# Introduction to Tractability and Approximability of Optimization Problems

JIANER CHEN

Computer Science Department
Texas A&M University

July 8, 2003

# 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)

> 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
>
> $$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n & \geq & a_1 \\ & \cdots\cdots & \\ a_{r1}x_1 + a_{r2}x_2 + ... + a_{rn}x_n & \geq & a_r \\ b_{11}x_1 + b_{12}x_2 + ... + b_{1n}x_n & \leq & b_1 \\ & \cdots\cdots & \\ b_{s1}x_1 + b_{s2}x_2 + ... + b_{sn}x_n & \leq & b_s \\ d_{11}x_1 + d_{12}x_2 + ... + d_{1n}x_n & = & d_1 \\ & \cdots\cdots & \\ d_{t1}x_1 + d_{t2}x_2 + ... + d_{tn}x_n & = & d_t \end{array} \tag{1.1}$$
>
> 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 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 $\langle I_Q, S_Q, f_Q, opt_Q \rangle$, 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 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 $opt_Q \in \{\max, \min\}$ 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 $opt_Q$) 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)

$I_Q$: the set of all weighted 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

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

$I_Q$: the set of all weighted graphs $G$ with two specified vertices $u$ and $v$ in $G$

$S_Q$: $S_Q(G)$ is the set of all paths connecting $u$ and $v$ in $G$

$f_Q$: $f_Q(G, u, v, P)$ is the length of the path $P$ (measured by the weight of edges) connecting $u$ and $v$ in $G$

$opt_Q$: 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

$I_Q$: the set of tuples $T = \{s_1, \ldots, s_n; v_1, \ldots, v_n; B\}$, where $s_i$ and $v_i$ are for the size and value of the $i$th 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. 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,

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

$opt_Q$: 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

$S_Q$: $S_Q(T)$ 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 $S$

$opt_Q$: min

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

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 $S$

$opt_Q$: 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].

### 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 *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 $\mathcal{A}$ *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 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 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 $\mathcal{A}$ is an algorithm solving an optimization problem $Q$. It is reasonable to assume that for input instances of large size, the algorithm $\mathcal{A}$ spends more computational time. Thus, 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 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 $C_5$ of functions such that for any $g \in C_5$, $g(n) = O(t(n)$ and $g(n) = \Omega(t(n))$. Roughly speaking, $\Theta(t(n))$ is the class of functions which are 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 $\mathcal{A}$ 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 $n$.

Based on these preparations, 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 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 $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ 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 *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,

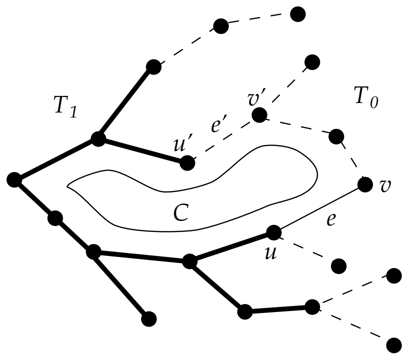



Figure 1.1: A cycle $C$ in $T_0 + e$, where heavy lines are for edges in the constructed subtree $T_1$, and dashed lines are for edges in the minimum spanning tree $T_0$ that are not in $T_1$.

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 $weight(e) \leq weight(e')$. 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. $\square$

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 undate 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 $\bar{e}$ incident on $v$ **do**
  **if** $\bar{e}$ is a fringe-edge
  **then** make $\bar{e}$ a cycle-edge
  **else if** $\bar{e}$ is an unseen-edge **then** make $\bar{e}$ 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 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 "$\times$" 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.

What is the time complexity for 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 $k$th "$\times$" 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

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

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 **Recursive-MCM** is at least

**Algorithm. Recursive-MCM**$(i, j)$

1. **if** $i \geq j$ **then** return 0; Stop;
2. $num = \infty$; $ind = 0$;
3. **for** $k = i$ **to** $j - 1$ **do**

3.1 recursively compute the minimum number $q_1$ of element multiplications for computing the product $M_i \times \cdots \times M_k$: call Recursive-MCM$(i, k)$;

3.2 recursively compute the minimum number $q_2$ of element multiplications for computing the product $M_{k+1} \times \cdots \times M_j$: call Recursive-MCM$(k + 1, j)$;

3.3 **if** $num > q_1 + q_2 + d_{i-1}d_kd_j$
**then** $ind = k$; $num = q_1 + q_2 + d_{i-1}d_kd_j$;

4. return $num$ and $ind$.

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

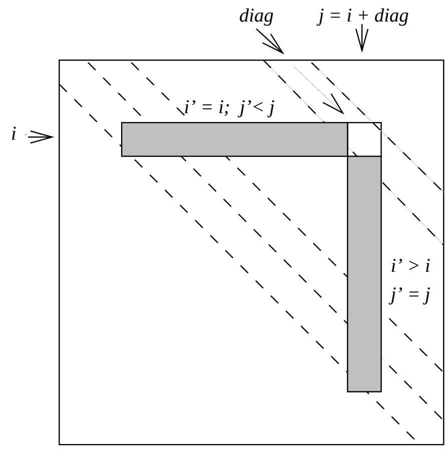



Figure 1.4: The order for computing the elements for NUM and IND.

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.

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 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$[k+1, n]$, 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.

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 so-

**Algorithm. Dyn-Prog-MCM**

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

Figure 1.5: Dynamic programming for Matrix-Chain Multiplication

lutions 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 [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 *yes-instance* for the problem, and an input instance taking the answer "no" will be called a *no-instance* for the problem.

The following *Satisfiability* (or shortly SAT) problem is a decision problem.

> 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 SATISFIABILITY problem, while every boolean formula in the conjunctive normal form that is not satisfiable is a no-instance for the 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 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 $\mathcal{A}$ *accepts* a decision problem $Q$ if on every yes-instance $x$ of $Q$, the algorithm $\mathcal{A}$ 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 $\mathcal{A}$ stops at a "no" state (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.

