Chapter 1

Introduction

This chapter starts with a number of samples of optimization problems and introduces the formal definition for optimization problems. Necessary background in computer algorithms will be reviewed. We then discuss in detail two important sample optimization problems: the Minimum Spanning Tree problem and the Matrix-Chain Multiplication problem. Algorithms and analysis are given for the problems. Two basic techniques, the greedy method and the dynamic programming method, are illustrated in the study of these two problems. Finally, we give a brief discussion on NP-completeness theory, which will play an important role throughout the book.

1.1 Optimization problems

Most computational optimization problems come from practice in industry and other fields. 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 format. Each instance is associated with a set of solutions such that each solution has a value given the instance. Solving the optimization problem is concerned with finding for each given instance a best (or optimal) solution which should have either the largest or the smallest associated value, depending on the description of the optimization problem.

Let us start with some sample problems.

The most famous optimization problem is the Traveling Salesman problem (or simply TSP).

Traveling Salesman (TSP)
INTRODUCTION

Given the cities in a territory 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 into instances of the LINEAR PROGRAMMING problem.

LINEAR PROGRAMMING (LP)

Given a vector \((c_1, \ldots, c_n)\) of real numbers, and a set of linear constraints

\[
\begin{align*}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & \geq a_1 \\
& \quad \cdots \\
a_{r1}x_1 + a_{r2}x_2 + \cdots + a_{rn}x_n & \geq a_r \\
b_{11}x_1 + b_{12}x_2 + \cdots + b_{1n}x_n & \leq b_1 \\
& \quad \cdots \\
b_{s1}x_1 + b_{s2}x_2 + \cdots + b_{sn}x_n & \leq b_s \\
d_{11}x_1 + d_{12}x_2 + \cdots + d_{1n}x_n & = d_1 \\
& \quad \cdots \\
d_{t1}x_1 + d_{t2}x_2 + \cdots + d_{tn}x_n & = d_t 
\end{align*}
\]

(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 \(c_1x_1 + \cdots + c_nx_n\).
OPTIMIZATION PROBLEMS

Both problems above are minimization problems. Below we give an example of maximization problem, which arises naturally in academic school administration.

**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 \( (\tau_i, \xi_i) \) indicating that the teacher \( \tau_i \) can teach the course \( \xi_i \), \( \tau_i \in T, \xi_i \in C, \) and \( i = 1, \ldots, n \), find a course assignment in which each teacher teaches at most one course and each course is taught by at most one teacher such that the maximum number of courses get taught.

Here an instance consists of the set of the pairs \( (\tau_i, \xi_i), \) \( i = 1, \ldots, n \), a solution to the instance is a subset of the pairs in which each teacher appears at most once and each course appears at most once, the value associated with the solution is the number of pairs in the subset, and the objective is to find such a subset with the maximum number of pairs.

Formally, an optimization problem can be given as follows.

**Definition 1.1.1** An optimization problem \( Q \) is a 4-tuple \( \langle I_Q, S_Q, f_Q, optQ \rangle \), 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 a set of solutions to \( x \), \( f_Q \) is the objective function such that for each pair \( x \in I_Q \) and \( y \in S_Q(x) \), \( f_Q(x, y) \) is a real number, and \( optQ \in \{ \text{max, min} \} \) specifies the problem to be a maximum problem or a minimum problem.

Solving the optimization problem \( Q \) is that given an 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 \( optQ \)) among 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 by a weighted graph, in which each vertex is for a node in the network and each edge is for 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)**
\[ I_Q: \text{the set of all weighted graphs } G \]
\[ S_Q: S_Q(G) \text{ is the set of all spanning trees of the graph } G \]
\[ f_Q: f_Q(G, T) \text{ is the weight of the spanning tree } T \text{ of } G. \]
\[ \text{opt}_Q: \min \]

In path scheduling or network communication, we often need to find the shortest path from a given position to another specified position. This problem is formulated as the Shortest Path problem.

