Chapter 5

Which Problems Are Not Tractable?

We have seen a number of optimization problems. Some of them are relatively simple, such as Minimum Spanning Tree. Solving each of these optimization problems in general requires a single (maybe smart) idea which can be implemented by an efficient algorithm of a couple of dozens of lines. Some other optimization problems, on the other hand, are much more nontrivial. Examples of this kind of optimization problems we have seen include Maximum Flow, Graph Matching, and Linear Programming. Solving each of these harder problems efficiently requires deep understanding and thorough analysis on structures and properties of the problem. A polynomial time algorithm for the problem is derived based on such a highly nontrivial structural investigation plus maybe a number of subtle algorithmic techniques. Moreover, it seems each of these problems requires a different set of techniques and there is no powerful *universal techniques* that can be applied to all of these problems.

This makes the task of solving an optimization problem very unpredictable. Suppose that you have an optimization problem and want to develop an efficient algorithm for it. If you are lucky and the problem is relatively easy, then you solve the problem in a couple of days, or in a couple of weeks. If the problem is as hard as, for example, the Linear Programming problem, but you work very hard and are also lucky enough to find a correct approach, you *may be* able to develop an efficient algorithm for the problem in several months or even in several years. Now what if all above are not the case: you work hard, you are smart, but the problem still remains unsolved after your enormous effort? You may start suspecting whether there even

exists an efficient algorithm at all for you problem. Therefore, you may start trying a proof to show that your problem is intrinsically difficult.

However, you may quickly realize that proving the problem's intrinsic difficulty is just as hard as, or even harder than, finding an efficient algorithm for the problem — there are simply very few known techniques available for proving the intrinsic difficulties for optimization problems. For example, suppose that your problem is the TRAVELING SALESMAN problem, for which no body has been able to develop an efficient algorithm. Experts would tell you that also nobody *in the world* has been able to prove that TRAVELING SALESMAN is even harder than MINIMUM SPANNING TREE.

Fortunately, an extremely useful system, the NP-hardness theory, has been developed. Although this system does not provide you with a formal proof that your problem is hard, it provides a *strong evidence* that your problem is hard. Essentially, the NP-hardness theory has collected several hundred problems that people believe to be hard, and provides systematic techniques to let you show that your own problem also belongs to this category so it is not easier than *any* of these hundreds of hard problems. Therefore, not just you cannot develop an efficient algorithm for the problem, *nobody in the world* so far can develop such an algorithm, either.

In this chapter, we formally introduce the concept of NP-hardness for optimization problems. We provide enough evidence to show that if an optimization problem is NP-hard, then it should be very hard. General techniques for proving NP-hardness for optimization problems are introduced with concrete examples.

Proving the NP-hardness of an optimization problem is just the beginning of work on the problem. It provides useful information that shows solving the problem precisely is a very ambitious, maybe too ambitious, attempt. However, this does not obviate our need for solving the problem if the problem is of practical importance. Therefore, approximation algorithms for NP-hard optimization problems have been naturally introduced. In the last section of this chapter, we will formally introduce the concept of approximation algorithms and the measures for evaluation of approximation algorithms. The rest of this book will be concentrating on the study of approximation algorithms for NP-hard optimization problems.

5.1 NP-hard optimization problems

Recall that a decision problem Q is NP-hard if every problem in the class NP is polynomial-time many-one reducible to Q. Therefore, if an NP-hard

decision problem Q can be solved in polynomial time, then all problems in NP are solvable in polynomial time, thus P = NP. According to our working conjecture that P $\neq$ NP, which is commonly believed, the NP-hardness of a problem Q is a strong evidence that the problem Q cannot be solved in polynomial time.

The polynomial-time reductions and the NP-hardness can be extended to optimization problems, as given by the following discussions.

Definition 5.1.1 An decision problem D is *polynomial-time reducible* to an optimization problem $Q = (I_Q, S_Q, f_Q, opt_Q)$ if there are two polynomial-time computable functions h and g such that
(1) for an instance x of the decision problem D, $h(x)$ is an instance of the optimization problem Q, and
(2) given an optimal solution y to the instance $h(x)$ of Q, $g(x, h(x), y) = 1$ if and only if x is a yes-instance for the decision problem D.

As an example, we show that the decision problem Partition is polynomial time reducible to the optimization problem c-Makespan with $c \geq 2$, which is a restricted version of the Makespan problem.

Recall that the Partition problem is defined as follows.

> Partition
> Given a set of integers $S = \{a_1, a_2, \ldots, a_n\}$, can the set S be partitioned into two disjoint sets S_1 and S_2 of equal size, that is, $S = S_1 \cup S_2$, $S_1 \cap S_2 = \emptyset$, and $\sum_{a_i \in S_1} a_i = \sum_{a_j \in S_2} a_j$?