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 $A$) 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 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 $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ is an *NP optimization* (or shortly NPO) problem if there is a polynomial $p(n)$ such that for 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 $Q_1 \leq_m^p Q_2$) 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 $Q_1 \leq_m^p Q_2$ 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 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 $Q_1 \leq_m^p Q_2$ 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. Now an algorithm $\mathcal{A}_3$ for the problem $Q_1$ can be derived 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 $O(n^c)$. Now 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 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.

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 $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ be an instance to 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 contradict 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 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 $F$ TRUE. Thus, for each clause $C_i$, the assignment $\alpha$ sets at least one literal appearance $l_{i,h_i}$ in $C_i$ 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 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 $l_{i,h_i}$ TRUE: if a boolean variable $x$ is one of the literal $l_{i,h_i}$, then $\alpha(x) = $ TRUE; if the negation $\overline{x}$ of a boolean variable $x$ is one of the literal $l_{i,h_i}$ then $\alpha(x) = $ 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 $F$ TRUE, thus makes the formula $F$ TRUE. Consequently, $F$ is a yes-instance for SATISFIABILITY.

This completes the polynomial-time reduction from the SATISFIABILITY problem to the DECISION-INDEP-SET 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 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 $\mathcal{A}$ 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 $y_x$ such that the algorithm $\mathcal{A}$ accepts $(x, y_x)$, and if $x$ is not a yes-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 reduced 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 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) = $ TRUE if and only if the algorithm $\mathcal{A}$ accepts the pair $(x_0, y_0)$. Now for a given instance $x_0$ for the problem $Q$, the instance for SATISFIABILITY is $F_0 = F(x_0, *)$. That is, the formula $F_0$ is obtained from the formula $F$ with the first paramenter $x$ 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 $\mathcal{A}$ accepts $(x_0, y_0)$. Thus, on this $y_0$, the formula $F_0(y_0) = F(x_0, y_0)$ gets value TRUE, i.e., the formula $F_0$ is satisfiable thus is a yes-instance for SATISFIABILITY. On the other hand, if $x_0$ is not a yes-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 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 $Q_1 \leq_m^p Q_2$ 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 $Q_3$.

This shows that $Q_1 \leq_m^p Q_3$. □

**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 $Q_1 \leq_m^p Q_2$, 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 DECISION-INDEP-SET problem, Corollary 1.4.4 enables us to conclude directly that the DECISION-INDEP-SET is NP-hard. In consequence, it is unlikely that the DECISION-INDEP-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 $Q' \leq_m^p Q$. 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 $\sum_{a_i \in S_1} a_i = \sum_{a_j \in S_2} a_j$ ?

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 $S$ 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?

# Part I

# Tractable Problems

# Chapter 2

# Maximum Flow

We start by considering the following problem: suppose that we have built a network of pipes to transport oil in an area. Each pipe has a fixed capacity for pumping oil, measured by barrels per hour. Now suppose that we want to transport a very large quantity of oil from city $s$ to city $t$. How do we use the network system of pipes so that the oil can be transported in the shortest time?

This problem can be modeled by the MAXIMUM FLOW problem, which will be the main topic of this chapter. The network of pipes will be modeled by a directed graph $G$ with two distinguished vertices $s$ and $t$. Each edge in the graph $G$ is associated with an integer, indicating the capacity of the corresponding pipe. Now the problem is to assign each edge with a flow, less than or equal to its capacity, so that the maximum amount of flow goes from vertex $s$ to vertex $t$.

The MAXIMUM FLOW problem arises in many settings in operations research and other fields. In particular, it can be used to model liquids flowing through pipes, parts through assembly lines, current through electrical networks, information through communication networks, and so forth. Efficient algorithms for the problem have received a great deal of attention in the last three decades.

In this chapter, we introduce two important techniques in solving the MAXIMUM FLOW problem. The first one is called the *shortest path saturation* method. Two algorithms, Dinic's algorithm and Karzanov's algorithm, are presented to illustrate this technique. The second method is a more recently developed technique, called the *preflow* method. An algorithm based on the preflow method is presented in Section 2.3. Remarks on related topics and further readings on the MAXIMUM FLOW problem are also given.

## 2.1 Preliminary

We start with the formal definitions.

**Definition 2.1.1** A *flow network* $G = (V, E)$ is a directed graph with two distinguished vertices $s$ (the source) and $t$ (the sink). Each edge $[u, v]$ in $G$ is associated with a positive integer $cap(u, v)$, called the *capacity* of the edge. If there is no edge from vertex $u$ to vertex $v$, then we define $cap(u, v) = 0$.

**Remark.** There is no special restriction on the directed graph $G$ that models a flow network. In particular, we allow edges in $G$ to be directed into the source and out of the sink.

Intuitively, a flow in a flow network should satisfy the following three conditions: (1) the amount of flow along an edge should not exceed the capacity of the edge (capacity constraint); (2) a flow from a vertex $u$ to a vertex $v$ can be regarded as a "negative" flow of the same amount from vertex $v$ to vertex $u$ (skew symmetry); and (3) except for the source $s$ and the sink $t$, the amount of flow getting into a vertex $v$ should be equal to the amount of flow coming out of the vertex (flow conservation). These conditions are formally given in the following definition.

**Definition 2.1.2** A *flow* $f$ in a flow network $G = (V, E)$ is an integer-valued function on pairs of vertices of $G$ satisfying the following conditions:

1. For all $u, v \in V$, $cap(u, v) \geq f(u, v)$ (*capacity constraint*);
2. For all $u, v \in V$, $f(u, v) = -f(v, u)$ (*skew symmetry*);
3. For all $u \neq s, t$, $\sum_{v \in V} f(u, v) = 0$ (*flow conservation*).

An edge $[u, v]$ is *saturated* by the flow $f$ if $cap(u, v) = f(u, v)$. A path $P$ in the flow network $G$ is *saturated* by the flow $f$ if at least one edge in $P$ is saturated by $f$.

