Chapter 1

Introduction

This chapter starts with some examples of optimization problems and introduces the formal definition of an optimization problem. Necessary background in computer algorithms will be reviewed. We then discuss in detail two important optimization problems: the MINIMUM SPANNING TREE and the MATRIX-CHAIN MULTIPLICATION problems. Algorithms for solving these two problems are given with analysis of their running time. Through these given algorithms, two basic techniques in the design of algorithms are illustrated: the greedy method and the dynamic programming method. Finally, we give a brief discussion on NP-completeness theory, which will be essential for the topics covered throughout the book.

1.1 Optimization problems

Most computational optimization problems arise from practical problems in industry and science. The concept of optimization is now well rooted as a principle underlying the analysis of many complex decision or allocation problems. In general, an optimization problem consists of a set of *instances*, which take a certain well-specified form. Each instance is associated with a set of *solutions* such that each solution has a *value*. *Solving the optimization problem* means finding for each given instance a best (i.e., optimal) solution which has either the largest or the smallest associated value, depending on the description of the optimization problem.

Let us start with some examples of optimization problems. We will try to keep the discussion informal at the beginning so that the reader gains an intuitive understanding of optimization problems before the formal model is introduced.

One of the most famous optimization problems is the TRAVELING SALESMAN problem (abbr. TSP).

> TRAVELING SALESMAN (TSP)
>
> Given a set of cities and the cost of traveling between each pair of cities, find a traveling tour that visits all the cities and minimizes the cost.

Here each instance of the problem consists of a collection of cities and the costs of traveling between the cities, a solution to the instance is a traveling tour that visits all the cities, the value associated with the solution is the cost of the corresponding traveling tour, and the objective is to find a traveling tour that minimizes the traveling cost.

Another optimization problem comes from mathematical programming, the LINEAR PROGRAMMING problem, which has played a unique role in the study of optimization problems. In fact, a vast array of optimization problems can be formulated as instances of LINEAR PROGRAMMING.

> LINEAR PROGRAMMING (LP)
>
> Given a vector $(c_1, \ldots, c_n)$ of real numbers, and a set of linear constraints
>
> $$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n & \geq & a_1 \\ & \vdots & \\ a_{r1}x_1 + a_{r2}x_2 + ... + a_{rn}x_n & \geq & a_r \\ b_{11}x_1 + b_{12}x_2 + ... + b_{1n}x_n & \leq & b_1 \\ & \vdots & \\ b_{s1}x_1 + b_{s2}x_2 + ... + b_{sn}x_n & \leq & b_s \\ d_{11}x_1 + d_{12}x_2 + ... + d_{1n}x_n & = & d_1 \\ & \vdots & \\ d_{t1}x_1 + d_{t2}x_2 + ... + d_{tn}x_n & = & d_t \end{array} \tag{1.1}$$
>
> find a vector $(x_1, \ldots, x_n)$ of real numbers such that the value
>
> $$c_1x_1 + \cdots + c_nx_n$$
>
> is minimized.

Here an instance consists of a vector $(c_1, \ldots, c_n)$ of real numbers plus a set of linear constraints of the form given in (1.1), a solution to the instance is a vector $(x_1, \ldots, x_n)$ of real numbers satisfying the linear constraints, the value associated with the solution is $c_1x_1 + \cdots + c_nx_n$, and the objective is to find a solution $(x_1, \ldots, x_n)$ that minimizes the value $c_1x_1 + \cdots + c_nx_n$.

Both problems above are minimization problems. Below we give an example of maximization problems.

> OPTIMAL COURSE ASSIGNMENT
>
> Given a set of teachers $T = \{t_1, \ldots, t_p\}$ and a set of courses $C = \{c_1, \ldots, c_q\}$, and a set of pairs (t_i, c_j) indicating that teacher t_i can teach the course c_j, where $t_i \in T$ and $c_j \in C$, find a course assignment in which each teacher teaches at most one course and each course is taught by at most one teacher, and such that the maximum number of courses get taught.

Here an instance x consists of the set of the pairs (t_i, c_j), a solution y to the instance x is a subset A of the pairs in x in which each teacher appears at most once and each course appears at most once, the value associated with the solution y is the number of pairs in the subset A, and the objective is to find such a subset A with the maximum number of pairs.

Formally, an optimization problem is defined as follows.

Definition 1.1.1 An *optimization problem* Q is a 4-tuple $\langle I_Q, S_Q, f_Q, opt_Q \rangle$, where I_Q is the set of *input instances*, S_Q is a function such that for each input $x \in I_Q$, $S_Q(x)$ is the set of *solutions* to x, f_Q is the *objective function* that for each pair $x \in I_Q$ and $y \in S_Q(x)$, associates the real number $f_Q(x, y)$, and $opt_Q \in \{\max, \min\}$ specifies whether the problem is a *maximization problem* or a *minimization problem*.

Solving the optimization problem Q means for each given input instance x in I_Q, to find a solution y in $S_Q(x)$ such that the objective function value $f_Q(x, y)$ is optimized (maximized or minimized depending on opt_Q) over all solutions in $S_Q(x)$.

Based on this formulation, we list a few more examples of optimization problems. The list shows that optimization problems arise naturally in many applications.

The MINIMUM SPANNING TREE problem arises in network communication, in which we need to find a cheapest subnetwork that connects all nodes in the network. A network can be modeled as a weighted graph, in which

each vertex represents a node in the network, and each edge represents a connection between two corresponding nodes in the network. The weight of an edge indicates the cost of the corresponding connection.

MINIMUM SPANNING TREE (MSP)

I_Q: the set of all weighted (undirected) graphs G

S_Q: $S_Q(G)$ is the set of all spanning trees of the graph G

f_Q: $f_Q(G, T)$ is the weight of the spanning tree T of G.

opt_Q: min.

A problem that often arises in the domain of network communication, is to find the shortest path from a given starting position to a given final position. This problem is formulated as the SHORTEST PATH problem

SHORTEST PATH

I_Q: the set of all weighted graphs G with two specified vertices s and t in G

S_Q: $S_Q(G, s, t)$ is the set of all paths connecting s and t in G

f_Q: $f_Q(G, s, t; P)$ is the weight of the path P (i.e., the sum of weights of the edges in P) connecting s and t in G

opt_Q: min.

The following problem is a "dual" of the SHORTEST PATH problem, which looks for a "longest" path in a graph. We will see soon in this chapter that this problem, even restricted to graphs of very simple structures, has important applications in areas such as scheduling and computational biology. Recall that a *simple path* in a graph is a sequence of non-repeating vertices $(v_1, v_2, \ldots, v_k)$ in the graph where for all i, $1 \leq i \leq k-1$, $[v_i, v_{i+1}]$ is an edge in the graph.

LONGEST PATH

I_Q: the set of all weighted graphs G

S_Q: $S_Q(G)$ is the set of all simple paths P in G

f_Q: $f_Q(G, P)$ is the weight of the path P (i.e., the sum of weights of the edges in P)

opt_Q: max.

Note that the condition that the paths are simple in the definition of LONGEST PATH is necessary to make the problem meaningful: otherwise, an arbitrarily long path can be formed by repeatedly looping around a cycle in the graph.