For a given positive integer c, the c-Makespan problem is defined by

> c-Makespan $= (I_Q, S_Q, f_Q, opt_Q)$
> I_Q: the set of tuples $T = \{t_1, \ldots, t_n\}$, where t_i is an integer that is the processing time for the i-th job
> S_Q: $S_Q(T)$ is the set of partitions $P = (T_1, \ldots, T_c)$ of the integers $\{t_1, \ldots, t_n\}$ in T into c disjoint parts
> f_Q: $f_Q(T, P) = \max_i\{\sum_{t_j \in T_i} t_j\}$
> opt_Q: min

Therefore, the c-Makespan problem is the Makespan problem in which the number of processors is a fixed constant c.

Lemma 5.1.1 *The* Partition *problem is polynomial-time reducible to the* c-Makespan *problem, for any integer* $c \geq 2$.

PROOF. The functions h and g for the polynomial-time reduction from PARTITION to c-MAKESPAN are described as follows.

Let $\alpha = \langle x_1, \ldots, x_n \rangle$ be an instance of PARTITION. We define $h(\alpha) = \langle t_1, \ldots, t_n, t_{n+1}, \ldots, t_{n+c-2} \rangle$, where $t_i = x_i$ for $1 \leq i \leq n$, and $t_{n+1} = \cdots = t_{n+c-2} = \lceil (\sum_{i=1}^{n} x_i)/2 \rceil$. Clearly, $h(\alpha)$ is an instance of c-MAKESPAN and can be constructed from α in polynomial time.

Now for any optimal solution $P = (T_1, \ldots, T_c)$ to the instance $h(\alpha)$ for the c-MAKESPAN problem, the function $g(\alpha, h(\alpha), P) = 1$ if and only if

$$\max_i \{ \sum_{t_j \in T_i} t_j \} = (\sum_{i=1}^{n} x_i)/2.$$

It is easy to see that if α is a yes-instance for the PARTITION problem, then every optimal schedule P on $h(\alpha)$ splits the numbers x_1, ..., x_n into two sets S_1 and S_2 of equal size $(\sum_{i=1}^{n} x_i)/2$, assigns each of the sets to a processor, and assigns each of the jobs of time t_j, $j = n+1, \ldots, n+c-2$, to a distinct processor. The schedule P has makespan $(\sum_{i=1}^{n} x_i)/2 = \lceil (\sum_{i=1}^{n} x_i)/2 \rceil$. On the other hand, if α is a no-instance for the PARTITION problem, then the schedule P on $h(\alpha)$ always has makespan larger than $(\sum_{i=1}^{n} x_i)/2$. In particular, if $\sum_{i=1}^{n} x_i$ is an odd number, then any schedule on $h(\alpha)$ has makespan at least $\lceil (\sum_{i=1}^{n} x_i)/2 \rceil > (\sum_{i=1}^{n} x_i)/2$.

Therefore, the function value $g(\alpha, h(\alpha), P) = 1$ if and only if α is a yes-instance for the PARTITION problem. Moreover, the function g is clearly computable in polynomial time. □

A polynomial-time reduction from a decision problem D to an optimization problem Q implies that the problem D cannot be much harder than the problem Q, in the following sense.

Lemma 5.1.2 *Suppose that a decision problem D is polynomial-time reducible to an optimization problem Q. If Q is solvable in polynomial time, then so is D.*

PROOF. Let h and g be the two polynomial-time computable functions for the reduction from D to Q. Let $\mathcal{A}$ be a polynomial time algorithm that solves the optimization problem Q. Now a polynomial-time algorithm for the decision problem D can be easily derived as follows: given an instance x for D, we first construct the instance $h(x)$ for Q; then apply the algorithm $\mathcal{A}$ to find an optimal solution y for $h(x)$; now x is a yes-instance for

D if and only if $g(x, h(x), y) = 1$. By our assumption, all $h(x)$, y, and $g(x, h(x), y)$ are polynomial-time computable (in particular note that since $h(x)$ is computable in polynomial time , the length $|h(x)|$ of $h(x)$ is bounded by a polynomial of $|x|$, and that since $\mathcal{A}$ runs in polynomial time, the length $|y|$ of y is bounded by a polynomial of $|h(x)|$ thus by a polynomial of $|x|$). Thus, this algorithm runs in polynomial time and correctly decides if x is a yes-instance for the decision problem D. □

The polynomial-time reduction from decision problems to optimization problems extends the concept of NP-hardness to optimization problems.

Definition 5.1.2 An optimization problem Q is NP-*hard* if there is an NP-hard decision problem D that is polynomial-time reducible to Q.

Let Q be an NP-hard optimization problem such that an NP-hard decision problem D is polynomial-time reducible to Q. If Q is solvable in polynomial time, then by Lemma 5.1.2, the NP-hard decision problem Q is solvable in polynomial time, which implies consequently, by Definition 1.4.5 and Lemma 1.4.1, that P = NP, violating our Working Conjecture in NP-completeness Theory (see Section 1.4). Therefor, the NP-hardness of an optimization problem Q provides a very strong evidence that the problem Q is intractable, i.e., not solvable in polynomial time.

Since the PARTITION problem is known to be NP-hard, Lemma 5.1.1 gives immediately

Theorem 5.1.3 *The* c-MAKESPAN *problem is an* NP-*hard optimization problem for any integer* $c \geq 2$.