Note that even when there is no edge from a vertex $u$ to a vertex $v$, the flow value $f(u, v)$ can still be non-zero. For example, suppose that there is no edge from $u$ to $v$ but there is an edge from $v$ to $u$ of capacity 10, and that the flow value $f(v, u)$ is equal to 8. Then by the skew symmetry property, the flow value $f(u, v)$ is equal to $-8$, which is not 0.

However, if there is neither edge from $u$ to $v$ and nor edge from $v$ to $u$, then the flow value $f(u, v)$ must be 0. This is because from $cap(u, v) = cap(v, u) = 0$, by the capacity constraint property, we must have $f(u, v) \leq 0$

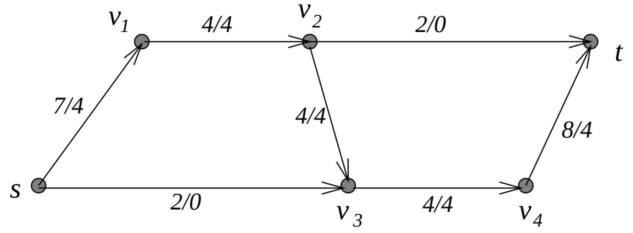



Figure 2.1: A flow network with a flow.

and $f(v,u) \leq 0$. By the skew symmetry property, $f(v,u) \leq 0$ implies $f(u,v) \geq 0$, which together with $f(u,v) \leq 0$ gives $f(u,v) = 0$.

Figure 2.1 is an example of a flow network $G$ with a flow, where on each edge $e = [u,v]$, we label a pair of numbers as "$a/b$" to indicate that the capacity of the edge $e$ is $a$ and the flow from vertex $u$ to vertex $v$ is $b$.

Given a flow network $G = (V, E)$ with the source $s$ and the sink $t$, let $f$ be a flow on $G$. The *value* of the flow is defined to be $\sum_{v \in V} f(s,v)$, denoted by $|f|$.

Now the Maximum Flow problem can be formally defined using our definition of optimization problems as a 4-tuple.

Maximum Flow $= \langle I_Q, S_Q, f_Q, opt_Q \rangle$

$I_Q$: the set of flow networks $G$

$S_Q$: $S_Q(G)$ is the set of flows $f$ in $G$

$f_Q$: $f_Q(G, f)$ is equal to the flow value $|f|$

$opt_Q$: max

Our first observation on the properties of a flow is as follows.

**Lemma 2.1.1** *Let $G = (V, E)$ be a flow network with the source $s$ and the sink $t$, and let $f$ be a flow in $G$. Then the value of the flow $f$ is equal to $\sum_{v \in V} f(v,t)$.*

Proof. We have

$$|f| = \sum_{v \in V} f(s,v) = \sum_{w \in V} \sum_{v \in V} f(w,v) - \sum_{w \neq s} \sum_{v \in V} f(w,v)$$

By the skew symmetry property, $f(w,v) = -f(v,w)$. Note that in the sum $\sum_{w \in V} \sum_{v \in V} f(w,v)$, for each pair of vertices $w$ and $v$, both $f(w,v)$ and

$f(v, w)$ appear exactly once. Thus, we have $\sum_{w \in V} \sum_{v \in V} f(w, v) = 0$. Now apply the skew symmetry property on the second term on the right hand side, we obtain

$$|f| = \sum_{w \neq s} \sum_{v \in V} f(v, w)$$

Thus, we have

$$|f| = \sum_{w \neq s} \sum_{v \in V} f(v, w) = \sum_{w \notin \{s,t\}} \sum_{v \in V} f(v, w) + \sum_{v \in V} f(v, t)$$

Finally, according to the skew symmetry property and the flow conservation property, for each $w \neq s, t$, we have

$$\sum_{v \in V} f(v, w) = -\sum_{v \in V} f(w, v) = 0$$

Thus, the sum $\sum_{w \notin \{s,t\}} \sum_{v \in V} f(v, w)$ is equal to 0. This gives the proof that $|f| = \sum_{v \in V} f(v, t)$. $\square$

The following lemma describes a basic technique to construct a positive flow in a flow network.

**Lemma 2.1.2** *Let $G = (V, E)$ be a flow network with the source $s$ and the sink $t$. There is a flow $f$ in $G$ with a positive value if and only if there is a path in $G$ from $s$ to $t$.*

PROOF. Suppose that there is a path $P$ from the source $s$ to the sink $t$. Let $e$ be an edge on $P$ with the minimum capacity $c > 0$ among all edges in $P$. Now it is easy to see that if we assign flow $c$ to each edge on the path $P$, and assign flow 0 to all other edges, we get a valid flow of value $c > 0$ in the flow network $G$.

For the other direction, suppose that $f$ is a flow of positive value in the flow network $G$. Suppose that the sink $t$ is not reachable from the source $s$. Let $V'$ be the set of vertices in $G$ that are reachable from $s$. Then $t \notin V'$.

Let $w$ be a vertex in $V'$. We first show

$$\sum_{v \in V'} f(w, v) = \sum_{v \in V} f(w, v) - \sum_{v \notin V'} f(w, v) \geq \sum_{v \in V} f(w, v) \qquad (2.1)$$

In fact, for any $v \notin V'$, since $v$ is not reachable from the source $s$, there is no edge from $w$ to $v$. Thus, $cap(w, v) = 0$, which implies $f(w, v) \leq 0$ by the capacity constraint property.

By Equation (2.1), we have

$$|f| = \sum_{v \in V} f(s,v) \le \sum_{v \in V'} f(s,v) = \sum_{w \in V'} \sum_{v \in V'} f(w,v) - \sum_{w \in V' - \{s\}} \sum_{v \in V'} f(w,v)$$

By the skew symmetry property, $\sum_{w \in V'} \sum_{v \in V'} f(w,v) = 0$. Thus,

$$|f| = - \sum_{w \in V' - \{s\}} \sum_{v \in V'} f(w,v)$$

For each $w \in V' - \{s\}$, according to Equation (2.1) and the flow conservation property, we have (note $t \notin V'$ so $w$ is neither $t$)

$$\sum_{v \in V'} f(w,v) \ge \sum_{v \in V} f(w,v) = 0$$

Thus, we have $|f| \le 0$. This contradicts our assumption that $f$ is a flow of positive value. This contradiction shows that the sink $t$ must be reachable from the source $s$, or equivalently, there is a path in the flow network $G$ from the source $s$ to the sink $t$. $\Box$

Thus, to construct a positive flow in a flow network $G$, we only need to find a path from the source to the sink. Many graph algorithms effectively find such a path.