Now we consider an optimization problem induced from the research in computational biology. In biological applications, people are often interested in knowing the "similarity" of two biological sequences, which can be either DNA sequences or protein sequences. The biological similarity can be defined in many ways, here in this example we consider the measure that is the length of a longest sequence that is "shared" by the two sequences. Formally, let $A = a_1 a_2 \cdots a_n$ be a sequence of symbols in a given alphabet Σ. For any subset $\{t_1, t_2, \ldots, t_k\}$ of indices in $\{1, 2, \ldots, n\}$ with $t_1 < t_2 < \cdots < t_k$, the sequence $a_{t_1} a_{t_2} \cdots a_{t_k}$ is called a *subsequence* of A. Note that the symbols in a subsequence do not have to be consecutive in the original sequence. For example, for the sequence $A = a_1 a_2 a_3 a_4 a_5$, the sequences $a_1 a_2 a_3$, $a_1 a_5$, and the "empty sequence" ϕ are all subsequences of A. Two sequences A_1 and A_2 are considered similar if they contain a very long subsequence in common. Therefore, we are interested in knowing the length of the longest sequence that is a subsequence of both the given sequences A_1 and A_2. Formally, the LONGEST COMMON SUBSEQUENCE problem is defined as follows:

LONGEST COMMON SUBSEQUENCE

I_Q: the set of all pairs (A_1, A_2) of sequences in a fixed alphabet Σ (Σ can be either finite or infinite)

S_Q: $S_Q(A_1, A_2)$ is the set of all sequences A' in Σ that is a common subsequence of A_1 and A_2

f_Q: $f_Q(A_1, A_2; A')$ is the length of A'

opt_Q: max.

Note for any two sequences A_1 and A_2, the solution set $S_Q(A_1, A_2)$ is always non-empty because at least the empty sequence ϕ is a common subsequence of A_1 and A_2.

The LONGEST COMMON SUBSEQUENCE problem can be naturally extended to the case for more than two sequences.

The next optimization problem takes its name from the following story: a thief robbing a safe finds a set of items of varying sizes and values that he could steal, but has only a small knapsack of capacity B that he can use to carry the goods. The thief tries to choose items for his knapsack in order to maximize the value of the total take. This problem can be interpreted as a

job scheduling problem in which each job corresponds to an item. The size of an item corresponds to the resource needed for finishing the job while the value of an item corresponds to the reward for finishing the job. Now with limited amount B of resources, we want to get the maximum reward.

KNAPSACK

I_Q: the set of tuples $T = \{\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle\}$, where s_i and v_i are for the size and value of the ith item, respectively, and B is the knapsack size

S_Q: $S_Q(T)$ is a subset S of pairs of form (s_i, v_i) in T such that the sum of all s_i in S is not larger than B

f_Q: $f_Q(T, S)$ is the sum of all v_i in S

opt_Q: max.

The following optimization problem arises in job scheduling on parallel processing systems. Suppose that we have a set of jobs J_1, ..., J_n, where the processing time of job J_i (on a single processor) is t_i, and a set of identical processors P_1, ..., P_m. Our objective is to assign the jobs to the processors so that the completion time of all jobs is minimized.

MAKESPAN

I_Q: the set of tuples $T = \{\langle t_1, \ldots, t_n; m\rangle\}$, where t_i is the processing time for the ith 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 subsets

f_Q: $f_Q(T, P)$ is the largest processing time of a 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 partition $P = (T_1, \ldots, T_m)$ of the job set $\{t_1, \ldots, t_n\}$ corresponds to a *schedule* $\mathcal{S}$ of the jobs $\{t_1, \ldots, t_n\}$ to the m processors so that the jobs in the subset T_i are assigned to the i-th processor, for $1 \leq i \leq m$. Thus, under the schedule $\mathcal{S}$, the processor T_i completes the jobs assigned to it at time $\sum_{t_j \in T_i} t_j$. The value $\max_i\{\sum_{t_j \in T_i} t_j\}$ is the time at which all processors complete the assigned jobs under the schedule $\mathcal{S}$, which is the *parallel completion time*, and also called the *makespan* of the schedule $\mathcal{S}$.

We close this section with the following graph optimization problem.

INDEPENDENT SET

I_Q: the set of undirected graphs G

S_Q: $S_Q(G)$ is the set of all subsets I of vertices in G such that no two vertices in I are adjacent

f_Q: $f_Q(G, I)$ is equal to the number of vertices in I

opt_Q: max.

The INDEPENDENT SET problem has been a very important optimization problem in the study and research in computational optiimizations and has drawn significant attentions in the research. The problem will also play an important role in our discussion.

1.2 Algorithmic preliminaries

The objective of this book is to discuss how optimization problems are solved using computer programs, which will be described as computer algorithms. The design and analysis of computer algorithms has been a very active research area in computer science since the introduction of the first modern computer. In this section, we briefly review some fundamentals in the design and analysis of computer algorithms. For further and more detailed discussion, the reader is referred to the excellent books in the area, such as Aho, Hopcroft, and Ullman [1], Cormen, Leiserson, Rivest, and Stein [29], and Knuth [85, 86].

Algorithms

The concept of algorithms came far earlier than modern computers. In fact, people have been using algorithms as long as they have been solving problems systematically. However, since the introduction of modern computers in the middle of the 20th century, it has become a common practice to refer by "algorithms" to "computer algorithms". Informally, an *algorithm* is a high level description of a computer program, which is a finite step-by-step specification of a halting procedure for solving a given problem. Each step of an algorithm consists of a finite number of operations, which in general include arithmetical operations, logical comparisons, transfer of control, and retrieving or storing data from/in computer memory.

We say that an algorithm $\mathcal{A}$ *solves* an optimization problem $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ if on each input instance $x \in I_Q$, the algorithm produces

an optimal solution $y \in S_Q(x)$ (by "optimal solution" y we mean that the solution y satisfies the condition $f_Q(x, y) = opt_Q\{f_Q(x, z) \mid z \in S_Q(x)\}$).

Encodings

To study the computational complexity of an algorithm, we first need to discuss how input instances and solutions of an optimization problem are represented in a computer. In general, an input instance or a solution to an input instance can be given as a sequence of symbols in a finite alphabet Σ. For example, an input instance of the MAKESPAN problem is a sequence starting with the symbol "(", then a sequence of integers separated by commas, then a symbol ";" followed by an integer m, and closed with the symbol ")". Thus, the alphabet for the input instances of MAKESPAN is $\Sigma = \{0, \ldots, 9, (,), ;, [,]\}$, (where $[,]$ means the symbol ","). Another example is the input instances of the TRAVELING SALESMAN problem, which are weighted graphs, and can be given by the adjacency matrix of the graphs organized in row major as a sequence of numbers. Now suppose that the finite alphabet Σ is fixed, then we can encode each sequence in Σ into a binary sequence as follows. Let q be the number of symbols in Σ, then each symbol in Σ can be encoded into a distinct binary string of length $\lceil \log q \rceil$. Therefore, each sequence of length n in Σ can be encoded into a binary sequence of length $n\lceil \log q \rceil$. Since q is in general a small constant, the binary representation of the sequence is not significantly different in length from the original sequence. Moreover, it is straightforward to convert a sequence in Σ into the corresponding binary sequence and vice versa. It is convincing that in general, input instances and solutions of an optimization problem, even when they are compound objects such as polygons, graphs, or formulas, can be effectively and efficiently encoded into binary sequences.

Therefore, we will use the *size* or *length* of an object w, denoted $|w|$, to refer to the length of the binary representation of the object w, where the object w can be an input instance, a solution to an input instance, or some other component of an optimization problem.

Asymptotic notations

Suppose that $\mathcal{A}$ is an algorithm that solves an optimization problem Q. It is reasonable to assume that for input instances of large size, the algorithm $\mathcal{A}$ spends more computational time. We will evaluate the performance of the algorithm $\mathcal{A}$ in terms of the size of input instances.