**Shortest Path**

\[ I_Q: \text{the set of all weighted graphs } G \text{ with two specified vertices } u \text{ and } v \text{ in } G \]
\[ S_Q: S_Q(G) \text{ is the set of all paths connecting } u \text{ and } v \text{ in } G \]
\[ f_Q: f_Q(G, u, v, P) \text{ is the length of the path } P \text{ (measured by the weight of edges) connecting } u \text{ and } v \text{ in } G \]
\[ \text{opt}_Q: \min \]

The next optimization problem takes its name from the following story: a thief robbing a safe finds a set of items of varying size and value that he could steal, but has only a small knapsack of capacity \( B \) which he can use to carry the goods. Now 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 resource, we want to get the maximum reward.

**Knapsack**

\[ I_Q: \text{the set of tuples } T = \{s_1, \ldots, s_n; v_1, \ldots, v_n; B\}, \text{ where } s_i \text{ and } v_i \text{ are the size and value of the } i\text{th item, respectively, and } B \text{ is the knapsack size} \]
\[ S_Q: S_Q(T) \text{ is a subset } S \text{ of pairs of form } (s_i, v_i) \text{ in } T \text{ such that the sum of all } s_i \text{ in } S \text{ is not larger than } B \]
\[ f_Q: f_Q(T, S) \text{ is the sum of all } v_i \text{ in } S \]
\[ \text{opt}_Q: \max \]

The following optimization problem arises in job scheduling on parallel processing systems. Suppose that we have a set of jobs \( J_1, \ldots, J_n \), where the
OPTIMIZATION PROBLEMS

processing time of job $J_i$ (on a single processor) is $t_i$, and a set of identical processors $P_1, \ldots, P_m$. Our objective is to assign the jobs to the processors so that the completion time of all jobs is minimized. This problem can be formulated as follows.

**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,

$$f_Q(T, P) = \max_i \{\sum_{j \in T_i} t_j\}$$

$opt_Q$: min

The following optimization problem has obvious application in scientific computing. Suppose that we use the ordinary matrix multiplication method to compute a matrix product $M_1 \times M_2$ of two matrices, where $M_1$ is a $p \times q$ matrix and $M_2$ is a $q \times r$ matrix. Then we need to perform $pqr$ element multiplications, and the resulting product is a $p \times r$ matrix. Now suppose that instead of multiplying two matrices, we need to compute the product $M_1 \times M_2 \times \cdots \times M_n$ of a chain of more than two matrices, $n > 2$, where $M_i$ is a $d_{i-1} \times d_i$ matrix (verify that this condition guarantees the validity for the matrices to be multiplied.) Since matrix multiplication is associative, we can add balanced parentheses to the sequence and change the order of the multiplications without affecting the final product matrix. On the other hand, changing the order may make significant difference in terms of the total number of element multiplications needed to compute the final product matrix. For example, consider the product $M_1 \times M_2 \times M_3$. Suppose that both $M_1$ and $M_2$ are $100 \times 100$ matrices, while $M_3$ is a $100 \times 1$ matrix. Then to obtain the final product matrix, the order $(M_1 \times M_2) \times M_3$ needs 1,010,000 element multiplications, while the order $M_1 \times (M_2 \times M_3)$ requires only 20,000 element multiplications. Therefore, given a chain of matrices, it is desired to find the order that requires the minimum number of element multiplications

**Matrix-Chain Multiplications**

$I_Q$: the set of tuples $T = \{d_0, d_1, \ldots, d_n\}$, where suppose that the $i$th matrix $M_i$ is a $d_{i-1} \times d_i$ matrix
INTRODUCTION

\( S_Q \): \( S_Q(T) \) is the set of the sequences \( S \) that are the sequence \( M_1 \times \cdots \times M_n \) with proper balance parentheses inserted, indicating an order of multiplications of the sequence

\( f_Q \): \( f_Q(T, S) \) is equal to the number of element multiplications needed in order to compute the final product matrix according to the order given by \( S \)

\( \text{opt}_Q \): \( \text{min} \)

We close this section with a graph optimization problem, which will play an important role in our discussion.

