there is a wide variety of criteria one can use to evaluate approximation algorithms. The chapters in this handbook use different criteria to evaluate approximation algorithms.

For any given optimization problem P, let A_1, A_2, ... be the set of current algorithms that generate a feasible solution for each instance of problem P. Suppose that we select a set of criteria C and a way to measure it that we feel is the most important. How can we decide which algorithm is best for problem P with respect to C? We may visualize every algorithm as a point in multidimensional space. Now, the approach used to compare feasible solutions for multiobjective function problems (see Chapters 28 and 29) can also be used in this case to label some of the algorithms as current Pareto optimal with respect to C. Algorithm A is said to be *dominated* by algorithm B with respect to C, if for each criterion $c \in C$ algorithm B is "not worse" than A, and for at least one criterion $c \in C$ algorithm B is "better" than A. An algorithm is said to be a *current Pareto optimal* algorithm with respect to C if none of the current algorithms dominates it.

In the next subsections, we define time and space complexity, NP-completeness, and different ways to measure the quality of the solutions generated by the algorithms.

1.3.1 Time and Space Complexity

There are many different ways one can use to judge algorithms. The main ones we use are the time and space required to solve the problem. This can be expressed in terms on n, the input size. It can be evaluated

empirically or analytically. For the analytical evaluation, we use the time and space complexity of the algorithm. Informally, this is a way to express the time the algorithm takes to solve a problem of size n and the amount of space needed to run the algorithm.

It is clear that almost all algorithms take different time to execute with different data sets even when the input size is the same. If you code it and run it on a computer you will see more variation depending on the different hardware and software installed in the system. It is impossible to characterize exactly the time and space required by an algorithm. We need a short cut. The approach that has been taken is to count the number of "operations" performed by the algorithm in terms of the input size. "Operations" is not an exact term and refers to a set of "instructions" whose number is independent of the problem size. Then we just need to count the total number of operations.

Counting the number of operations exactly is very complex for a large number of algorithms. So we just take into consideration the "highest"-order term. This is the O notation.

Big "oh" notation: A (positive) function $f(n)$ is said to be $O(g(n))$ if there exist two constants $c \geq 1$ and $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

The function $g(n)$ is the highest-order term. For example, if $f(n) = n^3 + 20n^2$, then $g(n) = n^3$. Setting $n_0 = 1$ and $c = 21$ shows that $f(n)$ is $O(n^3)$. Note that $f(n)$ is also $O(n^4)$, but we like $g(n)$ to be the function with the smallest possible growth. The function $f(n)$ cannot be $O(n^2)$ because it is impossible to find constants c and n_0 such that $n^3 + 20n^2 \leq cn^2$ for all $n \geq n_0$.

The time and space complexity of an algorithm is expressed in the O notation and describes their growth rate in terms of the problem size. Normally, the problem size is the number of vertices and edges in a graph, the number of tasks and machines in a scheduling problem, etc. But it can also be the number of bits used to represent the input.

When comparing two algorithms expressed in O notation, we have to be careful because the constants c and n_0 are hidden. For large n, the algorithm with the smallest growth rate is the better one. When two algorithms have similar constants c and n_0, the algorithm with the smallest growth function has a smaller running time. The book [2] discusses in detail the O notation as well as other notation.

1.3.2 NP-Completeness

Before the 1970s, researchers were aware that some problems could be computationally solved by algorithms with (low) polynomial time complexity ($O(n)$, $O(n^2)$, $O(n^3)$, etc.), whereas other problems had exponential time complexity, for example, $O(2^n)$ and $O(n!)$. It was clear that even for small values of n, exponential time complexity equates to computational intractability if the algorithm actually performs an exponential number of operations for some inputs. The convention of computational tractability being equated to polynomial time complexity does not really fit well, as an algorithm with time complexity $O(n^{100})$ is not really tractable if it actually performs n^{100} operations. But even under this relaxation of "tractability," there is a large class of problems that does not seem to have computationally tractable algorithms.