Now one may suspect that finding a maximum flow is pretty straightforward: each time we find a path from the source to the sink, and add a new flow to saturate the path. After adding the new flow, if any edge becomes saturated, then it *seems* the edge has become useless so we delete it from the flow network. For those edges that are not saturated yet, it seems reasonable to have a new capacity for each of them to indicate the amount of room left along that edge to allow further flow through. Thus, the new capacity should be equal to the difference of the original capacity minus the amount of flow through that edge. Now on the resulting flow network, we find another path to add further flow, and so forth.

One might expect that if we repeat the above process until the flow network contains no path from the source $s$ to the sink $t$, then the obtained flow must be a maximum flow. Unfortunately, this observation is incorrect.

Consider the flow network with the flow in Figure 2.1. After deleting all saturated edges, the sink $t$ is no longer reachable from the source $s$ (see Figure 2.2). However, it seems that we still can push a flow of value 2 along the "path" $s \to v_3 \to v_2 \to t$, where although we do not have an edge from $v_3$ to $v_2$, but we still can push a flow of value 2 from $v_3$ to $v_2$ by *reducing*

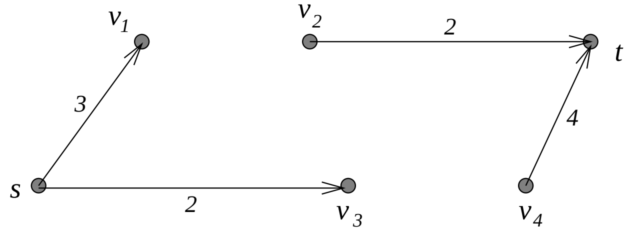



Figure 2.2: The sink $t$ is not reachable from the source after deleting saturated edges.

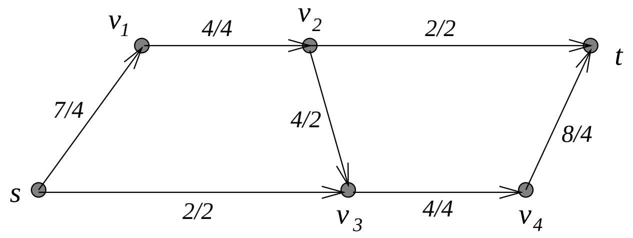



Figure 2.3: A flow larger than the one in Figure 2.1.

the original flow by 2 on edge $[v_2, v_3]$. This, in fact, does result in a larger flow in the original flow network, as shown in Figure 2.3.

Therefore, when a flow $f(u, v)$ is assigned on an edge $[u, v]$, it seems that not only do we need to modify the capacity of the edge $[u, v]$ to $cap(u, v) - f(u, v)$ to indicate the amount of further flow allowed through the edge, but also we need to record that a flow of amount $f(u, v)$ can be pushed along the opposite direction $[v, u]$, which is done by reducing the original flow along the edge $[u, v]$. In other words, we need add a new edge of capacity $f(u, v)$ from the vertex $v$ to the vertex $u$. Motivated by this discussion, we have the following definition.

**Definition 2.1.3** Given a flow network $G = (V, E)$ and given a flow $f$ in $G$, the *residual network* $G_f = (V, E')$ of $G$ (with respect to the flow $f$) is a flow network that has the same vertex set $V$ as $G$. Moreover, for each vertex pair $u, v$, if $cap(u, v) > f(u, v)$, then $[u, v]$ is an edge in $G_f$ with capacity $cap(u, v) - f(u, v)$.

Figure 2.4 is the residual network of the flow network in Figure 2.1 with respect to the flow given in the Figure. It can be clearly seen now that in the residual network, there is a path from $s$ to $t$: $s \to v_3 \to v_2 \to t$.

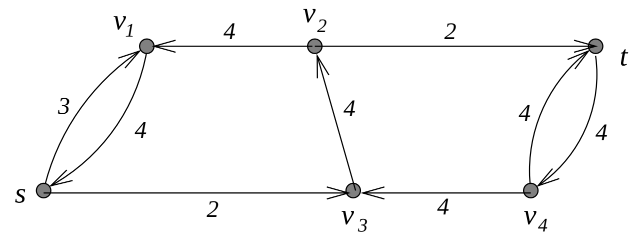



Figure 2.4: The residual network for Figure 2.1.

**Remark.** New edges may be created in the residual network $G_f$ that were not present in the original flow network $G$. For example, there is no edge from vertex $v_3$ to vertex $v_2$ in the original flow network in Figure 2.1, but in the residual network in Figure 2.4, there is an edge from $v_3$ to $v_2$. However, if there is neither an edge from $u$ to $v$ nor an edge from $v$ to $u$, then, since we must have $cap(u,v) = f(u,v) = 0$, there is also no edge from $u$ to $v$ in the residual network. This implies that the number of edges in a residual network cannot be more than twice of that in the original flow network. This fact will be useful when we analyze maximum flow algorithms.

**Lemma 2.1.3** *Let $G$ be a flow network and let $f$ be a flow in $G$. If $f^*$ is a flow in the residual network $G_f$, then the function $f^+ = f + f^*$, defined as $f^+(u,v) = f(u,v) + f^*(u,v)$ for all vertices $u$ and $v$, is a flow with value $|f^+| = |f| + |f^*|$ in $G$.*

PROOF. It suffices to verify that the function $f^+$ satisfies all the three constraints described in Definition 2.1.2. For each pair of vertices $u$ and $v$ in $G$, we denote by $cap(u,v)$ the capacity from $u$ to $v$ in the original flow network $G$, and by $cap_f(u,v)$ the capacity in the residual network $G_f$.