\textbf{Independent Set}

\( I_Q \): the set of undirected graphs \( G = (V, E) \)

\( S_Q \): \( S_Q(G) \) is the set of subsets \( S \) of \( V \) such that no two vertices in \( S \) are adjacent

\( f_Q \): \( f_Q(G, S) \) is equal to the number of vertices in \( S \)

\( \text{opt}_Q \): \( \text{max} \)

1.2 Algorithmic preliminary

The objective of this book is to discuss how optimization problems are solved using computer programs, which will be described by computer algorithms. The design and analysis of computer algorithms have been a very active research area in computer science since the introduction of the first modern computer in middle 1900’s. In this section, we briefly review the fundamentals for 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, and Rivest [28], and Knuth [83, 84].

\textbf{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. Since the introduction of modern computers in middle 1900’s, however, it has become popular to refer “algorithms” to “computer algorithms”. Informally, an \textit{algorithm} is a high level description of a computer program, which is a step-by-step specification of a 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/to computer memory. The language used in this book to describe algorithms is similar to the PASCAL programming language with certain grammatical flexibilities.

We say that an algorithm $A$ solves an optimization problem $Q = \langle I_Q, S_Q, f_Q, \text{opt}_Q \rangle$ 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 satisfies the condition $f_Q(x, y) = \text{opt}_Q\{f_Q(x, z) \mid z \in S_Q(x)\}$).

Encodings

To study the computational complexity of algorithms, 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 $\Sigma$. 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 for the graphs organized in row major in a sequence of numbers. Now suppose that the finite alphabet $\Sigma$ is fixed, then we can encode each sequence in $\Sigma$ into a binary sequence as follows. Let $q$ be the number of symbols in $\Sigma$, then each symbol in $\Sigma$ can be easily encoded into a distinct binary string of length $\lceil \log q \rceil$. Therefore, each sequence of length $n$ in $\Sigma$ 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 covert a sequence in $\Sigma$ into the corresponding binary sequence and vice versa. It is convincing that in general, input instances and solutions of an optimization problem, even they are compound objects such as a polygon, a graph, or a formula, can be effectively and efficiently encoded into binary sequences.

Therefore, we will use 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 $A$ is an algorithm solving an optimization problem $Q$. It is reasonable to assume that for input instances of large size, the algorithm $A$ spends more computational time. Thus, we will evaluate the performance of the algorithm $A$ 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 $A$ uses to find an optimal solution for a given input instance. There are several reasons for this. First, the 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 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?”

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 any $g \in C_1$, there is a constant $c_g$ such that $t(n) \geq c_g g(n)$ for all but a finite number of $n$'s. Roughly speaking, $O(t(n))$ is the class of functions that are at most as large as $t(n)$.

- $o(t(n))$: the class $C_2$ of functions such that for any $g \in C_2$, $\lim_{n \to \infty} g(n)/t(n) = 0$. Roughly speaking, $o(t(n))$ is the class of functions that are less than $t(n)$.

- $\Omega(t(n))$: the class $C_3$ of functions such that for any $g \in C_3$, there is a constant $c_g$ such that $t(n) \leq c_g g(n)$ for all but a finite number of $n$'s. Roughly speaking, $\Omega(t(n))$ is the class of functions which are at least as large as $t(n)$.

- $\omega(t(n))$: the class $C_4$ of functions such that for any $g \in C_4$, $\lim_{n \to \infty} t(n)/g(n) = 0$. Roughly speaking, $\omega(t(n))$ is the class of functions that are larger than $t(n)$. 
• $\Theta(t(n))$: the class $C_\Theta$ of functions such that for any $g \in C_\Theta$, $g(n) = O(t(n))$ and $g(n) = \Omega(t(n))$. Roughly speaking, $\Theta(t(n))$ is the class of functions which are of the same order as $t(n)$.