It is in general difficult and improper to calculate the precise number

of basic operations the algorithm $\mathcal{A}$ uses to find an optimal solution for a given input instance. There are several reasons for this. First, the machine (computer) model underlying the algorithm is not well-defined. For example, the operation "a++" (add 1 to a) can be implemented in one basic operation (using the C compiler) or three basic operations (retrieve a, add 1, and store the value back to a). Second, the time complexity for each different basic operation may vary significantly. For example, an integer multiplication operation is much more time-consuming than an integer addition operation. Third, one may not be happy to be told that the running time of an algorithm is $37|x|^3 + 13|x| \log(|x|) - 4723 \log^2(|x|)$. One would be more interested in: "*roughly* what is the complexity of algorithm $\mathcal{A}$?"

It has become standard in computer science to use asymptotic bounds in measuring the computational resources needed for an algorithm in order to solve a given problem. The following notations have been very useful in the asymptotic bound analysis. Given a function $t(n)$ mapping integers to integers, we denote by

- $O(t(n))$: the class C_1 of functions such that for each $g \in C_1$, there is a constant c_g such that $t(n) \geq c_g g(n)$ for all but a finite number of values of n. Roughly speaking, $O(t(n))$ is the class of functions that are at most as large as $t(n)$ asymptotically.
- $o(t(n))$: the class C_2 of functions such that for each $g \in C_2$, $\lim_{n\to\infty} g(n)/t(n) = 0$. Roughly speaking, $o(t(n))$ is the class of functions that are asymptotically less than $t(n)$.
- $\Omega(t(n))$: the class C_3 of functions such that for each $g \in C_3$, there is a constant c_g such that $t(n) \leq c_g g(n)$ for all but a finite number of values of n. Roughly speaking, $\Omega(t(n))$ is the class of functions which are asymptotically at least as large as $t(n)$.
- $\omega(t(n))$: the class C_4 of functions such that for each $g \in C_4$, $\lim_{n\to\infty} t(n)/g(n) = 0$. Roughly speaking, $\omega(t(n))$ is the class of functions that are asymptotically larger than $t(n)$.
- $\Theta(t(n))$: the class C_5 of functions such that for each $g \in C_5$, $g(n) = O(t(n)$ and $g(n) = \Omega(t(n))$. Roughly speaking, $\Theta(t(n))$ is the class of functions which are asymptotically of the same order as $t(n)$.

Complexity of algorithms

There are two common scenarios under which an algorithm can be analyzed: worst case and expected case scenarios. For the worst case analysis, we seek

the maximum amount of time used by the algorithm over all possible inputs. For the expected case analysis we normally assume a certain probability distribution on the input instances and study the performance of the algorithm for any input instance drawn from the distribution. Mostly, we are interested in the asymptotic analysis, i.e., the behavior of the algorithm as the input size approaches infinity. Since expected case analysis is usually harder to tackle, and moreover the probabilistic assumption sometimes is difficult to justify, emphasis will be placed on the worst case analysis. Unless otherwise specified, we shall consider only worst case analysis.

The *running time* of an algorithm on an input instance is defined to be the number of basic operations performed during the execution of the algorithm.

Definition 1.2.1 Let $\mathcal{A}$ be an algorithm solving an optimization problem Q and let $f(n)$ be a function. The *time complexity* of algorithm $\mathcal{A}$ is $O(f(n))$ if there is a function $f'(n) \in O(f(n))$ such that for every integer $n \geq 0$, the running time of $\mathcal{A}$ is bounded by $f'(n)$ for all input instances of size n.

Now we are ready for presenting an important terminology.

Definition 1.2.2 An algorithm $\mathcal{A}$ is a *polynomial-time algorithm* if there is a fixed constant c such that the time complexity of the algorithm $\mathcal{A}$ is $O(n^c)$. An optimization problem *can be solved in polynomial time* if it can be solved by a polynomial-time algorithm.

Note that this terminology is invariant for a large variety of encoding schemes and different definitions of input size, as long as these schemes and definitions define input size that are polynomially related. As we have seen above, the binary representation and the original representation of an input instance differ only by a small constant factor. Thus, the running time of a polynomial-time algorithm is not only bounded by a polynomial of the length of its binary representation, but also bounded by a polynomial of the length of its original representation. Even more, consider the INDEPENDENT SET problem for example. Let n be the number of vertices in the input instance graph G. Then n is polynomially related to the binary representation of the graph G – if we use an adjacency matrix for the graph G, the binary representation of the matrix has length $\Theta(n^2)$. Therefore, the running time of an algorithm solving INDEPENDENT SET is bounded by a polynomial in n if and only if it is bounded by a polynomial in the length of the input instance.

We must be a bit more careful however. For example, consider the problem FACTORING for which each input instance is an integer n and we are asked to factor n into its prime factors. For this problem, it is obviously improper to regard the input size as 1 or as n. The standard definition of the input size for FACTORING takes the input length as $\lceil \log n \rceil = O(\log n)$, which is the length of the binary representation of the integer n, and is not polynomially related to the quantities 1 and n.

Further assumptions on optimization problems

Polynomial-time algorithms are regarded as "easy", or feasible, computations. In general, given an optimization problem, our main concern is whether an optimal solution for each input instance can be found in polynomial time. For this, we shall assume that the other unimportant computational parts of the optimization problem can be ignored, or can be dealt with easily. In particular, we make the following assumptions using the terminology of polynomial-time computability. Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem. Throughout the book, we assume that

- there is a polynomial-time algorithm that can identify if a given string x represents a valid input instance in I_Q;
- there is a polynomial-time algorithm that, given an input instance $x \in I_Q$ and a string y, can test if y represents a valid solution to x, i.e., if $y \in S_Q(x)$;
- there is a polynomial-time algorithm that, given $x \in I_Q$ and $y \in S_Q(x)$, computes the value $f_Q(x, y)$.

1.3 Sample problems and their complexity

To illustrate the ideas for solving optimization problems using computer algorithms, we consider in this section the computational complexity for two sample optimization problems, and introduce two important techniques in designing optimization algorithms. We present an algorithm, using the *greedy method*, to solve the MINIMUM SPANNING TREE problem, and an algorithm, using the *dynamic programming method*, to solve the MATRIX-CHAIN MULTIPLICATION problem.

1.3.1 Minimum spanning trees

As described in Section 1.1, an input instance to the MINIMUM SPANNING TREE problem is a weighted graph $G = (V, E)$, and a solution to the input instance G is a spanning tree T in G. The spanning tree T is evaluated by its weight, i.e., the sum of weights of the edges in T. Our objective is to find a spanning tree with the minimum weight, which will be called a *minimum spanning tree*.

Suppose that we have constructed a subtree T_1 and that we know that T_1 is entirely contained in a minimum spanning tree T_0. Let us see how we can expand the subtree T_1 into a minimum spanning tree. Consider the set E' of edges that are not in T_1. We would like to pick an edge e in E' and add it to T_1 to make a larger subtree. For this, the edge e must satisfy the following two conditions:

1. $T_1 + e$ must remain a tree. That is, the edge e must keep $T_1 + e$ connected but not introduce a cycle in $T_1 + e$; and

2. the larger subtree $T_1 + e$ should be still contained in some minimum spanning tree.

The first condition can be easily tested. In fact, the condition is equivalent to the condition that the edge e has exactly one end in the subtree T_1. We will call an edge e a *fringe edge* if it satisfies this condition. Now let us consider the second condition. Since we have no information about any minimum spanning trees (we are attempting to construct one of them), how can we justify that a new edge e plus T_1 is still contained entirely in a minimum spanning tree? Naturally, a person working on this problem would think "well, since I am looking for a spanning tree of minimum weight, I *guess* I should pick the *lightest* fringe edge to keep the weight of my new subtree $T_1 + e$ small." This presents the main idea for an important optimization technique: the *greedy method*. In general, the greedy method always makes the choice that looks best at the moment hoping that this choice will lead to an optimal final solution for the problem.