Many NP-hard decision problems originate from optimization problems. Therefore, the polynomial-time reductions from these decision problems to the corresponding optimization problems are straightforward. Consequently, the NP-hardness of these optimization problems follow directly from the NP-hardness of the corresponding decision problems. For example, the NP-hardness for the decision versions of the problems TRAVELING SALESMAN, GRAPH COLORING, PLANAR GRAPH INDEP-SET, and PLANAR GRAPH VERTEX-COVER (see Section 1.4) implies directly the NP-hardness for the optimization versions of the same problems (see Appendix D for precise definitions), respectively.

We give another example for NP-hard optimization problems, whose NP-hardness is from a not so obvious polynomial time reduction. Suppose that in the LINEAR PROGRAMMING problem, we require that we work only on

the domain of integer numbers, then we get the INTEGER LINEAR PROGRAMMING problem, or for short the INTEGER LP problem. More formally, each instance of the INTEGER LP problem is a triple $\alpha = \mathbf{(b,\ c,\ A)}$, where for some integers n and m, $\mathbf{b}$ is an m-dimensional vector of integer numbers, $\mathbf{c}$ is an n-dimensional vector of integer numbers, and $\mathbf{A}$ is an $m \times n$ matrix of integer numbers. A solution $\mathbf{x}$ to the instance α is an n-dimensional vector of integer numbers such that $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$, and a solution $\mathbf{x}$ is optimal if it minimizes the inner product $\mathbf{c}^T\mathbf{x}$. This gives the *standard form* for the INTEGER LP problem.

It might seem that the INTEGER LP problem is easier than the general LINEAR PROGRAMMING problem since we are working on simpler numbers. This intuition is, however, not true. In fact, the INTEGER LP problem is computationally much harder than the general LINEAR PROGRAMMING problem. This may be seen from the following fact: the set of solutions to an instance $\alpha = \mathbf{(b,\ c,\ A)}$ of the INTEGER LP problem, defined by the constraints $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$, is no longer a convex set in the n-dimensional Euclidean space $\mathcal{E}^n$. It instead consists of discrete points in $\mathcal{E}^n$. Therefore, greedy algorithms based on local search, such as the simplex method, do not seem to work any more.

The hardness of the INTEGER LP problem is formally given as follows.

Theorem 5.1.4 *The optimization problem* INTEGER LP *is* NP*-hard.*

PROOF. We show that the well known NP-complete problem, the SATISFIABILITY problem, is polynomial-time reducible to INTEGER LP.

Formally, an instance α of the SATISFIABILITY problem is given by a Boolean expression in conjunctive normal form (CNF):

$$\alpha = C_1 \wedge C_2 \wedge ... \wedge C_m \tag{5.1}$$

where each C_i (called a *clause*) is an OR of Boolean literals. The question is whether there is a Boolean assignment to the Boolean variables x_1, x_2, ..., x_n in α that makes the expression TRUE.

We show how a polynomial-time computable function h converts the instance α in (5.1) of the SATISFIABILITY problem into an instance $h(\alpha)$ for the INTEGER LP problem.

Suppose that the clause C_i in α is

$$C_i = (x_{i_1} \vee \cdots \vee x_{i_s} \vee \overline{x}_{j_1} \vee \cdots \vee \overline{x}_{j_t})$$

We then construct a linear constraint

$$x_{i_1} + \cdots + x_{i_s} + (1 - x_{j_1}) + \cdots + (1 - x_{j_t}) \geq z \tag{5.2}$$

where z is a new variable. Moreover, for each Boolean variable x_j in α, we have the constraints

$$x_j \geq 0 \quad \text{and} \quad x_j \leq 1 \tag{5.3}$$

Thus, the integer variables x_j can take only the values 0 and 1. We let $x_j = 1$ simulate the assignment $x_j = \text{TRUE}$ and let $x_j = 0$ simulate the assignment $x_j = \text{FALSE}$. Therefore, the clause C_i is TRUE under a TRUE-FALSE assignment to the Boolean variables $x_1, \ldots, x_n$ if and only if

$$x_{i_1} + \cdots + x_{i_s} + (1 - x_{j_1}) + \cdots + (1 - x_{j_t}) \geq 1$$

under the corresponding 1-0 assignment to the integer variables $x_1, \cdots, x_n$.

Finally, our objective function is to maximize the variable value z.

So our instance for the INTEGER LP problem consists of the constraints (5.2) corresponding to all clauses C_i in α and all constraints in (5.3). Let this instance be β_α.[1] Now we define a function h such that given an instance α in (5.1) for the SATISFIABILITY problem, $h(\alpha) = \beta_\alpha$, where β_α is the instance constructed as above for the INTEGER LP problem. It is clear that the function h is computable in polynomial time.