We have been discussing optimization problems. But NP-completeness is defined with respect to decision problems. A decision problem is simply one whose answer is "yes" or "no." The scheduling on identical machines problems discussed earlier is an optimization problem. Its corresponding decision problem has its input augmented by an integer value B and the yes-no question is to determine whether or not there is a schedule with makespan at most B. Every optimization problem has a corresponding decision problem. Since the solution of an optimization problem can be used directly to solve the decision problem, we say that the optimization problem is at least as hard to solve as the decision problem. If we show that the decision problem is a computationally intractable problem, then the corresponding optimization problem is also intractable.

The development of NP-completeness theory in the early 1970s by Cook [6] and Karp [7] formally introduced the notion that there is a large class of decision problems that are computationally equivalent. By this we mean that either every problem in this class has a polynomial time algorithm that solves it, or none of them do. Furthermore, this question is the same as the $P = NP$ question, an open problem in

computational complexity. This question is to determine whether or not the set of languages recognized in polynomial time by deterministic Turing machines is the same as the set of languages recognized in polynomial time by nondeterministic Turing machines. The conjecture has been that $P \neq NP$, and thus the problems in this class do not have polynomial time algorithms for their solution. The decision problems in this class of problems are called *NP-complete* problems. Optimization problems whose corresponding decision problems are NP-complete are called *NP-hard* problems.

Scheduling tasks on identical machines is an NP-hard problem. The TSP and Steiner tree problem are also NP-hard problems. The minimum-weight spanning tree problem can be solved in polynomial and it is not an NP-hard problem, under the assumption that $P \neq NP$. There is a long list of practical problems arising in many different fields of study that are known to be NP-hard problems. In fact, almost all the optimization problems discussed in this handbook are NP-hard problems. The book [8] is an excellent source of information for NP-complete and NP-hard problems.

One establishes that a problem Q is an NP-complete problem by showing that the problem is in NP and giving a polynomial time transformation from an NP-complete problem to the problem Q.

A problem is said to be in *NP* if one can show that a yes answer to it can be verified in polynomial time. For the scheduling problem defined above, you may think of this as providing a procedure that given any instance of the problem and an assignment of tasks to machines, the algorithm verifies in polynomial time, with respect to the problem instance size, that the assignment is a schedule and its makespan is at most B. This is equivalent to the task a grader does when grading a question of the form "Does the following instance of the scheduling problem have a schedule with makespan at most 300? If so, give a schedule." Just verifying that the "answer" is correct is a simple problem. But solving a problem instance with 10,000 tasks and 20 machines seems much harder than simply grading it. In our oversimplification, it seems that $P \neq NP$. Polynomial time verification of a yes answer does not seem to imply polynomial time solvability.

A polynomial time transformation from decision problem P_1 to decision problem P_2 is an algorithm that takes as input any instance I of problem P_1 and constructs an instance $f(I)$ of P_2. The algorithm must take polynomial time with respect to the size of the instance I. The transformation must be such that $f(I)$ is a yes-instance of P_2 if, and only if, I is a yes-instance of P_1.

The implication of a polynomial transformation $P_1 \alpha P_2$ is that if P_2 can be solved in polynomial time, then so can P_1, and if P_1 cannot be solved in polynomial time, then P_2 cannot be solved in polynomial time.

Consider the partition problem. We are given n items $1, 2, \ldots, n$. Item j has size $s(j)$. The problem is to determine whether or not the set of items can be partitioned into two sets such that the sum of the size of the items in one set equals the sum of the size of the items in the other set. Now let us polynomially transform the partition problem to the decision version of the identical machines scheduling problem. Given any instance I of partition, we define the instance $f(I)$ as follows. There are n tasks and $m = 2$ machines. Task i represents item i and its processing time is $s(i)$. All the tasks are independent and $B = \sum_{i=1}^{i=n} s(i)/2$. Clearly, $f(I)$ has a schedule with maskespan B iff the instance I has a partition.

A decision problem is said to be *strongly NP-complete* if the problem is NP-complete even when all the "numbers" in the problem instance are less than or equal to $p(n)$, where p is a polynomial and n is the "size" of the problem instance. Partition is not NP-complete in the strong sense (under the assumption that $P \neq NP$) because there is a polynomial time dynamic programming algorithm to solve this problem when $\sum s(i) \leq p(n)$ (see Chapter 10). An excellent source for NP-completeness information is the book by Garey and Johnson [8].