**Complexity of algorithms**

There are two types of analyses of algorithms: worst case and expected case. For the worst case analysis, we seek the maximum amount of time used by the algorithm for all possible inputs. For the expected case analysis we normally assume a certain probabilistic distribution on the input and study the performance of the algorithm for any input 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 on the input instance.

**Definition 1.2.1** Let $A$ be an algorithm solving an optimization problem $Q$ and let $f(n)$ be a function. The **time complexity** of algorithm $A$ 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 $A$ is bounded by $f'(n)$ for all input instances of size $n$.

Based on these preparations, now we are ready for presenting an important terminology.

**Definition 1.2.2** An algorithm $A$ is a **polynomial-time algorithm** if there is a fixed constant $c$ such that the time complexity of the algorithm $A$ 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 length, as long as these schemes and definitions define input lengths 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. 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, a 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 if large numbers are present in an input instance. 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. The standard definition of the input length regards the input length as \( \lceil \log n \rceil = O(\log n) \), which is not polynomially related to the quantity 1.

**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 should assume that the other unimportant 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 \( Q = (I_Q, S_Q, f_Q, \text{opt}_Q) \) 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 PROBLEMS

sample optimization problems, and introduce two important techniques in designing optimization algorithms. We present an algorithm, using greedy method, to solve the Minimum Spanning Tree problem, and an algorithm, using dynamic programming method, to solve the Matrix-Chain Multiplication problem.

1.3.1 Minimum spanning tree

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 a 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 condition 1. Now let us consider the second condition. Since we have no idea about any minimum spanning trees (we are being constructing 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 my new subtree $T_1 + e$ small.” This presents the main idea for an important optimization technique, which is called the greedy method. In general, greedy method always makes the choice that looks best at the moment in the hope that this choice will lead to a best final solution for the problem.

It is conceivable that the greedy method does not always yield best solutions for a given problem. However, for quite a few optimization problems,
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 $T_1$.

it does. The **Minimum Spanning Tree** problem fortunately belongs to this class, as shown in 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 $G$.

**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 $T_0$. Let $e = (u, v)$, 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 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 $\text{weight}(e) \leq \text{weight}(e')$. 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.
SAMPLE PROBLEMS

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 at most $n - 1$ times should give us a subtree $T$ of $n - 1$ edges which should be entirely contained in a minimum spanning tree. In other words, the tree $T$ itself should be a minimum spanning tree.

What remains is how we start the above process, i.e., what is the first such a subtree. But 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.

This algorithm is called Prim’s Algorithm and due to R. C. Prim [107]. We give some more explanations for 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 for 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 that supports each of the following operations in time $O(\log m) = O(\log n)$: find-
Algorithm. PRIM

1. pick any vertex \( v \) and make it an in-tree vertex;
2. for each edge \( e \) incident on \( v \) do make \( e \) a fringe-edge;
3. let the subtree \( T_1 \) be an empty tree;
4. loop \( n - 1 \) times
   - pick a fringe-edge \( e = (u, v) \) of minimum weight, where \( u \) is
     in-tree and \( v \) is out-tree;
   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

ing the minimum weight edge, changing the weight for an edge (suppose we
make the weight \(+\infty\) for each edge that is not a fringe-edge). Then since
each edge is selected as the fringe-edge of minimum weight as 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 running time of the algorithm is bounded by \( O(m \log n) \).
More detailed description of this improvement can be found in [28].