Now note that if an optimal solution $\mathbf{x}$ to β_α, which is a 1-0 assignment to the variables $x_1, \ldots, x_n$, makes the objective function have value $z > 0$, then we have (note that z is an integer)

$$x_{i_1} + \cdots + x_{i_s} + (1 - x_{j_1}) + \cdots + (1 - x_{j_t}) \geq z \geq 1$$

for all linear constraints corresponding to the clauses of the instance α. In consequence, the corresponding TRUE-FALSE assignment to the Boolean variables $x_1, \ldots, x_n$ makes all clauses in α TRUE. That is, the instance α is a yes-instance for the SATISFIABILITY problem. On the other hand, if the optimal solution to β_α has objective function value $z \leq 0$, then no 1-0 assignment to $x_1, \ldots, x_n$ can make all linear constraints satisfy

$$x_{i_1} + \cdots + x_{i_s} + (1 - x_{j_1}) + \cdots + (1 - x_{j_t}) \geq 1$$

That is, no TRUE-FALSE assignment to $x_1, \ldots, x_n$ can satisfy all clauses in α. In other words, α is a no-instance to the SATISFIABILITY problem. Therefore, with the instances α and β_α and an optimal solution to β_α, it can be trivially decided whether α is a yes-instance for SATISFIABILITY.

[1] To follow the definitions strictly, we should also convert β_α into the standard form. However, since the discussion based on β_α is more convenient and the translation of β_α to the standard form is straightforward, we assume that our instance for the INTEGER LP problem is just β_α.

This proves that the NP-complete problem SATISFIABILITY is polynomial time reducible to the INTEGER LP problem. Consequently, the INTEGER LP problem is NP-hard. □

The general LINEAR PROGRAMMING problem can be solved in polynomial time [79]. Theorem 5.1.4 shows that the INTEGER LP problem is much harder than the general LINEAR PROGRAMMING problem. Our later study will show that INTEGER LP is actually one of the hardest NP-optimization problems.

The NP-hardness of an optimization problem can also be derived from the NP-hardness of other optimization problems. For this, we first need to introduce a new reduction.

Definition 5.1.3 An optimization problem Q_1 is *polynomial-time reducible* (or *p-reducible* for short) to an optimization problem Q_2 if there are two polynomial-time computable functions χ (the *instance function*) and ψ (the *solution function*) such that

(1) for any instance x_1 of Q_1, $\chi(x_1)$ is an instance of Q_2; and
(2) for any solution y_2 to the instance $\chi(x_1)$, $\psi(x_1, \chi(x_1), y_2)$ is a solution to x_1 such that y_2 is an optimal solution to $\chi(x_1)$ if and only if $\psi(x_1, \chi(x_1), y_2)$ is an optimal solution to x_1.

The following theorem follows directly from the definition.

Lemma 5.1.5 *If an optimization problem Q_1 is p-reducible to an optimization problem Q_2, and if Q_2 is solvable in polynomial time, then so is Q_1.*

PROOF. Suppose that Q_1 is p-reducible to Q_2 via the instance function χ and the solution function ψ, both computable in polynomial time. Then an optimal solution to an instance x of Q_1 can be obtained from $\psi(x, \chi(x), y_2)$, where y_2 is an optimal solution to $\chi(x)$ and is supposed to be constructible in polynomial time from the instance $\chi(x)$. □

Lemma 5.1.6 *Suppose that an optimization problem Q_1 is p-reducible to an optimization problem Q_2. If Q_1 is NP-hard, then so is Q_2.*

PROOF. Suppose that Q_1 is p-reducible to Q_2 via the instance function χ and the solution function ψ, both computable in polynomial time. Since Q_1 is NP-hard, there is an NP-hard decision problem D that is polynomial-time reducible to Q_1, via two polynomial-time computable functions h and g (see Definition 5.1.1). Define two new functions h_1 and g_1 as follows: for

any instance x of D, $h_1(x) = \chi(h(x))$; and for any solution y to $h_1(x)$, $g_1(x, h_1(x), y) = g(x, h(x), \psi(h(x), h_1(x), y))$. It is not hard to verify by the definitions that for any instance x of D, $h_1(x)$ is an instance of Q_2, and $g_1(x, h_1(x), y) = 1$ if and only if y is an optimal solution to Q_2 and x is a yes-instance of D. Moreover, the functions h_1 and g_1 are clearly polynomial-time computable.

This proves that the NP-hard decision problem D is polynomial-time reducible to the optimization problem Q_2. Consequently, the optimization problem Q_2 is NP-hard. □

As another example, we show that the KNAPSACK problem is NP-hard. The KNAPSACK problem is formally defined as follows.

$$\begin{aligned}
&\text{KNAPSACK} = (I_Q, S_Q, f_Q, opt_Q)\\
&I_Q = \{\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle \mid s_i, v_j, B \text{ are integers}\}\\
&S_Q(\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle) = \{S \subseteq \{1, \ldots, n\} \mid \textstyle\sum_{i\in S} s_i \le B\}\\
&f_Q(\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle, S) = \textstyle\sum_{i\in S} v_i\\
&opt_Q = \max
\end{aligned}$$

An "application" of the KNAPSACK problem can be described as follows. A thief robbing a store finds n items. The i-th item is worth v_i dollars and weighs s_i pounds. The thief wants to take as valuable a load as possible, but he can carry at most B pounds in his knapsack. Now the thief wants to decide what items he should take. Fortunately, the problem is NP-hard, as we prove in the following theorem.