It is conceivable that the greedy method does not always yield optimal solutions for a given problem. However, for quite a few optimization problems, it does. The MINIMUM SPANNING TREE problem fortunately belongs to this class of problems, as shown by the following theorem.

Theorem 1.3.1 *Suppose that the subtree T_1 is entirely contained in a minimum spanning tree of G. Let e be the fringe edge of minimum weight. Then the subtree $T_1 + e$ is entirely contained in a minimum spanning tree of G.*

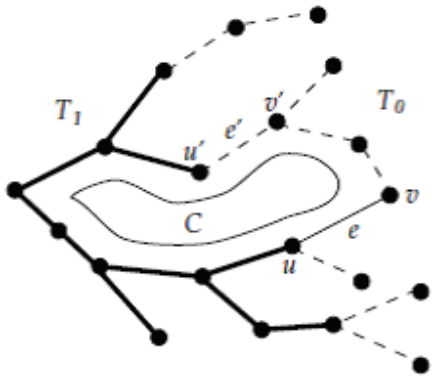

Figure 1.1: A cycle C in $T_0 + e$, where heavy lines are for edges in the constructed subtree T_1, and dashed lines are for edges in the minimum spanning tree T_0 that are not in T_1.

PROOF. Let T_0 be a minimum spanning tree that contains T_1. If the edge e is in T_0, then we are done. Thus, we assume that the edge e is not in the spanning tree T_0. Suppose that $e = [u, v]$, where the vertex u is in the subtree T_1 while the vertex v is not in T_1.

Then there is a cycle C in $T_0 + e$ that contains the edge $e = [u, v]$. Since u is in T_1 and v is not in T_1, and T_1 is entirely contained in T_0, there must be another edge $e' = [u', v']$ in the cycle C, $e' \neq e$, such that u' is in T_1 while v' is not in T_1. (See Figure 1.1, where heavy lines are for edges in the subtree T_1, dashed lines are for edges in T_0 that are not in T_1). In other words, e' is also a fringe edge. Moreover, $T_0' = T_0 + e - e'$ is also a spanning tree for the graph G. Since T_0 is a minimum spanning tree, we conclude that the weight of the tree T_0 is not larger than the weight of the tree T_0'.

On the other hand, since e' is also a fringe edge, by the choice we made in selecting the fringe edge e, we must have $weight(e) \leq weight(e')$. Therefore, the weight of the tree T_0 is not smaller than the weight of the tree $T_0' = T_0 + e - e'$.

In conclusion, the tree T_0' is also a minimum spanning tree for the graph G. Since the subtree $T_1 + e$ is entirely contained in T_0' (note that the edge e' is not in T_1), the theorem is proved. □

Therefore, starting with a smaller subtree contained in a minimum spanning tree, the greedy method will lead to a larger subtree contained in a minimum spanning tree. Since a spanning tree has exactly $n - 1$ edges, where

n is the number of vertices in the graph G, applying the greedy method $n-1$ times gives us a subtree T of $n-1$ edges which is entirely contained in a minimum spanning tree. In other words, the tree T itself is a minimum spanning tree.

What remains is to indicate how the above process can be started, i.e., what is the *first* such subtree. This is easy: pick any vertex v in G and let v be the first such a subtree. The vertex v is obviously contained in every minimum spanning tree of G.

We implement all these ideas into the following algorithm. Each vertex in the graph G can be either an "in-tree" vertex if it is contained in the currently constructed subtree T_1, or an "out-tree" vertex if it is not. Each edge in G may have one of the following four statuses: "tree-edge" if it is contained in the currently constructed subtree T_1, "cycle-edge" if it is not a tree-edge but both ends of it are in-tree vertices, "fringe-edge" if it has exactly one end in the currently constructed subtree T_1, and "unseen" otherwise. The formal algorithm is presented in Figure 1.2.

Algorithm. MST-PRIM

1. pick any vertex w and make it an in-tree vertex;
2. **for** each edge e incident on w **do** make e a fringe-edge;
3. let T_1 be the single-vertex tree consisting of the vertex w;
4. **loop** $n-1$ times
   pick a fringe-edge $e=[u,v]$ of minimum weight, where u is an in-tree vertex and v is an out-tree vertex;
4.1 $T_1 = T_1 + e$; make e a tree-edge;
4.2 **for** each edge e' incident on v **do**
   if e' is a fringe-edge
   then make e' a cycle-edge
   else if e' is an unseen-edge **then** make e' a fringe-edge
4.3 make v an in-tree vertex.

Figure 1.2: Prim's Algorithm for MINIMUM SPANNING TREE.

This algorithm is called *Prim's Algorithm* and is due to R. C. Prim [109]. We give some explanations on the detailed implementation of Prim's Algorithm. Suppose that the graph G has n vertices and m edges. We use an array of size n for the vertices and an array of size m for the edges. The status of a vertex is recorded in the vertex array and the status of an edge is recorded in the edge array. To find the fringe-edge of the minimum weight,

we only need to scan the edge array (the weight of an edge can be directly read from the adjacency matrix for G). Moreover, to update the status of the edges incident to a vertex v, we can again scan the edge array and work on those edges of which one end is v. Therefore, each execution of the loop body of the **loop** in Step 4 takes time $O(m)$. Since the loop body is executed exactly $n-1$ times, we conclude that the running time of Prim's Algorithm is bounded by $O(nm)$, which is certainly bounded by a polynomial of the length of the input instance G. In conclusion, the MINIMUM SPANNING TREE problem can be solved in polynomial time.

It is possible to improve the running time of the algorithm. For example, the edge array can be replaced by a more efficient data structure, which supports $O(\log n)$-time operations for finding the minimum weight edge and for updating the weight for an edge. Then since each edge is selected as the fringe-edge of minimum weight at most once, and the status of each edge is changed at most twice (from an unseen-edge to a fringe-edge and from a fringe-edge to a tree-edge or to a cycle-edge), we conclude that the Prim's Algorithm can be implemented so its running time is $O(m \log n)$. More detailed description of this improvement can be found in [29].

1.3.2 Matrix-chain multiplication

In this subsection, we study another important optimization technique: dynamic programming method. We illustrate the technique by presenting an efficient algorithm for the MATRIX-CHAIN MULTIPLICATION problem. An instance of the MATRIX-CHAIN MULTIPLICATION problem is a list of $n+1$ positive integers $D = (d_0, d_1, \ldots, d_n)$, representing the dimensions for n matrices M_1, ..., M_n, where M_i is a $d_{i-1} \times d_i$ matrix. A solution to the instance D is an indication R of the order of the matrix multiplications for the product $M_1 \times M_2 \times \cdots \times M_n$. The value for the solution R is the number of element multiplications performed to compute the matrix product according to the order R. Our objective is to find the computation order so that the number of element multiplications is minimized.

We start with a simple observation. Suppose that the optimal order is to first compute the matrix $P_1 = M_1 \times \cdots \times M_k$ and the matrix $P_2 = M_{k+1} \times \cdots \times M_n$, and then compute the final product by multiplying P_1 and P_2. The number of element multiplications for computing $P_1 \times P_2$ is easy: it should be $d_0 d_k d_n$ since P_1 is a $d_0 \times d_k$ matrix and P_2 is a $d_k \times d_n$ matrix. Now how do we decide the minimum number of element multiplications for computing the matrices P_1 and P_2? We notice that the corresponding matrix chains for the matrices P_1 and P_2 are shorter than the chain in

the original instance. Thus, we can apply the same method recursively to find the numbers of element multiplications for computing P_1 and P_2. The numbers of element multiplications found by the recursive process plus the number $d_0 d_k d_n$ give the number of element multiplications for this optimal order.

However, how do we find the index k? We have no idea. Thus, we try all possible indices from 1 to $n-1$, apply the above recursive process, and pick the index that gives us the minimum number of element multiplications.