1.3.3 Performance Evaluation of Algorithms

The main criterion used to compare approximation algorithms has been the quality of the solution generated. Let us consider different ways to compare the quality of the solutions generated when measuring the worst case. That is the main criterion discussed in Section 1.2.

For some problems, it is very hard to judge the quality of the solution generated. For example, approximating colors, can only be judged by viewing the resulting images and that is subjective (see Chapter 86). Chapter 85 covers digital reputation schemes. Here again, it is difficult to judge the quality of the solution generated. Problems in the application areas of bioinformatics and VLSI fall into this category because, in general, these are problems with multiobjective functions.

In what follows, we concentrate on problems where it is possible to judge the quality of the solution generated. At this point, we need to introduce additional notation. Let P be an optimization problem and let A be an algorithm that generates a feasible solution for every instance I of problem P. We use $\hat{f}_A(I)$ to denote the objective function value of the solution generated by algorithm A for instance I. We drop A and use $\hat{f}(I)$ when it is clear which algorithm is being used. Let $f^*(I)$ be the objective function value of an optimal solution for instance I. Note that normally we do not know the value of $f^*(I)$ exactly, but we have bounds that should be as tight as possible.

Let G be an undirected graph that represents a set of cities (vertices) and roads (edges) between a pair of cities. Every edge has a positive number called the weight (or cost) and represents the cost of driving (gas plus tolls) between the pair of cities it joins. A *shortest path* from vertex s to vertex t in G is an st-path (path from s to t) such that the sum of the weight of the edges in it is the "'least possible among all possible st-paths." There are well-known algorithms that solve this shortest-path problem in polynomial time [2]. Let A be an algorithm that generates a feasible solution (st-path) for every instance I of problem P. If for every instance I, algorithm A generates an st-path such that

$$\hat{f}(I) \leq f^*(I) + c$$

where c is some fixed constant, then A is said to be an *absolute* approximation algorithm for problem P with (additive) approximation bound c. Ideally, we would like to design a linear (or at least polynomial) time approximation algorithm with the smallest possible approximation bound. It is not difficult to see that this is not a good way of measuring the quality of a solution. Suppose that we have a graph G and we are running an absolute approximation algorithm for the shortest path problem concurrently in two different countries with the edge weight expressed in the local currency. Furthermore, assume that there is a large exchange rate between the two currencies. Any approximation algorithm solving the weak currency instance will have a much harder time finding a solution within the bound of c, than when solving the strong currency instance. We can take this to the extreme. We now claim that the above absolute approximation algorithm A can be used to generate an optimal solution for every problem instance within the same time complexity bound.

The argument is simple. Given any instance I of the shortest-path problem, we construct an instance I_{c+1} using the same graph, but every edge weight is multiplied by $c+1$. Clearly, $f^*(I_{c+1}) = (c+1)\, f^*(I)$. The st-path for I_{c+1} constructed by the algorithm is also an st-path in I with weight $\hat{f}(I) = \hat{f}(I_{c+1})/(c+1)$. Since $\hat{f}(I_{c+1}) \leq f^*(I_{c+1}) + c$, then by substituting the above bounds we know that

$$\hat{f}(I) = \frac{\hat{f}(I_{c+1})}{(c+1)} \leq \frac{f^*(I_{c+1})}{c+1} + \frac{c}{c+1} = f^*(I) + \frac{c}{c+1}$$

Since all the edges have integer weights, it then follows that the algorithm solves the problem optimally. In other words, for the shortest path problem any algorithm that generates a solution with (additive) approximation bound c can be used to generate an optimal solution within the same time complexity bound. This same property can be established for almost all NP-hard optimization problems. Because of this, the use of absolute approximation has never been given a serious consideration.