Theorem 5.1.7 *The* KNAPSACK *problem is* NP*-hard.*

PROOF. By Theorem 5.1.3, the 2-MAKESPAN problem is NP-hard. Thus, by Lemma 5.1.6, it suffices to show that 2-MAKESPAN is p-reducible to KNAPSACK. The instance function χ and the solution function ψ are described as follows.

Given an instance $\alpha = \langle t_1, \ldots, t_n\rangle$ of 2-MAKESPAN, $\chi(\alpha)$ is the instance $\chi(\alpha) = \langle t_1, \ldots, t_n; t_1, \ldots, t_n; B\rangle$ of KNAPSACK, where $B = \lceil \sum_{i=1}^{n} t_i/2 \rceil$. Given any solution S to $\chi(\alpha)$, which is a subset of $\{t_1, \ldots, t_n\}$ satisfying $\sum_{t_j\in S} t_j \le B$, the value of $\psi(\alpha, \chi(\alpha), S)$ is the partition $(S, \{t_1, \ldots, t_n\} - S)$ of the set $\{t_1, \ldots, t_n\}$, which assigns all the jobs in S to Processor-1, and all other jobs to Processor-2. Since an optimal solution S to $\chi(\alpha)$ is a subset of $\{t_1, \ldots, t_n\}$ that maximizes the value $\sum_{t_j\in S} t_j$ subject to the constraint $\sum_{t_j\in S} t_j \le \lceil \sum_{i=1}^{n} t_i/2 \rceil$, the solution S must give the "most even" splitting

$(S, \{t_1, \ldots, t_n\} - S)$ for the set $\{t_1, \ldots, t_n\}$. Therefore, S is an optimal solution to the instance $\chi(\alpha)$ of KNAPSACK if and only if $(S, \{t_1, \ldots, t_n\} - S)$ is an optimal solution to the instance α of 2-MAKESPAN. Moreover, the instance function χ and the solution function ψ are clearly computable in polynomial time. This completes the proof. □

Some optimization problems have subproblems that are of independent interest. Moreover, sometimes the complexity of a subproblem may help the study of the complexity of the original problem.

Definition 5.1.4 *Let $Q = (I_Q, S_Q, f_Q, opt_Q)$ be an optimization problem. An optimization problem Q' is a subproblem of Q if $Q' = (I'_Q, S_Q, f_Q, opt_Q)$, where $I'_Q \subseteq I_Q$.*

Note that for an optimization problem Q' to be a subproblem of another optimization problem Q, we not only require that the instance set I'_Q of Q' be a subset of the instance set I_Q of Q, but also that the solution set function S_Q, the objective function f_Q, and the optimization type opt_Q be all identical for both problems. These requirements are important when we study the computational complexity of a problem and its subproblems. For example, every instance of the INTEGER LP problem is an instance of the LINEAR PROGRAMMING problem. However, the INTEGER LP problem is *not* a subproblem of the LINEAR PROGRAMMING problem since for each instance α of the INTEGER LP problem, the solution set for α as an instance for the INTEGER LP problem is *not* identical to the solution set for α as an instance for the LINEAR PROGRAMMING problem.

Theorem 5.1.8 *Let Q be an optimization problem and Q' be a subproblem of Q. If the subproblem Q' is* NP*-hard, then so is the problem Q.*

PROOF. Since the subproblem Q' is NP-hard, there is an NP-hard decision problem D that is polynomial-time reducible to the optimization problem Q' via polynomial-time computable functions h and g. It is straightforward to verify that the functions h and g also serve for a polynomial time reduction from the NP-hard decision problem D to the optimization problem Q. Thus, the optimization problem Q is also NP-hard. □

For example, consider the PLANAR GRAPH INDEP-SET problem (given a planar graph G, find the largest subset S of vertices in G such that no two vertices in S are adjacent) and the INDEPENDENT SET problem (given a graph G, find the largest subset S of vertices in G such that no two vertices

in S are adjacent). Clearly, PLANAR GRAPH INDEP-SET is a subproblem of INDEPENDENT SET. Since PLANAR GRAPH INDEP-SET is NP-hard (see the remark following Theorem 5.1.3), we conclude that INDEPENDENT SET is also NP-hard. Similarly, from the NP-hardness of PLANAR GRAPH VERTEX-COVER (given a planar graph G, find a minimum set S of vertices such that every edge in G has at least one end in S), we derive the NP-hardness for the VERTEX COVER problem (given a graph G, find a minimum set S of vertices such that every edge in G has at least one end in S).

Corollary 5.1.9 *The* INDEPENDENT SET *problem and the* VERTEX COVER *problem are* NP*-hard.*

5.2 Polynomial time approximation

We have established a powerful system, the NP-hardness theory, by which we can show that a large number of optimization problems are computationally intractable, based on our believing that $\mathsf{P} \neq \mathsf{NP}$. However, this does not obviate the need for solving these hard problems — they are of obvious practical importance. Knowing the computational difficulty of the problems, one possible approach is that we could relax the requirement that we always find the optimal solution. In practice, a near-optimal solution will work fine in many cases. Of course, we expect that the algorithms for finding the near-optimal solutions be efficient.