*The Capacity Constraint Condition.* We compute the value $cap(u,v) - f^+(u,v)$. By the definition we have

$$cap(u,v) - f^+(u,v) = cap(u,v) - f(u,v) - f^*(u,v)$$

Now by the definition of $cap_f$, we have $cap(u,v) - f(u,v) = cap_f(u,v)$. Moreover, since $f^*(u,v)$ is a flow in the residual network $G_f$, $cap_f(u,v) - f^*(u,v) \geq 0$. Consequently, we have $cap(u,v) - f^+(u,v) \geq 0$.

*The Skew Symmetry Condition.* Since both $f(u,v)$ and $f^*(u,v)$ are flows in the flow networks $G$ and $G_f$, respectively, we have $f(u,v) = -f(v,u)$ and

$f^*(u,v) = -f^*(v,u)$. Thus,

$$f^+(u,v) = f(u,v) + f^*(u,v) = -f(v,u) - f^*(v,u) = -f^+(v,u)$$

*The Flow Conservation Condition.* Again, since both $f(u,v)$ and $f^*(u,v)$ are flows in the flow networks $G$ and $G_f$, respectively, we have for all $u \neq s, t$

$$\sum_{v \in V} f^+(u,v) = \sum_{v \in V} f(u,v) + \sum_{v \in V} f^*(u,v) = 0$$

Thus, $f^+$ is a flow in the flow network $G$. For the flow value of $f^+$, we have

$$|f^+| = \sum_{v \in V} f^+(s,v) = \sum_{v \in V} f(s,v) + \sum_{v \in V} f^*(s,v) = |f| + |f^*|$$

□

Now we are ready for the following fundamental theorem for maximum flow algorithms.

**Theorem 2.1.4** *Let $G$ be a flow network and let $f$ be a flow in $G$. The flow $f$ is a maximum flow in $G$ if and only if the residual network $G_f$ has no positive flow, or equivalently, if and only if there is no path from the source $s$ to the sink $t$ in the residual network $G_f$.*

PROOF. The equivalence of the second condition and the third condition is given by Lemma 2.1.2. Thus, we only need to prove that the first condition and the second condition are equivalent.

Suppose that $f$ is a maximum flow in $G$. If the residual network $G_f$ has a positive flow $f^*$, $|f^*| > 0$, then by Lemma 2.1.3, $f^+ = f + f^*$ is also a flow in $G$ with flow value $|f| + |f^*|$. This contradicts the assumption that $f$ is a maximum flow in $G$ since $|f^*| > 0$. Thus, the residual network $G_f$ has no positive flow.

For the other direction, we assume that $f$ is not a maximum flow in $G$. Let $f_{\max}$ be a maximum flow in $G$. Thus, $|f_{\max}| - |f| > 0$. Now define a function $f^-$ on each pair $(u,v)$ of vertices in the residual network $G_f$ as follows,

$$f^-(u,v) = f_{\max}(u,v) - f(u,v)$$

We claim that $f^-$ is a valid flow in the residual network $G_f$.

The function $f^-$ satisfies the capacity constraint condition: since $cap_f(u,v) = cap(u,v) - f(u,v)$, we have

$$cap_f(u,v) - f^-(u,v) = cap(u,v) - f(u,v) - f^-(u,v)$$

**Algorithm. Ford-Fulkerson**

Input: a flow network $G$
Output: a maximum flow $f$ in $G$

1. let $f(u,v) = 0$ for all pairs $(u,v)$ of vertices in $G$;
2. construct the residual network $G_f$;
3. **while** there is a positive flow in $G_f$ **do**
   construct a positive flow $f^*$ in $G_f$;
   let $f = f + f^*$ be the new flow in $G$;
   construct the residual network $G_f$;

Figure 2.5: Ford-Fulkerson's method for maximum flow

Note that $f(u,v) + f^-(u,v) = f_{\max}(u,v)$. Since $f_{\max}$ is a flow in $G$, we have $cap(u,v) - f_{\max}(u,v) \geq 0$. Consequently, we have $cap_f(u,v) - f^-(u,v) \geq 0$.

The function $f^-$ satisfies the skew symmetry condition:

$$f^-(u,v) = f_{\max}(u,v) - f(u,v) = -f_{\max}(v,u) + f(v,u) = -f^-(v,u)$$

The function $f^-$ satisfies the flow conservation condition: for all $u \neq s,t$, we have

$$\sum_{v \in V} f^-(u,v) = \sum_{v \in V} f_{\max}(u,v) - \sum_{v \in V} f(u,v) = 0$$

Thus, $f^-$ is a valid flow in the residual network $G_f$. Moreover, since we have

$$|f^-| = \sum_{v \in V} f^-(s,v) = \sum_{v \in V} f_{max}(s,v) - \sum_{v \in V} f(s,v) = |f_{max}| - |f| > 0$$

We conclude that the residual network $G_f$ has a positive flow.

This completes the proof of the theorem. □

Theorem 2.1.4 suggests a classical method (called *Ford-Fulkerson's method*), described in Figure 2.5 for constructing a maximum flow in a given flow network.

According to Lemma 2.1.2, there is a positive flow in the residual network $G_f$ if and only if there is a directed path from the source $s$ to the sink $t$ in $G_f$. Such a directed path can be found by a number of efficient graph search algorithms. Thus, the condition in the **while** loop in step 3 in the algorithm **Ford-Fulkerson** can be easily checked. Theorem 2.1.4 guarantees that when the algorithm halts, the obtained flow is a maximum flow.

The only problem left is how we construct a positive flow each time the residual network $G_f$ is given. In order to make the algorithm efficient, we need to adopt a strategy that constructs a positive flow in the given residual network $G_f$ effectively so that the number of executions of the **while** loop in step 3 is as small as possible. Many algorithms have been proposed for finding such a positive flow. In the next section, we describe an important technique, the *shortest path saturation* method, for constructing a positive flow given a flow network. We will see that when this method is adopted, the algorithm **Ford-Fulkerson** is efficient.