Sahni [14] defines as an ϵ-approximation algorithm for problem P an algorithm that generates a feasible solution for every problem instance I of P such that

$$\left|\frac{\hat{f}(I) - f^*(I)}{f^*(I)}\right| \leq \epsilon$$

It is assumed that $f^*(I) > 0$. For a minimization problem, $\epsilon > 0$ and for a maximization problem, $0 < \epsilon < 1$. In both cases, ϵ represents the percentage of error. The algorithm is called an ϵ-approximation

algorithm and the solution is said to be an ϵ-approximate solution. Graham's list scheduling algorithm [1] is a $1-1/n$-approximation algorithm, and Sahni and Gonzalez [16] algorithm for the k-maxcut problem is a $\frac{1}{k}$-approximation algorithm (see Section 1.2). Note that this notation is different from the one discussed in Section 1.2. The difference is 1 unit, i.e., the ϵ in this notation corresponds to $1+\epsilon$ in the other.

Johnson [12] used a slightly different, but equivalent notation. He uses the approximation ratio ρ to mean that for every problem instance I of P, the algorithm satisfies $\frac{\hat{f}(I)}{f^*(I)} \leq \rho$ for minimization problems, and $\frac{f^*(I)}{\hat{f}(I)} \leq \rho$ for maximization problems. The one for minimization problems is the same as the one given in Ref. [1]. The value for ρ is always greater than 1, and the closer to 1, the better the solution generated by the algorithm. One refers to ρ as the *approximation ratio* and the algorithm is a ρ-approximation algorithm. The list scheduling algorithm in the previous section is a $(2-\frac{1}{m})$-approximation algorithm and the algorithm for the k-maxcut problem is a $(\frac{k}{k-1})$-approximation algorithm. Sometimes, $1/\rho$ is used as the approximation ratio for maximization problems. Using this notation, the algorithm for the k-maxcut problem in the previous section is a $1-\frac{1}{k}$-approximation algorithm.

All the above forms are in use today. The most popular ones are ρ for minimization and $1/\rho$ for maximization. These are referred to as approximation ratios or approximation factors. We refer to all these algorithms as ϵ-approximation algorithms. The point to remember is that one needs to be aware of the differences and be alert when reading the literature. In the above discussion, we make ϵ and ρ look as if they are fixed constants. But, they can be made dependent on the size of the problem instance I. For example, it may be $\ln n$, or n^ϵ for some problems, where n is some parameter of the problem that depends on I, e.g., the number of nodes in the input graph, and ϵ depends on the algorithm being used to generate the solutions.

Normally, one prefers an algorithm with a smaller approximation ratio. However, it is not always the case that an algorithm with smaller approximation ratio always generates solutions closer to optimal than one with a larger approximation ratio. The main reason is that the notation is for the worst-case ratio and the worst case does not always occur. But there are other reasons too. For example, the bound for the optimal solution value used in the analysis of two different algorithms may be different. Let P be the shortest-path minimization problem and let A be an algorithm with approximation ratio 2. In this case, we use d as the lower bound for $f^*(I)$, where d is some parameter of the problem instance. Algorithm B is a 1.5-approximation algorithm, but $f^*(I)$ used to establish it is the exact optimal solution value. Suppose that for problem instance I the value of d is 5 and $f^*(I) = 8$. Algorithm A will generate a path with weight at most 10, whereas algorithm B will generate one with weight at most $1.5 \times 8 = 12$. So the solution generated by Algorithm B may be worse than the one generated by A even if both algorithms generate the worst values for the instance. One can argue that the average "error" makes more sense than worst case. The problem is how to define and establish bounds for average "error." There are many other pitfalls when using worst-case ratios. It is important to keep all this in mind when making comparisons between algorithms. In practice, one may run several different approximation algorithms concurrently and output the best of the solutions. This has the disadvantage that the running time of this compound algorithm will be the one for the slowest algorithm.

There are a few problems for which the worst-case approximation ratio applies only to problem instances where the value of the optimal solution is small. One such problem is the bin packing problem discussed in Section 1.2. Informally, ρ_A^∞ is the smallest constant such that there exists a constant $K < \infty$ for which

$$\hat{f}(I) \leq \rho_A^\infty f^*(I) + K$$

The *asymptotic approximation ratio* is the multiplicative constant and it hides the additive constant K. This is most useful when K is small. Chapter 32 discusses this notation formally. The asymptotic notation is mainly used for bin packing and some of its variants.