This idea is also applied to any subchain in the matrix-chain $M_1 \times M_2 \times \cdots \times M_n$. For a subchain $M_i \times \cdots \times M_j$ of h matrices, we consider factoring the chain at the first, the second, ..., and the $(h-1)$st matrix multiplication "$\times$" in the subchain. For each factoring, we compute the desired number for each of the two corresponding smaller subchains. Note that this recursive process must terminate — since for subchain of one matrix, the desired number is 0 by the definition of the problem.

We organize the idea into the recursive algorithm given in Figure 1.3, which computes the minimum number of element multiplications for the subchain $M_i \times \cdots \times M_j$. We use *ind* to record the index for the best factoring we have seen so far, and use *num* to record the number of element multiplications based on this factoring.

Algorithm. Recursive-MCM(i, j)
1. **if** $i \geq j$ **then** return $(i, 0)$; Stop;
2. $num = \infty$; $ind = 0$;
3. **for** $k = i$ **to** $j - 1$ **do**
3.1 \\ recursively work on $M_i \times \cdots \times M_k$:
$(ind_1, num_1) =$ Recursive-MCM(i, k);
3.2 \\ recursively work on $M_{k+1} \times \cdots \times M_j$:
$(ind_2, num_2) =$ Recursive-MCM$(k+1, j)$;
3.3 **if** $(num > num_1 + num_2 + d_{i-1} d_k d_j)$
then $ind = k$; $num = num_1 + num_2 + d_{i-1} d_k d_j$;
4. return (ind, num).

Figure 1.3: Recursive algorithm for MATRIX-CHAIN MULTIPLICATION

What is the time complexity of this algorithm? Let $T(h)$ be the running time of the algorithm **Recursive-MCM** when it is applied to a matrix chain of h matrices. On the matrix chain of h matrices, the algorithm needs to try, for $k = 1, \ldots, h-1$, the factoring at the kth "$\times$" in the chain, which

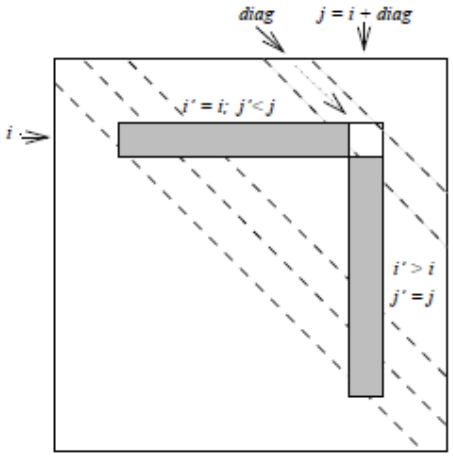

Figure 1.4: The order for computing the elements in NUM and IND.

induces the recursive executions of the algorithm on a chain of k matrices and on a chain of $h-k$ matrices. Thus, we have.

$$\begin{aligned} T(h) &\geq [T(1)+T(h-1)]+[T(2)+T(h-2)]+\cdots+[T(h-1)+T(1)] \\ &= 2[T(1)+T(2)+\cdots+T(h-1)] \\ &\geq hT(h/2), \end{aligned}$$

with a terminating condition $T(h) = O(1)$ for $h \leq 1$.

From the relation $T(h) \geq hT(h/2)$, it is easy to see that $T(h) = h^{\Omega(\log h)}$. Thus, for a chain of n matrices, i.e., if the input instance is a list of $n+1$ integers, the running time of the algorithm **Recursive-MCM** is at least $n^{\Omega(\log n)}$, which is much larger than any polynomial of n.

We now discuss how the above idea can be modified to achieve a more efficient algorithm. Observe that in the above recursive algorithm, for each subchain, the recursive process is applied on the subchain many times. For example, suppose we apply the algorithm on the matrix chain $M_1 \times \cdots \times M_7$, then the algorithm **Recursive-MCM** is applied to the subchain $M_1 \times M_2$ at least once when we factor the original chain at each of the ith "$\times$", for $2 \leq i \leq 6$. It is the repeatedly applications of the recursive process on the same subchain that make the algorithm inefficient.

A natural solution to this is to store the intermediate results when they are computed. Therefore, when next time we need the results again, we can retrieve them directly, instead of re-computing them. Now let us come back to the original MATRIX-CHAIN MULTIPLICATION problem. We use two

2-dimensional arrays NUM$[1..n, 1..n]$ and IND$[1..n, 1..n]$, where IND$[i, j]$ is used to record the index in the subchain $M_i \times \cdots \times M_j$ at which factoring the subchain gives the minimum number of element multiplications, and NUM$[i, j]$ is used to record the minimum number of element multiplications for computing the product of the subchain.[1] Since for computing the values for NUM$[i, j]$ and IND$[i, j]$, where $i \leq j$, we need to know the values for NUM$[i', j']$, for $i' = i$ and $j' < j$ and for $i' > i$ and $j' = j$, the values for the two 2-dimensional arrays IND and NUM will be computed from the diagonals of the arrays then moving toward the upper right corner (See Figure 1.4). Note that the values for the diagonal elements in the arrays IND and NUM are obvious: NUM$[i, i] = 0$, and IND$[i, i]$ has no meaning.

The algorithm is presented in Figure 1.5.

```
Algorithm.  Dyn-Prog-MCM
1.  for i = 1 to n do NUM[i, i] = 0;
2.  for diag = 1 to n − 1 do
        for i = 1 to n − diag do
            j = i + diag;
            num = ∞;
            for k = i to j − 1 do
                if num > NUM[i, k] + NUM[k + 1, j] + d_{i−1} d_k d_j
                then  num = NUM[i, k] + NUM[k + 1, j] + d_{i−1} d_k d_j;
                      IND[i, j] = k;
            NUM[i, j] = num;
```

Figure 1.5: Dynamic programming for MATRIX-CHAIN MULTIPLICATION

The analysis of the algorithm **Dyn-Prog-MCM** is straightforward: Step 2 dominates the running time and consists of loops of depth 3. Each execution of the inner loop body takes constant time. Thus, the running time of the algorithm is $O(n^3)$. This concludes that the problem MATRIX-CHAIN MULTIPLICATION can be solved in polynomial time.

We make a final remark to explain how a solution can be obtained from the results of the algorithm **Dyn-Prog-MCM**. With the values of the arrays NUM and IND being available, by reading the value IND$[1, n]$, suppose IND$[1, n] = k$, we know that input matrix chain $M_1 \times \cdots \times M_n$ should be factored at the index k. Now with the values IND$[1, k]$ and IND$[k + 1, n]$,

[1] Here we use the word "2-dimensional array" instead of "matrix" to distinguish NUM$[1..n, 1..n]$ and IND$[1..n, 1..n]$ from the matrices in the input instances.

we will know where the two subchains $M_1 \times \cdots \times M_k$ and $M_{k+1} \times \cdots \times M_n$ should be factored, and so on. A simple recursive algorithm can be written that, with the array IND as input, prints the expression, which is the chain $M_1 \times \cdots \times M_n$ with proper balanced parentheses inserted, indicating the order for computing the matrix product with the minimum number of element multiplications.

The algorithm **Dyn-Prog-MCM** illustrates the principle of an important technique for optimization algorithms — the *dynamic programming method*. A dynamic programming algorithm stores intermediate results and/or solutions for small subproblems and looks them up, rather than recomputing them when they are needed later for solving larger subproblems. In general, a dynamic programming algorithm solves an optimization problem in a bottom-up fashion, which includes charactering optimal solutions to a large problem in terms of solutions to smaller subproblems, computing the optimal solutions for the smallest subproblems, saving the solutions to subproblems to avoid re-computations, and combining solutions to subproblems to compute optimal solution for the original problem.