Definition 5.2.1 An algorithm $\mathcal{A}$ is an *approximation algorithm* for an optimization problem $Q = (I_Q, S_Q, f_Q, opt_Q)$, if on any instance $x \in I_Q$, the algorithm $\mathcal{A}$ produces a solution $y \in S_Q(x)$.

Note that here we have put no requirement on the approximation quality for an approximation algorithm. Thus, an algorithm that always produces a "trivial" solution for a given instance is an approximation algorithm. For example, an algorithm that always returns the empty set is an approximation algorithm for the KNAPSACK problem. To measure the quality of an approximation algorithm, we introduce the following concept.

Definition 5.2.2 An approximation algorithm $\mathcal{A}$ for an optimization problem $Q = (I_Q, S_Q, f_Q, opt_Q)$ has an *approximation ratio* $r(n)$, if on each instance $x \in I_Q$, the solution y produced by the algorithm $\mathcal{A}$ satisfies

$$\frac{Opt(x)}{f_Q(x,y)} \leq r(|x|) \quad \text{if } opt_Q = \max$$

$$\frac{f_Q(x,y)}{Opt(x)} \leq r(|x|) \quad \text{if } opt_Q = \min$$

where $Opt(x)$ is defined to be $\max\{f(x,y) \mid y \in S_Q(x)\}$ if $opt_Q = \max$ and to be $\min\{f(x,y) \mid y \in S_Q(x)\}$ if $opt_Q = \min$.

Remark 5.2.3 By the definition, an approximation ratio is at least as large as 1. The closer the approximation ratio to 1, the better the approximation quality of the approximation algorithm.

Definition 5.2.4 An optimization problem *can be polynomial-time approximated to a ratio* $r(n)$ if it has a polynomial-time approximation algorithm whose approximation ratio is bounded by $r(n)$.

As an example, let we consider the general MAKESPAN problem:

MAKESPAN

I_Q: the set of tuples $T = \{t_1, \ldots, t_n; m\}$, where t_i is the processing time for the i-th job and m is the number of identical processors

S_Q: $S_Q(T)$ is the set of partitions $P = (T_1, \ldots, T_m)$ of the numbers $t_1, \ldots, t_n$ into m parts

f_Q: $f_Q(T,P)$ is equal to the processing time of the largest subset in the partition P, that is, $f_Q(T,P) = \max_i\{\sum_{t_j \in T_i} t_j\}$

opt_Q: min

A simple approximation algorithm is based on the greedy method: to minimize the makespan, we always assign the next job to the processor that has the lightest load. This algorithm is due to R. Graham [59], and is given in Figure 5.1.

Using a data structure such as a 2-3 tree to organize the m processors using their loads as the keys, we can find the lightest loaded processor and update its load in the data structure in time $O(\log m)$. With this implementation, the algorithm **Graham-Schedule** runs in time $O(n \log m)$.

We study the approximation ratio of the algorithm **Graham-Schedule**.

Algorithm. Graham-Schedule
Input: $I = \langle t_1, \ldots, t_n; m \rangle$, all integers
Output: a schedule of the n jobs of processing time $t_1, \ldots, t_n$ on m identical processors

1. **for** $(i = 1$ **to** $n)$ **do** assign t_i to the processor with the lightest load;

Figure 5.1: Graham-Schedule

Theorem 5.2.1 *The algorithm* **Graham-Schedule** *for the* MAKESPAN *problem has approximation ratio bounded by* $2 - (1/m)$.

PROOF. Let $\alpha = \langle t_1, \ldots, t_n; m \rangle$ be an input instance for the MAKESPAN problem. Suppose that the algorithm **Graham-Schedule** constructs a schedule S for α with makespan T. Let P_1 be a processor that has the execution time T assigned by the scheduling S, i.e., P_1 finishes its work the latest under the schedule S.

If the processor P_1 is assigned only one job, then the job has processing time T, and any schedule on α has makespan at least T. In this case, the schedule S is an optimal schedule with approximation ratio 1.

So suppose that the processor P_1 is assigned at least two jobs. Let the last job J_0 assigned to the processor P_1 have processing time t_0. We have $T - t_0 > 0$. By our strategy, at the time the job J_0 is about to be assigned to the processor P_1, all processors have load at least $T - t_0$. This gives:

$$\sum_{i=1}^{n} t_i \geq m(T - t_0) + t_0 = mT - (m-1)t_0.$$

Thus

$$T \leq \frac{\sum_{i=1}^{n} t_i + (m-1)t_0}{m} = \frac{\sum_{i=1}^{n} t_i}{m} + \frac{m-1}{m} t_0.$$

Observe that the makespan $Opt(\alpha)$ of an optimal schedule on the instance α is at least $\left(\sum_{i=1}^{n} t_i\right)/m$, and at least t_0. We conclude

$$T \leq Opt(\alpha) + \frac{m-1}{m} Opt(\alpha).$$

This gives

$$\frac{T}{Opt(\alpha)} \leq 2 - \frac{1}{m},$$

and completes the proof. □

Let Q be an optimization problem. Suppose that we have developed a polynomial-time approximation algorithm A for Q and have derived that the approximation ratio of the algorithm A is bounded by r_0. Three natural questions regarding the approximation algorithm A are as follows.