1.3.2 Matrix-chain multiplication

In this subsection, we describe another important optimization technique:
dynamic programming method. We illustrate the technique by presenting
an efficient algorithm for the **Matrix-Chain Multiplication** problem. Recall
that each input 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 \( M_1, \ldots, M_n \), 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
SAMPLE PROBLEMS

to first compute the product $P_1 = M_1 \times \cdots \times M_k$ and the product $P_2 = M_{k+1} \times \cdots \times M_n$, 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 number of element multiplications for the best orders for computing the products $P_1$ and $P_2$? We notice that for the products $P_1$ and $P_2$, the corresponding matrix chains are shorter than $n$. Thus, we can apply the same method recursively to find the numbers of element multiplications for the two products. The numbers of element multiplications found by the recursive process plus the number $d_0 d_k d_n$ give us the total number of element multiplications for this best 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 "×" 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 $\text{ind}$ to record the index for the best factoring we have seen so far, and use $\text{num}$ to record the number of element multiplications based on this factoring.

What is the time complexity for this algorithm? Let $T(h)$ be the running time of the algorithm \textbf{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 $k$th "×" in the chain, which induces the recursive executions of the algorithm on a chain of $k$ matrices and on a chain of $h - k$ matrices. Thus, we have

$$T(h) \geq [T(1) + T(h - 1)] + [T(2) + T(h - 2)] + \cdots + [T(h - 1) + T(1)]$$
$$= 2[T(1) + T(2) + \cdots + T(h - 1)]$$
$$\geq hT(h/2)$$

From the relation $T(h) \geq hT(h/2)$, it is easy to see that $T(h) \geq h \log h - 1$. 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 \textbf{Recursive-MCM} is at least
Figure 1.3: Recursive algorithm for Matrix-Chain Multiplication

\[ \Omega(n^{\log n - 1}) \]

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 the \( i \)th "\( \times \)". for \( 2 \leq i \leq 6 \). It is the repeatedly applications of the recursive process on the same subchain that make the algorithm time-consuming.

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. Since to compute the values for NUM[i, j] and IND[i, 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: \( \text{NUM}[i, i] = 0 \), and \( \text{IND}[i, i] \) has no meaning.

The algorithm is presented in Figure 1.5.

The analysis of the algorithm \textbf{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 \( O(n^2) \). This concludes that the problem \textsc{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 algorithm \textbf{Dyn-Prog-MCM}. With the values of the arrays \text{NUM} and \text{IND} being available, by reading the value \( \text{IND}[1, n] \), suppose \( \text{IND}[1, n] = k \), we know that input matrix chain \( M_1 \times \cdots \times M_n \) should be factored at the index \( k \). Now with the values \( \text{IND}[1, k] \) and \( \text{IND}[k + 1, n] \), 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 \text{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.

Algorithm \textbf{Dyn-Prog-MCM} illustrates the principle of an important technique for optimization algorithms — the \textit{dynamic programming method}. A dynamic programming algorithm stores intermediate results and/or so-
Algorithm. Dyn-Prog-MCM

1. for $i = 1$ to $n$ do \( \text{NUM}[i, i] = 0; \)
2. for $\text{diag} = 1$ to $n - 1$ do
   for $i = 1$ to $n - \text{diag}$ do
     $j = i + \text{diag}$;
     \( \text{num} = \infty; \)
     for $k = i$ to $j - 1$ do
       if $\text{num} > \text{NUM}[i, k] + \text{NUM}[k + 1, j] + d_{i-1}d_kd_j$
       then $\text{num} = \text{NUM}[i, k] + \text{NUM}[k + 1, j] + d_{i-1}d_kd_j$;
       $\text{IND}[i, j] = k$
     $\text{NUM}[i, j] = \text{num}$;

Figure 1.5: Dynamic programming for Matrix-Chain Multiplication

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 characterizing 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 [50].

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, as a matter of 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 such a problem for which each input instance only needs to take one of the two possible answers—“yes” or “no”. An input instance taking the answer “yes”
will be called a \textit{yes-instance} for the problem, and an input instance taking the answer “no” will be called a \textit{no-instance} for the problem.

The following \textsf{Satisfiability} (or shortly SAT) problem is a decision problem.