Ausiello et al. [30] introduced the *differential ratio*. Informally, an algorithm is said to be a δ differential ratio approximation algorithm if for every instance I of P

$$\frac{\omega(I) - \hat{f}(I)}{\omega(I) - f^*(I)} \leq \delta$$

where $\omega(I)$ is the value of a worst solution for instance I. Differential ratio has some interesting properties for the complexity of the approximation problems. Chapter 16 discusses differential ratio approximation and its variations.

As said earlier, there are many different criteria to compare algorithms. What if we use both the approximation ratio and time complexity? For example, the approximation algorithms in Ref. [15] and the one in Ref. [29] are current Pareto optimal with respect to these criteria for the TSP defined over metric graphs. Neither of the algorithms dominates the others in both time complexity and approximation ratio. The same can be said about the simple linear time approximation algorithm for the k-maxcut problem in Ref. [16] and the complex one given in Ref. [52] or the more recent ones that apply for all k.

The best algorithm to use also depends on the instance being solved. It makes a difference whether we are dealing with an instance of the TSP with optimal tour cost equal to a billion dollars and one with optimal cost equal to just a few pennies. Though, it also depends on the number of such instances being solved.

More elaborate approximation algorithms have been developed that generate a solution for any fixed constant ϵ. Formally, a PTAS for problem P is an algorithm A that given any fixed constant $\epsilon > 0$, it constructs a solution to problem P such that $|\frac{\hat{f}(I) - f^*(I)}{f^*(I)}| \leq \epsilon$ in polynomial time with respect to the length of the instance I. Note that the time complexity may be exponential with respect to $1/\epsilon$. For example, the time complexity could be $O(n^{(1/\epsilon)})$ or $O(n + 4^{O(1/\epsilon)})$. Equivalent PTAS are also defined using different notation, for example, based on $\frac{\hat{f}(I)}{f^*(I)} \leq 1 + \epsilon$ for minimization problems.

One would like to design PTAS for all problems, but that is not possible unless $P = NP$. Clearly, with respect to approximation ratios, the PTAS is better than the ϵ-approximation algorithms for some ϵ. But their main drawback is that they are not practical because the time complexity is exponential on $1/\epsilon$. This does not preclude the existence of a practical PTAS for "natural" occurring problems. However, a PTAS establishes that a problem can be approximated for all fixed constants. Different types of PTAS are discussed in Chapter 9. Additional PTAS are presented in Chapters 42, 45, and 51.

A PTAS is said to be an FPTAS if its time complexity is polynomial with respect to n (the problem size) and $1/\epsilon$. FPTAS are for the most part practical algorithms. Different methodologies for designing FPTAS are discussed in Chapter 10.

Approximation schemes based on asymptotic approximation and on randomized algorithms have been developed. Chapters 11 and 45 discuss asymptotic approximation schemes and Chapter 12 discusses randomized approximation schemes.

References

[1] Graham, R. L., Bounds for certain multiprocessing anomalies, *Bell System Tech. J.*, 45, 1563, 1966.

[2] Sahni, S., *Data Structures, Algorithms, and Applications in C++*, 2nd ed., Silicon Press, Summit, NJ, 2005.

[3] Gilbert, E. N. and Pollak, H. O., Steiner minimal trees, *SIAM J. Appl. Math.*, 16(1), 1, 1968.

[4] Graham, R. L., Bounds on multiprocessing timing anomalies, *SIAM J. Appl. Math.,* 17, 263, 1969.

[5] Leung, J. Y.-T., Ed., *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, Chapman & Hall/CRC, Boca Raton, FL, 2004.

[6] Cook, S. A., The complexity of theorem-proving procedures, *Proc. STOC'71*, 1971, p. 151.

[7] Karp, R. M., Reducibility among combinatorial problems, in R. E. Miller and J. W. Thatcher, eds., *Complexity of Computer Computations*, Plenum Press, New York, 1972, p. 85.

[8] Garey, M. R. and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, NY, 1979.

[9] Garey, M. R., Graham, R. L., and Ullman, J. D., Worst-case analysis of memory allocation algorithms, *Proc. STOC*, ACM, 1972, p. 143.