1. Is the approximation ratio r_0 tight for the algorithm A? That is, is there another $r' < r_0$ such that the approximation ratio of the algorithm A is bounded by r'?

2. Is the approximation ratio r_0 tight for the problem Q? That is, is there another polynomial-time approximation algorithm A' of approximation ratio r' for the problem Q such that $r' < r_0$?

3. Can a faster approximation algorithm A' be constructed for the problem Q with approximation ratio at least as good as r_0?

To answer the first question, either we need to develop smarter analysis techniques that derive a smaller approximation ratio bound $r' < r_0$ for the algorithm A, or we construct input instances for the problem Q and show that on these input instances, the approximation ratio of the algorithm A can be arbitrarily close to r_0 (thus r_0 is a tight ratio for the algorithm A).

To answer the second question, either we need to develop a new (and smarter) approximation algorithm for Q with a smaller approximation ratio, or we need to develop a formal proof that no polynomial-time approximation algorithm for the problem Q can have approximation ratio smaller than r_0. Both directions could be very difficult. Developing a new approximation algorithm with a better approximation ratio may require a deeper understanding of the problem Q and new analysis techniques. On the other hand, only for very few optimization problems, a tight approximation ratio of polynomial time approximation algorithms has been derived. In general, it has been very little understood how to prove that to achieve certain approximation ratio would require more than polynomial time.

The third question is more practically oriented. Most approximation algorithms are simple thus their running time is bounded by a low degree polynomial such as $O(n^2)$ and $O(n^3)$. However, there are certain optimization problems for which the running time of the approximation algorithms is a very high degree polynomial such as n^{20}. These algorithms may provide a very good approximation ratio for the problems thus are of great theoretical interests. On the other hand, however, these algorithms seem impractical. Therefore, to keep the same approximation ratio but improve the running time of these algorithms is highly demanded in the computer implementations.

In the following, we will use the approximation algorithm **Graham-Schedule** for the MAKESPAN problem as an example to illustrate these three aspects regarding approximation algorithms for optimization problems.

Lemma 5.2.2 *The approximation ratio* $2 - (1/m)$ *for the approximation algorithm* **Graham-Schedule** *is tight.*

PROOF. To prove the lemma, we consider the following input instance for MAKESPAN: $\alpha = \langle t_1, t_2, \ldots, t_n; m \rangle$, where $n = m(m-1)+1$, $t_1 = t_2 = \cdots = t_{n-1} = 1$, and $t_n = m$. The algorithm **Graham-Schedule** assigns the first $n - 1 = m(m-1)$ jobs $t_1, \ldots, t_{n-1}$ to the m processors, each then has a load $m - 1$. Then the algorithm assigns the job t_n to the first processor, which then has a load $2m-1$. Therefore, the algorithm **Graham-Schedule** results in a schedule of the n jobs on m processors with makespan $2m - 1$.

On the other hand, the optimal schedule for the instance α is to assign the job t_n to the first processor and then assign the rest $n - 1 = m(m-1)$ jobs to the rest $m - 1$ processors. By this schedule, each processor has load exactly m. Thus, the optimal schedule has makespan m.

Thus, on this particular instance α, the approximation ratio of the algorithm **Graham-Schedule** is $(2m-1)/m = 2 - (1/m)$. This proves that $2 - (1/m)$ is a tight bound for the approximation ratio of the algorithm **Graham-Schedule**. □

Now we consider the second question: can we have a polynomial-time approximation algorithm for the MAKESPAN problem that has an approximation ratio better than $2-(1/m)$? By looking at the instance α constructed in the proof of Lemma 5.2.2, we should realize that the bad performance for the algorithm **Graham-Schedule** occurs in the situation where we first assign small jobs, which somehow gives a balanced assignment among the processors, while a latter large job may simply break the balance by increasing the load of one processor significantly while unchanging the load of the other processors. This then results in a very unbalanced assignment among the processors thus worsens the makespan. To avoid this situation, we pre-sort the jobs, in a non-increasing order of their processing time, before we apply the algorithm **Graham-Schedule**. We call this the **Modified Graham-Schedule** algorithm.

Theorem 5.2.3 *The* **Modified Graham-Schedule** *algorithm for the* MAKESPAN *problem has an approximation ratio bounded by* $4/3$.

PROOF. According to the algorithm, after the pre-sorting, we have the

instance satisfying $t_1 \geq t_2 \geq \cdots \geq t_n$, which is an input instance $\alpha = \langle t_1, \ldots, t_n; m \rangle$ to the algorithm **Graham-Schedule**. We analyze the approximation ratio of the algorithm.

Let T_0 be the makespan of an optimal schedule on the instance α of the MAKESPAN problem. Suppose k is the first index such that when the algorithm assigns the job t_k to a processor, the makespan of the schedule exceeds T_0. We first prove that $t_k \leq T_0/3$.