## 2.2 Shortest path saturation method

The method of *shortest path saturation* is among the most successful methods in constructing a positive flow in the residual network $G_f$ to limit the number of executions of the **while** loop in step 3 of the algorithm **Ford-Fulkerson**, where the *length* of a path is measured by the number of edges in the path.

In the rest discussions in this chapter, we always assume that the flow network $G$ has $n$ vertices and $m$ edges.

We first briefly describe an algorithm suggested by Edmond and Karp [35]. Edmond and Karp considered the method of constructing a positive flow for the residual network $G_f$ by finding a shortest path from $s$ to $t$ in $G_f$ and saturating it. Intuitively, each execution of this process saturates at least one edge from a shortest path from $s$ to $t$ in the residual network $G_f$. Thus, after $O(m)$ such executions, all shortest paths from $s$ to $t$ in $G_f$ are saturated, and the distance from the source $s$ to the sink $t$ should be increased. This implies that after $O(nm)$ such executions, the distance from $s$ to $t$ should be larger than $n$, or equivalently, the sink $t$ will become unreachable from the source $s$. Therefore, if we adopt Edmond and Karp's method to find positive flow in the residual network $G_f$, then the **while** loop in step 3 in the algorithm **Ford-Fulkerson** is executed at most $O(nm)$ times. Since a shortest path in the residual network $G_f$ can be found in time $O(m)$ (using, for example, breadth first search), this concludes that Edmond-Karp's algorithm finds the maximum flow in time $O(nm^2)$.

Dinic [33] proposed a different approach. Instead of finding a single shortest path in the residual network $G_f$, Dinic finds *all* shortest paths from $s$ to $t$ in $G_f$, then saturates *all* of them. In the following, we give a detailed analysis for this approach.

**Definition 2.2.1** Let $G$ be a flow network. A flow $f$ in $G$ is a *shortest* *saturation flow* if (1) $f(u, v) > 0$ implies that $[u, v]$ is an edge in a shortest path from $s$ to $t$ in $G$, and (2) the flow $f$ saturates every shortest path from $s$ to $t$ in $G$.

For each vertex $v$ in a flow network $G$ with source $s$ and sink $t$, denote by $dist(v)$ the length of (i.e., the number of edges in) the shortest path in $G$ from the source $s$ to $v$ (the *distance from $s$ to $v$*). Similarly, if $f$ is a flow in $G$, we let $dist_f(v)$ be the length of the shortest path from $s$ to $v$ in the residual network $G_f$.

**Lemma 2.2.1** *Let $G$ be a flow network with source $s$ and sink $t$, and let $f$ be a shortest saturation flow in $G$. Then $dist(t) < dist_f(t)$.*

PROOF. First we note that if a vertex $v$ is on a shortest path $P$ from the source $s$ to a vertex $w$ in $G$, then the subpath of $P$ from $s$ to $v$ is a shortest path from $s$ to $v$.

We prove two facts.

**Fact 1**: Suppose $[v, w]$ is an edge in $G_f$, then $dist(w) \leq dist(v) + 1$.

Suppose $[v, w]$ is an edge in $G$. Then since any shortest path from $s$ to $v$ plus the edge $[v, w]$ is a path from $s$ to $w$, whose length cannot be smaller than $dist(w)$, we have $dist(w) \leq dist(v) + 1$.

If $[v, w]$ is not an edge in $G$, then since $[v, w]$ is an edge in the residual network $G_f$ of $G$ with respect to the flow $f$, the flow value $f(w, v)$ must be larger than 0. Since $f$ is a shortest saturation flow, only for edges in shortest paths from $s$ to $t$ in $G$, $f$ may have positive flow value. Thus $[w, v]$ must be an edge in a shortest path $P$ from $s$ to $t$ in $G$. Then the subpath of $P$ from $s$ to $w$ is a shortest path from $s$ to $w$, and the subpath of $P$ from $s$ to $v$ is a shortest path from $s$ to $v$. That is, $dist(w) = dist(v) - 1$, which of course implies $dist(w) \leq dist(v) + 1$.

**Fact 2**: For any vertex $v$, we have $dist(v) \leq dist_f(v)$.

Suppose $r = dist_f(v)$. Let $(s, v_1, v_2, \ldots, v_{r-1}, v)$ be a shortest path in $G_f$ from $s$ to $v$. Then by Fact 1, we have $dist(v) \leq dist(v_{r-1}) + 1$, $dist(v_i) \leq dist(v_{i-1}) + 1$, for $i = 2, \ldots, r-1$, and $dist(v_1) \leq dist(s) + 1$. Thus,

$$\begin{aligned} dist(v) &\leq dist(v_{r-1}) + 1 \\ &\leq dist(v_{r-2}) + 2 \\ &\cdots \\ &\leq dist(v_1) + (r-1) \\ &\leq dist(s) + r \end{aligned}$$

$$= \quad r \quad = \quad dist_f(v)$$

This proves Fact 2.

Now we are ready to prove our lemma.

Fact 2 shows that $dist(t) \leq dist_f(t)$. Hence, to prove the lemma, we only need to show that $dist(t)$ and $dist_f(t)$ are distinct. Let us assume the contrary that $dist(t) = dist_f(t) = r$ and derive a contradiction.

Let $P = (v_0, v_1, \ldots, v_{r-1}, v_r)$ be a shortest path in the residual network $G_f$ from the source $s$ to the sink $t$, where $v_0 = s$ and $v_r = t$. By Fact 1, we have

$$\begin{aligned} dist(v_r) &\leq dist(v_{r-1}) + 1 \\ &\leq dist(v_{r-2}) + 2 \\ &\cdots \\ &\leq dist(v_0) + r \\ &= dist(s) + r = r \end{aligned}$$

By our assumption, we also have $dist(v_r) = dist(t) = r$. Thus, all inequalities "$\leq$" in the above formula should be equality "$=$". This gives $dist(v_{i+1}) = dist(v_i) + 1$ for all $i = 0, \ldots, r-1$. But this implies that all $[v_i, v_{i+1}]$ are also edges in the original flow network $G$. In fact, if $[v_i, v_{i+1}]$ is not an edge in $G$, then since $[v_i, v_{i+1}]$ is an edge in the residual network $G_f$, $[v_{i+1}, v_i]$ must be an edge in $G$ with the flow value $f(v_{i+1}, v_i) > 0$. Since $f$ is a shortest saturation flow, $f(v_{i+1}, v_i) > 0$ implies that the edge $[v_{i+1}, v_i]$ is in a shortest path in $G$ from $s$ to $t$. But this would imply that $dist(v_{i+1}) = dist(v_i) - 1$, contradicting the fact $dist(v_{i+1}) = dist(v_i) + 1$.