\textbf{Satisfiability (Sat)}

Given a boolean formula $F$ in the conjunctive normal form, is there an assignment to the variables in $F$ so that the formula $F$ has value TRUE?

Thus, every boolean formula in the conjunctive normal form that is satisfiable (i.e., it can take value TRUE on some assignment) is a yes-instance for the \textsf{Satisfiability} problem, while every boolean formula in the conjunctive normal form that is not satisfiable is a no-instance for the \textsf{Satisfiability} problem.

An optimization problem $Q$ can be converted into a decision problem by introducing a parameter, which is used to compare with the optimal value of an input instance. For example, a decision version of the \textsf{Traveling Salesman} problem can be formulated as follows. An input instance of the decision problem is of form $(G, k)$, where $G$ is a weighted complete graph and $k$ is an integer. The question the decision problem asks on the 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 $A$ \textit{accepts} a decision problem $Q$ if on every yes-instance $x$ of $Q$, the algorithm $A$ stops at a “yes” state (i.e., “accepts” $x$), while on all other inputs $x'$ (including the inputs that do not encode an input instance of $Q$), the algorithm $A$ stops at a “no” state (i.e., “rejects” $x'$).

\textbf{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 $P$.

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 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 accepted 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 the problem $Q$, then there is a binary string $y$ of length bounded by $p(|x|)$ such that on input $(x, y)$ the algorithm $\mathcal{A}$ stops at a “yes” state;

2. If $x$ is not a yes-instance for the problem $Q$, then for any binary string $y$ of length bounded by $p(|x|)$, on input $(x, y)$ the algorithm $\mathcal{A}$ stops at a “no” state.

Thus, a problem $Q$ in NP is the one whose yes-instances $x$ can be easily (i.e., in polynomial time) checked (by the algorithm $\mathcal{A}$) when a short (i.e., bounded by the polynomial $p$ of $|x|$) proof (i.e., $y$) 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 not a yes-instance for the problem $Q$, then no matter what hint $y$ is given, the algorithm $\mathcal{A}$ cannot be fooled to conclude “yes”.

Therefore, 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 derived. Alternatively, the class NP can be defined to be the set of those
NP-COMPLETENESS

decision problems that can be accepted by *nondeterministic polynomial-time algorithms*, which can always correctly guess the proof. Therefore, on an input $x$ that is a yes-instance, the nondeterministic polynomial-time algorithm guesses a correct proof $y$, checks the pair $(x,y)$, and accepts $x$; while on an input $x$ that is not a yes-instance, with any guessed proof $y$, 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: given an instance $(G,k)$, where $G$ is a weighted complete graph and $k$ is an integer, it is asked 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 not a yes-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 used in Definition 1.4.2 that 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 no decision problems. However, based on the characterization of “having a short hint”, 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 $Q = (I_Q,S_Q,f_Q,\text{opt}_Q)$ is an *NP optimization* (or shortly NPO) problem if there is a polynomial $p(n)$ such that for any 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 come back to NP-completeness theory. A very important concept in NP-completeness theory is the reducibility, which is defined as follows.

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

The relation $Q_1 \leq^p_m Q_2$ 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 $Q_1 \leq^p_m Q_2$ sets a lower bound for the computational complexity for the problem $Q_2$ in terms of the problem $Q_1$, and also sets an upper bound for the computational complexity for the problem $Q_1$ in terms 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 $Q_1 \leq^p_m Q_2$ 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 $A_1$ 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 $A_2$ 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. Now an algorithm $A_3$ for the problem $Q_1$ can be derived as follows. On an input $x$, $A_3$ first computes $r(x)$ by calling the algorithm $A_1$ as a subroutine. This takes time $O(|x|^c)$. Note since the running time of $A_1$ is bounded by $O(|x|^c)$, the length $|r(x)|$ of the output of $A_1$ is also bounded by $O(n^c)$. Now the algorithm $A_3$ calls the algorithm $A_2$ 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 $A_3$ 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 $A_3$ correctly accepts the decision problem $Q_1$. Moreover, since the running time of the algorithm $A_3$ 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 for the polynomial-time reduction by showing how the Satisfiability problem is polynomial-time reduced to the following decision version of the Independent Set problem.
NP-COMPLETENESS

DECISION-INDEP-SET

Given a graph \( G \) and an integer \( k \), is there a set \( S \) of at least \( k \) vertices in \( G \) such that no two vertices in \( S \) are adjacent?