1.4 NP-completeness theory

NP-completeness theory plays a fundamental role in the study of optimization problems. In this section, we give a condensed description for NP-completeness theory. For a more formal and detailed discussion, the reader is referred to Garey and Johnson [52].

NP-completeness theory was motivated by the study of computational optimization problems, in the hope of providing convincing lower bounds on the computational complexity for certain optimization problems. However, for discussion convenience and for mathematical accuracy, NP-completeness theory is developed to be applied only to a class of simplified optimization problems — decision problems. A *decision problem* is a problem for which each instance has one of two possible answers — "yes" or "no". An instance with the answer "yes" will be called a *yes-instance* for the problem, and an instance with the answer "no" will be called a *no-instance* for the problem.

The following SATISFIABILITY (abbr. SAT) problem is a very well-known example of decision problems.

> SATISFIABILITY (SAT)
>
> Given a Boolean formula F in the conjunctive normal form (simply, a CNF formula F), is there an assignment to the variables in F so that the formula F evaluates to TRUE?

Thus, a CNF formula that is satisfiable (i.e., takes the value TRUE on some assignment) is a yes-instance for SATISFIABILITY, while a CNF formula that is not satisfiable is a no-instance for SATISFIABILITY.

An optimization problem Q can be converted into a decision problem by introducing a parameter to be compared with the optimal value of an instance. For example, a decision version of TRAVELING SALESMAN can be formulated as follows. An instance of the decision problem is of the form (G, k), where G is a weighted complete graph and k is an integer. The question the decision problem asks on instance (G, k) is: "Is there a traveling tour in G that visits all vertices of G and has weight bounded by k?"

In general, the decision version of an optimization problem is somehow easier than the original optimization problem. Therefore, the computational hardness of the decision problem implies the computational hardness for the original optimization problem. NP-completeness theory provides strong evidence for the computational hardness for a large class of decision problems, which implies convincingly the computational difficulties for a large variety of optimization problems.

We say that an algorithm $\mathcal{A}$ *accepts* a decision problem Q if on every yes-instance x of Q, the algorithm $\mathcal{A}$ returns "yes" (i.e., "accepts" x), while on all other inputs x' (including the inputs that do not encode an instance of Q), the algorithm $\mathcal{A}$ returns "no" (i.e., "rejects" x').

Definition 1.4.1 A decision problem Q is in *the class* P if it can be accepted by a polynomial-time algorithm.

In a more general and extended sense, people also say that a problem Q is in the class P if Q can be solved in polynomial time, even through sometimes the problem Q is not a decision problem. For example, people do say that the MINIMUM SPANNING TREE problem and the MATRIX-CHAIN MULTIPLICATION problem are in the class P. Thus, the class P represents the class of computational problems that are "feasible" in practical computing.

Unfortunately, many decision problems, in particular many decision problems converted from optimization problems, do not seem to be in the class P. A large class of these problems seem to be characterized (note: not "solved") by polynomial-time algorithms in a more generalized sense, as described by the following definition.

Definition 1.4.2 A decision problem Q is in *the class* NP if it can be characterized by a polynomial time algorithm $\mathcal{A}$ in the following manner. There is a fixed polynomial $p(n)$ such that

1. If x is a yes-instance for Q, then there is a binary string y of length $\leq p(|x|)$ such that on input (x, y) the algorithm $\mathcal{A}$ returns "yes";

2. If x is a no-instance for Q, then for *any* binary string y of length $\leq p(|x|)$, on input (x, y) the algorithm $\mathcal{A}$ returns "no".

Thus, a problem Q in NP is the one whose yes-instances x can be easily (i.e., in polynomial time) *verified* (by the algorithm A) when a short *proof* (i.e., y, whose length is bounded by the polynomial p of $|x|$) is given. The polynomial time algorithm $\mathcal{A}$ works in the following manner. If the input x is a yes-instance for the problem Q (this fact is not known to the algorithm $\mathcal{A}$ in advance), then with a *correct* proof or "hint" y, the algorithm $\mathcal{A}$ will be convinced and correctly conclude "yes". On the other hand, if the input x is a no-instance for the problem Q, then *no matter what hint y is given*, the algorithm $\mathcal{A}$ cannot be fooled to conclude "yes". In other words, the polynomial-time algorithm $\mathcal{A}$ simulates a proof checking process for theorems with short proofs. The polynomial-time algorithm $\mathcal{A}$ can be regarded as an experienced college professor. If a true theorem x is given together with a correct (and short) proof y, then the professor will conclude the truth for the theorem x. On the other hand, if a false theorem x is presented, then no matter what "proof" is provided (it has to be invalid!) the professor would not be fooled to conclude the truth for the theorem x.

We should point out that although the polynomial-time algorithm $\mathcal{A}$ can check the proof y for an instance x, $\mathcal{A}$ has no idea how the proof y can be developed. Alternatively, the class NP can be defined to be the set of those decision problems that can be accepted by *nondeterministic polynomial-time algorithms*, which can guess the proof. Therefore, on an input x that is a yes-instance, the nondeterministic polynomial-time algorithm guesses a proof y, checks the pair (x, y), and accepts x if y is a correct proof of x; while on an input x that is a no-instance, with any guessed proof y (that has to be incorrect), the algorithm checking the pair (x, y) would conclude "no".

The decision version of the TRAVELING SALESMAN problem, for example, is in the class NP: an instance (G, k), where G is a weighted complete graph and k is an integer, asks whether there is a traveling tour in G that visits all vertices and has weight bounded by k. A polynomial-time algorithm $\mathcal{A}$ can be easily designed as follows. On input pair (x, y), where $x = (G, k)$, the algorithm $\mathcal{A}$ accepts if and only if y represents a tour in G that visits all vertices and has weight not larger than k. Thus, if $x = (G, k)$ is a yes-instance, then with a proof y, which is a tour in G that visits all vertices and has weight not larger than k, the algorithm $\mathcal{A}$ will accept the pair (x, y). On

the other hand, if $x = (G, k)$ is a no-instance, then no matter what proof y is given, the algorithm $\mathcal{A}$ will find out that y is not the desired tour (since there does not exist a desired tour in G), so $\mathcal{A}$ rejects the pair (x, y).

We also point out that every decision problem in the class P is also in the class NP: suppose that Q is a problem in the class P and that $\mathcal{A}$ is a polynomial-time algorithm solving Q. The algorithm $\mathcal{A}$ can be modified so that the new algorithm ignores the hint y and computes the correct answer for a given instance x directly.

Unlike the class P, it is not that natural and obvious how the concept NP can be generalized to problems that are not decision problems. On the other hand, based on the characterization of "having a short proof y", people did extend the concept NP to optimization problems, as given in the following definition. This definition has become standard.

Definition 1.4.3 An optimization problem $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ is an NP *optimization* (or shortly NPO) problem if there is a polynomial $p(n)$ such that for each instance $x \in I_Q$, there is an optimal solution $y \in S_Q(x)$ whose length $|y|$ is bounded by $p(|x|)$.

Most interesting optimization problems are NPO problems. In particular, all optimization problems listed in Section 1.1 plus all optimization problems we are studying in this book are NPO problems. In general, if an optimization problem is an NPO problem, then it has a decision problem version that is in the class NP.

Now let us go back to NP-completeness theory. A very important concept in NP-completeness theory is *reducibility*, which is defined as follows.

Definition 1.4.4 Let Q_1 and Q_2 be decision problems. Problem Q_1 is *polynomial-time* (*many-one*) *reducible to* problem Q_2 (written as $Q_1 \leq_m^p Q_2$) if there is a function r computable in polynomial time such that for all x, x is a yes-instance for Q_1 if and only if $r(x)$ is a yes-instance for Q_2.