[10] Johnson, D. S., Near-Optimal Bin Packing Algorithms, Ph.D. thesis, Massachusetts Institute of Technology, Department of Mathematics, Cambridge, 1973.

[11] Johnson, D. S., Fast algorithms for bin packing, *JCSS*, 8, 272, 1974.

[12] Johnson, D. S., Approximation algorithms for combinatorial problems, *JCSS*, 9, 256, 1974.
[13] Sahni, S., On the Knapsack and Other Computationally Related Problems, Ph.D. thesis, Cornell University, 1973.
[14] Sahni, S., Approximate algorithms for the 0/1 knapsack problem, *JACM*, 22(1), 115, 1975.
[15] Rosenkrantz, R., Stearns, R., and Lewis, L., An analysis of several heuristics for the traveling salesman problem, *SIAM J. Comput.*, 6(3), 563, 1977.
[16] Sahni, S. and Gonzalez, T., P-complete approximation problems, *JACM*, 23, 555, 1976.
[17] Gens, G. V. and Levner, E., Complexity of approximation algorithms for combinatorial problems: A survey, *SIGACT News*, 12(3), 52, 1980.
[18] Levner, E. and Gens, G. V., *Discrete Optimization Problems and Efficient Approximation Algorithms,* Central Economic and Mathematics Institute, Moscow, 1978 (in Russian).
[19] Garey, M. R. and Johnson, D. S., The complexity of near-optimal graph coloring, *SIAM J. Comput.*, 4, 397, 1975.
[20] Ibarra, O. and Kim, C., Fast approximation algorithms for the knapsack and sum of subset problems, *JACM*, 22(4), 463, 1975.
[21] Sahni, S., Algorithms for scheduling independent tasks, *JACM*, 23(1), 116, 1976.
[22] Horowitz, E. and Sahni, S., Exact and approximate algorithms for scheduling nonidentical processors, *JACM*, 23(2), 317, 1976.
[23] Babat, L. G., Approximate computation of linear functions on vertices of the unit N-dimensional cube, in *Studies in Discrete Optimization,* Fridman, A. A., Ed., Nauka, Moscow, 1976 (in Russian).
[24] Babat, L. G., A fixed-charge problem, *Izv. Akad. Nauk SSR, Techn, Kibernet.*, 3, 25, 1978 (in Russian).
[25] Lawler, E., Fast approximation algorithms for knapsack problems, *Math. Oper. Res.*, 4, 339, 1979.
[26] Garey, M. R. and Johnson, D. S., Strong NP-completeness results: Motivations, examples, and implications, *JACM*, 25, 499, 1978.
[27] Lin, S. and Kernighan, B. W., An effective heuristic algorithm for the traveling salesman problem, *Oper. Res.*, 21(2), 498, 1973.
[28] Papadimitriou, C. H. and Steiglitz, K., On the complexity of local search for the traveling salesman problem, *SIAM J. Comput.*, 6, 76, 1977.
[29] Christofides, N., Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem. Technical Report 338, Grad School of Industrial Administration, CMU, 1976.
[30] Ausiello, G., D'Atri, A., and Protasi, M., On the structure of combinatorial problems and structure preserving reductions, in *Proc. ICALP'77*, Lecture Notes in Computer Science, Vol. 52 Springer, Berlin, 1977, p. 45.
[31] Cornuejols, G., Fisher, M. L., and Nemhauser, G. L., Location of bank accounts to optimize float: An analytic study of exact and approximate algorithms, *Manage. Sci.*, 23(8), 789, 1977.
[32] Khachiyan, L. G., A polynomial algorithms for the linear programming problem, *Dokl. Akad. Nauk SSSR*, 244(5), 1979 (in Russian).
[33] Karmakar, N., A new polynomial-time algorithm for linear programming, *Combinatorica,* 4, 373, 1984.
[34] Grötschel, M., Lovász, L., and Schrijver, A., The ellipsoid method and its consequences in combinatorial optimization, *Combinatorica,* 1, 169, 1981.
[35] Schrijver, A., *Theory of Linear and Integer Programming*, Wiley-Interscience Series in Discrete Mathematics and Optimization, Wiley, New York, 2000.
[36] Vanderbei, R. J., *Linear Programming Foundations and Extensions,* Series: International Series in Operations Research & Management Science, Vol. 37, Springer, Berlin.
[37] Lovász, L., On the ratio of optimal integral and fractional covers, *Discrete Math.*, 13, 383, 1975.
[38] Chvátal, V., A greedy heuristic for the set-covering problem, *Math. Oper. Res.*, 4(3), 233, 1979.
[39] Hochbaum, D. S., Approximation algorithms for set covering and vertex covering problems, *SIAM J. Comput.*, 11, 555, 1982.
[40] Bar-Yehuda, R. and Even, S., A linear time approximation algorithm for the weighted vertex cover problem, *J. Algorithms*, 2, 198, 1981.