Thus all $[v_i, v_{i+1}]$, $i = 0, \ldots, r-1$, are edges in the original flow network $G$, so $P$ is also a path in $G$. Since $P$ is of length $r$ and $dist(t) = r$, $P$ is a shortest path from $s$ to $t$. Since $f$ is a shortest saturation flow, the path $P$ in $G$ must be saturated by $f$, i.e., one of the edges in $P$ is saturated by the flow $f$, which, by the definition of residual networks, should not appear in the residual network $G_f$. But this contradicts the assumption that $P$ is a path in $G_f$.

This contradiction shows that we must have $dist(t) < dist_f(t)$. $\square$

Now we are ready to discuss how the shortest path saturation method is applied to the algorithm **Ford-Fulkerson**.

**Theorem 2.2.2** *If in each execution of the* **while** *loop in step 3 in the algorithm* **Ford-Fulkerson***, we construct a shortest saturation flow $f^*$ for* *the residual network $G_f$, then the number of executions of the* **while** *loop is bounded by $n - 1$.*

PROOF. Suppose that $f^*$ is a shortest saturation flow in the residual network $G_f$. By Lemma 2.2.1, the distance from $s$ to $t$ in the residual network $(G_f)_{f^*}$ (of $G_f$ with respect to $f^*$) is at least 1 plus the distance from $s$ to $t$ in the original residual network $G_f$. Note that the residual network $(G_f)_{f^*}$ of $G_f$ with respect to $f^*$ is the residual network $G_{f+f^*}$ of the original flow network $G$ with respect to the new flow $f + f^*$. This can be easily verified by the following relation:

$$\begin{aligned} cap_f(u,v) - f^*(u,v) &= cap(u,v) - (f(u,v) + f^*(u,v)) \\ &= cap(u,v) - [f + f^*](u,v) \end{aligned}$$

Thus, $cap_f(u,v) > f^*(u,v)$ if and only if $cap(u,v) > [f + f^*](u,v)$, or equivalently, $[u,v]$ is an edge in $(G_f)_{f^*}$ if and only if it is an edge in $G_{f+f^*}$.

Therefore, the distance from $s$ to $t$ in the current residual network $G_f$ is at least 1 plus the distance from $s$ to $t$ in the residual network $G_f$ in the previous execution of the **while** loop. Since before the **while** loop, the distance from $s$ to $t$ in $G_f = G$ is at least 1 (the source $s$ and the sink $t$ are distinct in $G$), we conclude that after $n - 1$ executions of the **while** loop, the distance from $s$ to $t$ in the residual network $G_f$ is at least $n$. This means that the sink $t$ is not reachable from the source $s$ in the residual network $G_f$. By Theorem 2.1.4, the algorithm **Ford-Fulkerson** stops with a maximum flow $f$. $\Box$

The problem left is how a shortest saturation flow can be constructed for the residual network $G_f$. By the definition, a shortest saturation flow saturates all shortest paths from $s$ to $t$ and has positive value only on edges on shortest paths from $s$ to $t$. Thus, constructing a shortest saturation flow can be split into two steps: (1) finding all shortest paths from $s$ to $t$ in $G_f$, and (2) saturating all these paths.

Since there can be too many (up to $\Omega(2^{cn})$ for some constant $c > 0$) shortest paths from $s$ to $t$, it is infeasible to enumerate all of them. Instead, we construct a subnetwork $L_0$ in $G_f$, called the *layered network*, that contains exactly those edges contained in shortest paths of $G_f$ from $s$ to $t$.

The layered network $L_0$ of $G_f$ can be constructed using a modification of the well-known *breadth first search* process, given in Figure 2.6, where $Q$ is a queue that is a data structure serving for "first-in-first-out".

Stage 1 of the algorithm **Layered-Network** is a modification of the standard breadth first search process. The stage assigns a value $dist(v)$

**Algorithm. Layered-Network**

Input: the residual network $G_f = (V_f, E_f)$
Output: the layered network $L_0 = (V_0, E_0)$ of $G_f$

**Stage 1.** {constructing all shortest paths from $s$ to each vertex}
1. $V_0 = \emptyset$; $E_0 = \emptyset$;
2. **for** all vertices $v$ in $G_f$ **do** $dist(v) = \infty$;
3. $dist(s) = 0$; $Q \leftarrow s$;
4. **while** $Q$ is not empty **do**
   $v \leftarrow Q$;
   **for** each edge $[v, w]$ **do**
      **if** $dist(w) = \infty$ **then**
         $Q \leftarrow w$; $dist(w) = dist(v) + 1$;
         $V_0 = V_0 \cup \{w\}$; $E_0 = E_0 \cup \{[v, w]\}$;
      **else if** $dist(w) = dist(v) + 1$ **then** $E_0 = E_0 \cup \{[v, w]\}$;

**Stage 2.** {deleting vertices not in a shortest path from $s$ to $t$}
5. let $L_0^r$ be $L_0 = (V_0, E_0)$ with all edge directions reversed;
6. perform a breadth first search on $L_0^r$, starting from $t$;
7. delete the vertices $v$ from $L_0$ if $v$ is not marked in step 6.