The relation $Q_1 \leq_m^p Q_2$ indicates that, up to a polynomial time computation, the problem Q_2 is not easier than the problem Q_1 (or equivalently, the problem Q_1 is not harder than the problem Q_2). Therefore, the relation $Q_1 \leq_m^p Q_2$ sets a lower bound for the computational complexity of the problem Q_2 in terms of that of problem Q_1, and also sets an upper bound for the computational complexity for the problem Q_1 in terms of that of the problem Q_2. In particular, we have the following consequence.

Lemma 1.4.1 *Let Q_1 and Q_2 be two decision problems. If $Q_1 \leq_m^p Q_2$ and Q_2 is in the class* P*, then the problem Q_1 is also in the class* P*.*

PROOF. Let r be the function that is computed by an algorithm $\mathcal{A}_1$ of running time $O(n^c)$, such that x is a yes-instance for Q_1 if and only if $r(x)$ is a yes-instance for Q_2, and let $\mathcal{A}_2$ be another algorithm that accepts the decision problem Q_2 and has running time $O(n^d)$, where both c and d are fixed constants. An algorithm $\mathcal{A}_3$ for the problem Q_1 can be designed as follows. On an input x, $\mathcal{A}_3$ first computes $r(x)$ by calling the algorithm $\mathcal{A}_1$ as a subroutine. This takes time $O(|x|^c)$. Note since the running time of $\mathcal{A}_1$ is bounded by $O(|x|^c)$, the length $|r(x)|$ of the output of $\mathcal{A}_1$ is also bounded by $O(|x|^c)$. The algorithm $\mathcal{A}_3$ calls the algorithm $\mathcal{A}_2$ to check whether $r(x)$ is a yes-instance for the problem Q_2. This takes time $O(|r(x)|^d) = O(((O(|x|))^c)^d) = O(|x|^{cd})$. Now the algorithm $\mathcal{A}_3$ concludes that x is a yes-instance for Q_1 if and only if $r(x)$ is a yes-instance for Q_2. According to the definitions, the algorithm $\mathcal{A}_3$ correctly accepts the decision problem Q_1. Moreover, since the running time of the algorithm $\mathcal{A}_3$ is bounded by $O(|x|^c) + O(|x|^{cd})$, which is bounded by a polynomial of $|x|$, we conclude that the problem Q_1 is in the class P. □

We give an example of a polynomial-time reduction by showing how the SATISFIABILITY problem is polynomial-time reducible to the following decision version of the INDEPENDENT SET problem.

> INDEPENDENT-SET(D)
>
> Given a graph G and an integer k, is there a set I of at least k vertices in G such that no two vertices in I are adjacent?

The algorithm $\mathcal{A}$ computing the reduction function r from the SATISFIABILITY problem to the INDEPENDENT-SET(D) problem works as follows. Let $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be an instance of the SATISFIABILITY problem, where each C_i (called a *clause*) is a disjunction $C_i = (l_{i,1} \vee l_{i,2} \vee \cdots \vee l_{i,n_i})$ of boolean literals (a *boolean literal* is either a boolean variable or its negation). The algorithm $\mathcal{A}$ constructs a graph G_F that has $\sum_{i=1}^{m} n_i$ vertices such that each vertex in G_F corresponds to a literal *appearance* in the formula F (note that each literal may have more than one appearance in F). Two vertices in G_F are adjacent if one of the following two conditions holds: (1) the two corresponding literal appearances are in the same clause, or (2) the two corresponding literal appearances are opposite to each other, i.e., one is the negation of the other. Now, for the instance F for the SATISFIABILITY problem, the value of the function $r(F)$ is (G_F, m), which is an instance for the

INDEPENDENT-SET(D) problem. It is easy to see that given the instance F for SATISFIABILITY, the instance (G_F, m) for INDEPENDENT-SET(D) can be constructed in polynomial time by the algorithm $\mathcal{A}$.

We show that F is a yes-instance for SATISFIABILITY if and only if (G_F, m) is a yes-instance for INDEPENDENT-SET(D). Suppose that F is a yes-instance for SATISFIABILITY. Then there is an assignment α to the variables in F that makes F TRUE. Thus, for each clause C_i, the assignment α sets at least one literal appearance l_{i,h_i} in C_i TRUE. Now pick the set I of the m vertices in G_F that correspond to the m literal appearances l_{i,h_i}, $i = 1, \ldots, m$. No two of these m vertices in I are adjacent by the construction of the graph G_F since (1) they are not in the same clause and (2) the assignment α cannot set two opposite literals both to TRUE. Therefore $r(F) = (G_F, m)$ is a yes-instance for INDEPENDENT-SET(D).

Now suppose that $r(F) = (G_F, m)$ is a yes-instance for INDEPENDENT-SET(D). Let $I = \{v_1, \ldots, v_m\}$ be a set of m vertices in G_F such that no two vertices in I are adjacent. Let I corresponds to the set $L_I = \{l_{1,h_1}, \ldots, l_{m,h_m}\}$ of m literal appearances in F. Since any two literal appearances in the same clause in F correspond to two adjacent vertices in G_F, each clause in F has exactly one literal appearance in the set L_I. Moreover, no two literal appearances in L_I contradict each other — otherwise the two corresponding vertices in I would have been adjacent. Therefore, an assignment α to the variables in F can be constructed that sets all the literal appearances in L_I TRUE: if a boolean variable x is in L_I, then $\alpha(x) =$ TRUE; if the negation $\overline{x}$ of a boolean variable x is in L_I then $\alpha(x) =$ FALSE; and if neither x nor $\overline{x}$ is in L_I, then α sets x arbitrarily. Note that the assignment α sets at least one literal in each clause in F TRUE, thus making the formula F TRUE. Consequently, F is a yes-instance for SATISFIABILITY.

This completes the polynomial-time reduction from the SATISFIABILITY problem to the INDEPENDENT-SET(D) problem.

The foundation of NP-completeness theory was laid by the following theorem.

Theorem 1.4.2 (Cook's Theorem) *Every decision problem in the class* NP *is polynomial-time many-one reducible to the* SATISFIABILITY *problem.*

PROOF. A formal proof for this theorem involves a very careful investigation on the precise definitions of algorithms and of the underlying computational models supporting the algorithms. Here we give a proof for the theorem that explains the main proof ideas but omits the detailed discussion related to computational models. A more complete proof for the theorem can be

found in Garey and Johnson [52].

Suppose that Q is a decision problem in NP, and that $\mathcal{A}$ is a polynomial-time algorithm such that for any instance x of Q, if x is a yes-instance, then there is a binary string y_x of length polynomial in $|x|$, such that the algorithm $\mathcal{A}$ accepts (x, y_x), and if x is a no-instance, then for any binary string y, the algorithm $\mathcal{A}$ rejects (x, y), where the length of the binary string y is bounded by a polynomial of $|x|$. We show how the problem Q is polynomial-time reducible to the SATISFIABILITY problem.

The algorithm $\mathcal{A}$ can be converted into a boolean formula F (this statement needs a thorough justification but is not surprising: computer algorithms are implementable in a digital computer, which basically can *only* do boolean operations.) Moreover, the formula F can be made in the conjunctive normal form. The input to the formula F is of the form (x, y), where both x and y are binary strings (i.e., sets of Boolean vairables), such that $F(x, y) = \text{TRUE}$ if and only if the algorithm $\mathcal{A}$ accepts the pair (x, y). Now for a given instance x_0 for the problem Q, the instance for SATISFIABILITY is $F_0 = F(x_0, y)$. That is, the formula F_0 is obtained from the formula $F(x, y)$ with the first parameter x assigned to the value x_0 and the second parameter y left unassigned. Thus, with the parameter x_0 that is an instance of the problem Q, $F_0 = F(x_0, y)$ is a CNF formula whose variables correspond to the unassigned parameter y. It can be proved that there is a polynomial-time algorithm that given x_0 constructs F_0.