[41] Bar-Yehuda, R. and Even, S., A local-ratio theorem for approximating the weighted set cover problem, *Ann. of Disc. Math.*, 25, 27, 1985.
[42] Bar-Yehuda, R. and Bendel, K., Local ratio: A unified framework for approximation algorithms, *ACM Comput. Surv.*, 36(4), 422, 2004.
[43] Raghavan, R. and Thompson, C., Randomized rounding: A technique for provably good algorithms and algorithmic proof, *Combinatorica*, 7, 365, 1987.
[44] Fernandez de la Vega, W. and Lueker, G. S., Bin packing can be solved within $1 + \epsilon$ in linear time, *Combinatorica,* 1, 349, 1981.
[45] Karmakar, N. and Karp, R. M., An efficient approximation scheme for the one-dimensional bin packing problem, *Proc. FOCS*, 1982, p. 312.
[46] Papadimitriou, C. H. and Yannakakis, M., Optimization, approximation and complexity classes, *J. Comput. Syst. Sci.*, 43, 425, 1991.
[47] Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M., Proof verification and hardness of approximation problems, *Proc. FOCS*, 1992.
[48] Feige, U., Goldwasser, S., Lovasz, L., Safra, S., and Szegedy, M., Interactive proofs and the hardness of approximating cliques, *JACM*, 43, 1996.
[49] Feige, U., A threshold of ln n for approximating set cover, *JACM*, 45(4), 634, 1998. (Prelim. version in STOC'96.)
[50] Engebretsen, L. and Holmerin, J., Towards optimal lower bounds for clique and chromatic number, *TCS*, 299, 2003.
[51] Hastad, J., Some optimal inapproximability results, *JACM*, 48, 2001. (Prelim. version in STOC'97.)
[52] Goemans, M. X. and Williamson, D. P., Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming, *JACM*, 42(6), 1115, 1995.
[53] Goemans, M. X. and Williamson, D. P., A general approximation technique for constrained forest problems, *SIAM J. Comput.*, 24(2), 296, 1995.
[54] Jain, K., Mahdian, M., Markakis, E., Saberi, A., and Vazirani, V. V., Approximation algorithms for facility location via dual fitting with factor-revealing LP, *JACM*, 50, 795, 2003.
[55] Glover, F., Future paths for integer programming and links to artificial intelligence, *Comput. Oper. Res.*, 13, 533, 1986.
[56] Kirkpatrick, S., Gelatt, C. D., Jr. and Vecchi, M. P., Optimization by simulated annealing, *Science,* 220, 671, 1983.
[57] Černý, V., Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm, *J. Optimization Theory Appl.*, 45, 41, 1985.
[58] Fogel, L. J., Toward inductive inference automata, in *Proc. Int. Fed. Inf. Process. Congr.*, 1962, 395.
[59] Fogel, L. J., Owens, A. J., and Walsh, M. J., *Artificial Intelligence through Simulated Evolution*, Wiley, New York, 1966.
[60] Rechenberg, I., *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, Frommann-Holzboog, Stuttgart, 1973.
[61] Holland, J. H., *Adaption in Natural and Artificial Systems*, The University of Michigan Press, Ann Harbor, MI, 1975.
[62] Dorigo, M., Maniezzo, V., and Colorni, A., Positive Feedback as a Search Strategy, Technical Report 91-016, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1991.
[63] Moscato, P., On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms, Report 826, California Institute of Technology, 1989.