The algorithm \( \mathcal{A} \) computing the reduction function \( r \) from the SATISFIABILITY problem to the DECISION-INDEP-SET problem works as follows. Let \( F = C_1 \land C_2 \land \cdots \land C_m \) be an instance of the SATISFIABILITY problem, where each \( C_i \) (called a clause) is a disjunction \( C_i = (l_{i,1} \lor l_{i,2} \lor \cdots \lor l_{i,n_i}) \) 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 contradict to each other, i.e., one is the negation of the other. Now, for the instance \( F \) of the SATISFIABILITY problem, the value of the function \( r(F) \) is \( (G_F, m) \), which is an instance for the DECISION-INDEP-SET problem. It is easy to see that given the instance \( F \) for SATISFIABILITY, the instance \((G_F, m)\) for DECISION-INDEP-SET 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 DECISION-INDEP-SET. Suppose that \( F \) is a yes-instance for SATISFIABILITY. Then there is an assignment \( \alpha \) to the variables in \( F \) that makes \( F \) \textsc{true}. Thus, for each clause \( C_i \), the assignment \( \alpha \) sets at least one literal appearance \( l_{i,h_i} \) in \( C_i \) \textsc{true}. Now pick 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 are adjacent by the construction of the graph \( G_F \) since \( 1 \) they are not in the same clause and \( 2 \) the assignment \( \alpha \) cannot set two contradicting literals both \textsc{true}. Therefore, \( r(F) = (G_F, m) \) is a yes-instance for DECISION-INDEP-SET.

Now suppose that \( r(F) = (G_F, m) \) is a yes-instance for DECISION-INDEP-SET. Let \( S = \{v_1, \ldots, v_m\} \) be a set of \( m \) vertices in \( G_F \) such that no two vertices in \( S \) are adjacent. 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 \( l_{i,h_i} \) corresponding to a vertex in the set \( S \). Moreover, no two \( l_{i,h_i} \) and \( l_{j,h_j} \) of these \( m \) literal appearances contradict each other — otherwise the two corresponding vertices in \( S \) would have been adjacent. Therefore, an assignment \( \alpha \) to the variables in \( F \) can be constructed that sets all these literal appearances \( l_{i,h_i} \) \textsc{true}: if a boolean variable \( x \) is one of the literal \( l_{i,h_i} \), then \( \alpha(x) = \textsc{true} \); if the negation \( \overline{x} \) of
a boolean variable \( x \) is one of the literals \( l_{i,h_i} \) then \( \alpha(x) = \text{false} \); if neither \( x \) nor \( \overline{x} \) is any of the literals \( l_{i,h_i} \), then \( \alpha \) sets \( x \) arbitrarily. Note that the assignment \( \alpha \) sets at least one literal in each clause in \( F \text{ true} \), thus makes the formula \( F \text{ true} \). Consequently, \( F \) is a yes-instance for \textsc{Satisfiability}.

This completes the polynomial-time reduction from the \textsc{Satisfiability} problem to the \textsc{Decision-Indep-Set} problem.

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

\textbf{Theorem 1.4.2 (Cook's Theorem)} \textit{Every decision problem in the class NP is polynomial-time many-one reducible to the \textsc{Satisfiability} problem.}

\textbf{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 computation models. A more complete proof for the theorem can be found in Garey and Johnson [50].