Now if x_0 is a yes-instance for the problem Q, then by the definition, there is a binary string y_0 such that the algorithm $\mathcal{A}$ accepts (x_0, y_0). Thus, on this assignment y_0 to F_0, the formula $F_0(y_0) = F(x_0, y_0)$ gets value TRUE, i.e., the formula F_0 is satisfiable and is a yes-instance for SATISFIABILITY. On the other hand, if x_0 is a no-instance, then the algorithm $\mathcal{A}$ does not accept any pair (x_0, y), i.e., the formula $F_0(y) = F(x_0, y)$ is not TRUE for any y. Therefore, F_0 is not satisfiable and is a no-instance for SATISFIABILITY.

This completes the polynomial-time reduction that for an instance x_0 of Q produces an instance $F_0 = (x_0, y)$ of SATISFIABILITY such that x_0 is a yes-instance of Q if and only if F_0 is a yes-instance of SATISFIABILITY. Thus, the problem Q in NP is polynomial-time reducible to the SATISFIABILITY problem. Since Q is an arbitrary problem in NP, the theorem is proved. □

According to the definition of polynomial-time reducibility, Theorem 1.4.2 indicates that no problems in the class NP is essentially harder than the SATISFIABILITY problem. This gives a (relative) lower bound on the computational complexity for the SATISFIABILITY problem. Motivated by this theorem, we have the following definition.

Definition 1.4.5 A decision problem Q is NP-*hard* if every problem in the class NP is polynomial-time many-one reducible to Q.

A decision problem Q is NP-*complete* if Q is in the class NP and Q is NP-hard.

In particular, the SATISFIABILITY problem is NP-hard and NP-complete (it is easy to see that the SATISFIABILITY problem is in the class NP).

According to Definition 1.4.5 and Lemma 1.4.1, if an NP-hard problem can be solved in polynomial time, then so can *all* problem in NP. On the other hand, the class NP contains numerous hard problems, such as the decision version of the TRAVELING SALESMAN problem and of the INDEPENDENT SET problem. It can be shown that if these decision versions can be solved in polynomial time, then so can the corresponding optimization problems. People have worked very hard for decades to derive polynomial-time algorithms for these decision problems and optimization problems without much success. This fact somehow has convinced people that there are problems in the class NP that cannot be solved in polynomial time. Therefore, if we can show that a problem is NP-hard, then it should be a very strong evidence that the problem cannot be solved in polynomial time. This essentially is the basic philosophy behind the development of the NP-completeness theory.

However, how do we show the NP-hardness of a given problem? It is in general not feasible to examine *all* problems in NP and show that each of them is polynomial-time reducible to the given problem. Techniques used in Theorem 1.4.2 do not seem to generalize: Theorem 1.4.2 is kind of fortuitous because the SATISFIABILITY problem is a logic problem and algorithms *happen* to be characterized by logic expressions. Thus, to prove the NP-hardness for other problems, it seems that we need new techniques, which are, actually not new, the reduction techniques we have seen above.

Lemma 1.4.3 *Let Q_1, Q_2, and Q_3 be decision problems. If $Q_1 \leq_m^p Q_2$ and $Q_2 \leq_m^p Q_3$, then $Q_1 \leq_m^p Q_3$.*

PROOF. Suppose that r_1 is a polynomial-time computable function such that x is a yes-instance for Q_1 if and only if $r_1(x)$ is a yes-instance for Q_2, and suppose that r_2 is a polynomial-time computable function such that y is a yes-instance for Q_2 if and only if $r_2(y)$ is a yes-instance for Q_3. It is easy to verify that the function $r(x) = r_2(r_1(x))$ is also polynomial-time computable. Moreover, x is a yes-instance for Q_1 if and only if $r_1(x)$ is a yes-instance for Q_2, which is true if and only if $r(x) = r_2(r_1(x))$ is a yes-instance for Q_3. Thus, $Q_1 \leq_m^p Q_3$. □

Corollary 1.4.4 *Let Q_1 and Q_2 be decision problems such that the problem Q_1 is* NP*-hard and that $Q_1 \leq_m^p Q_2$. Then the problem Q_2 is* NP*-hard.*

PROOF. Let Q be any problem in NP. Since Q_1 is NP-hard, by the definition, $Q \leq_m^p Q_1$. This, together with $Q_1 \leq_m^p Q_2$ and Lemma 1.4.3, implies $Q \leq_m^p Q_2$. Since Q is an arbitrary problem in NP, we conclude that the problem Q_2 is NP-hard. □

Since we already know that the SATISFIABILITY problem is NP-hard (Theorem 1.4.2) and that the SATISFIABILITY problem is polynomial-time reducible to the INDEPENDENT-SET(D) problem, Corollary 1.4.4 enables us to conclude directly that the INDEPENDENT-SET(D) is NP-hard. In consequence, it is unlikely that the INDEPENDENT-SET(D) problem can be solved in polynomial-time.

The idea of Corollary 1.4.4 has established an extremely useful working system for proving the computational hardness for problems: suppose we want to show a given problem Q is computationally hard, we may pick a known NP-hard problem Q' (well, we already have two here) and show $Q' \leq_m^p Q$. If we succeed, then the problem Q is NP-hard and hence it is unlikely that Q can be solved in polynomial time. Moreover, now the problem Q can be added to the list of NP-hard problems, which may be helpful later in proving the NP-hardness of other problems. In the last four decades, people have successfully used this technique and derived NP-hardness for over thousands of problems. Thus, all these numerous problems are unlikely to be solved in polynomial time. Of course, this working system is completely based on the following assumption:

Working Conjecture in NP-completeness Theory

P$\neq$ NP, that is, there are problems in NP that are not solvable in polynomial time.

No proof for this working conjecture has been derived. In fact, very little is known for a proof for the conjecture. On the other hand, the conjecture is strongly believed by most people working in computational sciences.

In the following, we give a list of some NP-complete problems, whose NP-hardness will be used in our latter discussion. The proof for the NP-hardness for these problems can be found in Garey and Johnson [52]. For those problems that also have an optimization version, we attach a "(D)" to the end of the problem names to indicate that these are decision problems.

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 $\sum_{a_i \in S_1} a_i = \sum_{a_j \in S_2} a_j$?

GRAPH COLORING (D)

Given a graph G and an integer k, can the vertices of G be colored with at most k colors so that no two adjacent vertices in G are colored with the same color?

GRAPH EDGE COLORING (D)

Given a graph G and an integer k, can the edges of G be colored with at most k colors so that no two edges sharing a common vertex are colored with the same color?

PLANAR GRAPH INDEP-SET (D)

Given a planar graph G and an integer k, is there a subset I of at least k vertices of G such that no two vertices in I are adjacent?

PLANAR GRAPH VERTEX-COVER (D)

Given a planar graph G and an integer k, is there a subset C of at most k vertices of G such that every edge in G has at least one end in C?

HAMILTONIAN CIRCUIT

Given a graph G, is there a simple cycle in G that contains all vertices of G?

EUCLIDEAN TRAVELING SALESMAN (D)

Given a set S of n points in the plane and a real number k, is there a tour of length bounded by k that visits all points in S?

MAXIMUM CUT (D)

Given a graph G and an integer k, is there a partition of the vertices of G into two sets V_1 and V_2 such that the number of edges with one end in V_1 and the other end in V_2 is at least k?

3-D MATCHING

Given a set of triples $M = X \times Y \times Z$, where X, Y, and Z are disjoint sets having the same number q of elements, is there a subset M' of M of q triples such that no two triples in M' agree on any coordinate?