Suppose that $t_k > T_0/3$. Thus, we have $t_i > T_0/3$ for all $i \leq k$. Consider the moment when the algorithm **Modified Graham-Schedule** has made assignment on the jobs $t_1, \ldots, t_{k-1}$. By our assumption, the makespan of this assignment on the jobs $t_1, \ldots, t_{k-1}$ is not larger than T_0. Since each job t_i with $i \leq k$ is larger than $T_0/3$, this assignment has at most two of these $k-1$ jobs in each processor. Without loss of generality, we can assume that each P_i of the first h processors is assigned a single job t_i, $1 \leq i \leq h$, while each of the rest $m - h$ processors is assigned exactly two jobs from $t_{h+1}, \ldots, t_{k-1}$. Thus,

$$k - h - 1 = 2(m - h) \tag{5.4}$$

and by the assumption on the index k, for each $i \leq h$, we have $t_i + t_k > T_0$. Now consider any optimal schedule $\mathcal{S}_0$ on the instance α. The makespan of $\mathcal{S}_0$ is T_0. If a processor is assigned a job t_i by $\mathcal{S}_0$ with $i \leq h$, then the processor cannot be assigned any other jobs in $t_1, \ldots, t_k$ by $\mathcal{S}_0$ since t_k is the smallest among $t_1, \ldots, t_k$ and $t_i + t_k > T_0$. Moreover, no processor is assigned more than two jobs in $t_{h+1}, \ldots, t_k$ since each of these jobs has processing time larger than $T_0/3$. Therefore, we need at least $h + \lceil (k - h)/2 \rceil = h + (m - h + 1) = m + 1$ processors for the jobs $t_1, \ldots, t_k$ in order to keep the makespan larger than T_0 (note here we have used the equality (5.4)). This contradicts the fact that $\mathcal{S}_0$ is an optimal schedule of makespan T_0 on the instance $\langle t_1, \ldots, t_n; m \rangle$.

Thus, if t_k is the first job such that when the algorithm **Modified Graham-Schedule** assigns t_k to a processor, the makespan of the schedule exceeds T_0, then we must have $t_k \leq T_0/3$.

Now let $\mathcal{S}$ be the schedule constructed by the algorithm **Modified Graham-Schedule** with makespan T for the instance α. If $T = T_0$, then the approximation ratio is 1 so less than 4/3. If $T > T_0$, let processor P_j have load T and let t_k be the last job assigned to processor P_j. Since when the algorithm assigns t_k to P_j the makespan of $t_1, \ldots, t_k$ exceeds T_0, by the above discussion, we must have $t_k \leq T_0/3$ (t_k may not be the first such a job but recall that the jobs are sorted in non-increasing order). Let $T = t + t_k$. Then at the time the job t_k was assigned, all processors had load at least

t. Therefore, $mt \leq \sum_{i=1}^{k-1} t_i \leq \sum_{i=1}^{n} t_i$, which implies $t \leq \sum_{i=1}^{n} t_i/m \leq T_0$. This gives immediately

$$T = t + t_k \leq T_0 + T_0/3 = 4T_0/3.$$

That is, $T/T_0 \leq 4/3$. The theorem is proved. □

Note that $2 - (1/m) > 4/3$ for all $m > 1$. Therefore, though $2 - (1/m)$ is a tight bound for the approximation ratio of the algorithm **Graham-Schedule**, it is not a tight bound on the approximation ratios for approximation algorithms for the MAKESPAN problem.

We should point out that the bound $4/3$ is not quite tight for the algorithm **Modified Graham-Schedule**. A tight bound on the approximation ratio for the algorithm **Modified Graham-Schedule** has been derived, which is $(4 - 1/m)/3$. As an exercise, we leave the formal derivation of this bound for the algorithm to interested readers.

It is natural to ask whether the bound $(4 - 1/m)/3$ is tight on approximation ratios for approximation algorithms for the MAKESPAN problem. It is, in fact, not. For example, Hochbaum and Shmoys [66] have developed a polynomial time approximation algorithm of approximation ratio $1 + \epsilon$ for any fixed constant $\epsilon > 0$. Such an algorithm is called a *polynomial time approximation scheme* for the problem. Therefore, there are polynomial time approximation algorithms for the MAKESPAN problem whose approximation ratio can be arbitrarily close to 1. These kind of approximation algorithms will be investigated in more details in the next few chapters.

Unfortunately, the algorithm given in [66] runs in time $O((n/\epsilon)^{1/\epsilon^2})$, which is totally impractical even for moderate values of n and ϵ. Thus, we come to the third question: can we keep the approximation ratio $1 + \epsilon$ while improving the running time for approximation algorithms for the MAKESPAN problem? Progress has been made towards this direction. For example, more recently, Hochbaum and Shmoys [68] have developed an approximation algorithm for the MAKESPAN problem whose approximation ratio remains $1+\epsilon$ with running time improved to $O(n)+f(1/\epsilon)$, where $f(1/\epsilon)$ is a function independent of n.