Suppose that \( Q \) is a decision problem in NP, and that \( A \) 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 \( y_x \) such that the algorithm \( A \) accepts \( (x, y_x) \), and if \( x \) is not a yes-instance, then for any binary string \( y \), the algorithm \( A \) 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 reduced to the \textsc{Satisfiability} problem.

The algorithm \( A \) 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 can be made in the conjunctive normal form. The input to the formula \( F \) is of the form \( (x, y) \) such that for any assignment \( x_0 \) and \( y_0 \) to \( x \) and \( y \), respectively, \( F(x_0, y_0) = \text{true} \) if and only if the algorithm \( A \) accepts the pair \( (x_0, y_0) \). Now for a given instance \( x_0 \) for the problem \( Q \), the instance for \textsc{Satisfiability} is \( F_0 = F(x_0, * ) \). That is, the formula \( F_0 \) is obtained from the formula \( F \) with the first parameter \( x \) assigned by the value \( x_0 \). 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 \( A \) accepts \( (x_0, y_0) \). Thus, on this \( y_0 \), the formula \( F_0(y_0) = F(x_0, y_0) \) gets value \text{true}, i.e., the formula \( F_0 \) is satisfiable thus is a yes-instance for \textsc{Satisfiability}. On the other
NP-COMPLETENESS

hand, if \( x_0 \) is not a yes-instance, then the algorithm \( A \) 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 thus is not a yes-instance for SATISFIABILITY.

Thus, the problem \( Q \) in NP is polynomial-time reduced to the SATISFIABILITY problem. Since \( Q \) is an arbitrary problem in NP, the theorem is proved. \( \square \)

According to the definition of the polynomial-time reduction, Theorem 1.4.2 shows that no problems in the class NP is essentially harder than the SATISFIABILITY problem. This hints a lower bound on the computational complexity for the SATISFIABILITY problem. Motivated by this theorem, we introduce 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 many very 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, but all failed. 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 in the development of the NP-completeness theory.

However, how do we show the NP-hardness for 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 generalized: 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 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 three decision problems. If $Q_1 \leq^p_m Q_2$ and $Q_2 \leq^p_m Q_3$, then $Q_1 \leq^p_m Q_3$.

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 $Q_3$.

This shows that $Q_1 \leq^p_m Q_3$. □

Corollary 1.4.4 Let $Q_1$ and $Q_2$ be three decision problems. Suppose that the problem $Q_1$ is NP-hard and that $Q_1 \leq^p_m Q_2$, 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, $Q \leq^p_m Q_1$. This together with $Q_1 \leq^p_m Q_2$ and Lemma 1.4.3, implies $Q \leq^p_m Q_2$. 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 Decision-Independent Set problem, Corollary 1.4.4 enables us to conclude directly that the Decision-Independent Set is NP-hard. In consequence, it is unlikely that the Decision-Independent Set problem can be solved in polynomial-time.

The idea of Corollary 1.4.4 has established an extremely useful working system for proving 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 $Q' \leq^p_m Q$. If we succeed, then the problem $Q$ is NP-hard, thus 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 NP-hardness for other problems. In the last two decades, people have successfully used this technique and derived NP-hardness for over thousands of problems. Thus, all these thousands of 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. However, the conjecture is strongly believed by most people working in computer science.

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 [50]. 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 $\sum_{i \in S_1} a_i = \sum_{j \in S_2} a_j$?

**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 $S$ of at least $k$ vertices of $G$ such that no two vertices in $S$ are adjacent?

**Planar Graph Vertex-Cover (D)**

Given a planar graph $G$ and an integer $k$, is there a subset $S$ of at most $k$ vertices of $G$ such that every edge in $G$ has at least one end in $S$?
Hamiltonian Circuit
Given a graph $G$ of $n$ vertices, is there a simple cycle in $G$ that contains all vertices?

Euclidean Traveling Salesman (D)
Given a set $S$ of $n$ points in the plane and an integer $k$, 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 of $M'$ of $M$ of $q$ triples such that no two triples in $M'$ agree in any coordinate?