Figure 2.6: Construction of the layered network $L_0$

to each vertex $v$, which equals the distance from the source $s$ to $v$, and includes an edge $[v, w]$ in $L_0$ only if $dist(v) = dist(w) - 1$. The difference of this stage from the standard breadth first search is that for an edge $[v, w]$ with $dist(v) = dist(w) - 1$, even if the vertex $w$ has been in the queue $Q$, we still include the edge $[v, w]$ in $L_0$ to record the shortest paths from $s$ to $w$ that contain the edge $[v, w]$. Therefore, after stage 1, for each vertex $v$, exactly those edges contained in shortest paths from $s$ to $v$ are included in the network $L_0 = (V_0, E_0)$.

Stage 2 of the algorithm is to delete from $L_0$ all vertices (and their incident edges) that are not in shortest paths from the source $s$ to the sink $t$. Since $L_0$ contains only shortest paths from $s$ to each vertex and every vertex in $L_0$ is reachable from $s$ in $L_0$, a vertex $v$ is not contained in any shortest path from $s$ to $t$ if and only if $t$ is not reachable from $v$ in the network $L_0$, or equivalently, $v$ is not reachable from $t$ in the reversed network $L_0^r$. Step 6 in the algorithm identifies those vertices that are reachable from $t$ in $L_0^r$, and step 7 deletes those vertices that are not identified in step 6.

Therefore, the algorithm **Layered-Network** correctly constructs the layered network $L_0$ of the residual network $G_f$. By the well-known analysis for the breadth first search process, the running time of the algorithm **Layered-Network** is bounded by $O(m)$.

Having obtained the layered network $L_0$, we now construct a shortest saturation flow so that for each path from $s$ to $t$ in $L_0$, at east one edge is saturated. There are two different methods for this, which are described in the following two subsections.

### 2.2.1 Dinic's algorithm

Given the layered network $L_0$, Dinic's algorithm for saturating all shortest paths from $s$ to $t$ in $G_f$ is very simple, and can be described as follows. Starting from the vertex $s$, we follow the edges in $L_0$ to find a maximal path $P$ of length at most $dist(t)$. Since the network $L_0$ is layered and contains only edges in the shortest paths from $s$ to $t$ in $G_f$, the path $P$ can be found in a straightforward way (i.e., at each vertex, simply follow an arbitrary outgoing edge from the vertex). Thus, the path $P$ can be constructed in time $O(dist(t)) = O(n)$. Now if the ending vertex is $t$, then we have found a path from $s$ to $t$. We trace back the path $P$ to find the edge $e$ on $P$ with minimum capacity $c$. Now we can push $c$ amount of flow along the path $P$. Then we delete the edges on $P$ that are saturated by the new flow. Note that this deletes at least one edge from the layered network $L_0$. On the other hand, if the ending vertex $v$ of $P$ is not $t$, then $v$ must be a "deadend". Thus, we can delete the vertex $v$ (and all incoming edges to $v$). In conclusion, in the above process of time $O(n)$, at least one edge is removed from the layered network $L_0$. Thus, after at most $m$ such processes, the vertices $s$ and $t$ are disconnected, i.e., all shortest paths from $s$ to $t$ are saturated. This totally takes time $O(nm)$. A formal description for this process is given in Figure 2.7.

For completeness, we present in Figure 2.8 the complete Dinic's algorithm for constructing a maximum flow in a given flow network.

**Theorem 2.2.3** *The running time of Dinic's maximum flow algorithm (Algorithm* **Max-Flow-Dinic** *in Figure 2.8) is* $O(n^2m)$.

PROOF. Theorem 2.2.2 claims that the **while** loop in step 3 in the algorithm is executed at most $n-1$ times. In each execution of the loop, constructing the layered network $L_0$ by **Layered-Network** takes time $O(m)$. Constructing the shortest saturation flow $f^*$ in $G_f$ from the layered network $L_0$ by **Dinic-Saturation** takes time $O(nm)$. All other steps in the loop takes time at most $O(n^2)$. Therefore, the total running time for the

**Algorithm. Dinic-Saturation**

Input: the layered network $L_0$
Output: a shortest saturation flow $f^*$ in $G_f$

1. **while** there is an edge from $s$ in $L_0$ **do**
   find a path $P$ of maximal length from $s$ in $L_0$;
   **if** $P$ leads to $t$
   **then** saturate $P$ and delete at least one edge on $P$;
   **else** delete the last vertex of $P$ from $L_0$.

Figure 2.7: Dinic's algorithm for a shortest saturation flow

algorithm **Max-Flow-Dinic** is bounded by $O(n^2m)$. □

### 2.2.2 Karzanov's algorithm

In Dinic's algorithm **Max-Flow-Dinic**, the computation time for each execution of the **while** loop in step 3 is dominated by the substep of constructing the shortest saturation flow $f^*$ in $G_f$ from the layered network $L_0$. Therefore, if this substep can be improved, then the time complexity of the whole algorithm can be improved. In this subsection, we show an algorithm by Karzanov [80] that improves this substep.

Let us have a closer look at our construction of the shortest saturation flow $f^*$ in the algorithm **Max-Flow-Dinic**. With the layered network $L_0$ being constructed, we iterate the process of searching a path in $L_0$ from the source $s$ to the sink $t$, pushing flow along the path, and saturating (thus cutting) at least one edge on the path. In the worst case, for each such a path, we may only be able to cut one edge. Therefore, to ensure that the source $s$ is eventually separated from the sink $t$ in $L_0$, we may have to perform the above iteration $m$ times.

The basic idea of Karzanov's algorithm is to reduce the number of times we have to perform the above iteration from $m$ to $n$. In each iteration, instead of saturating an edge in $L_0$, Karzanov saturates a vertex in $L_0$. Since there are at most $n$ vertices in the layered network $L_0$, the number of iterations will be bounded by $n$.

**Definition 2.2.2** Let $v$ be a vertex in the layered network $L_0 = (V_0, E_0)$.

**Algorithm. Max-Flow-Dinic**

Input: a flow network $G$
Output: a maximum flow $f$ in $G$

1. let $f(u,v) = 0$ for all pairs $(u,v)$ of vertices in $G$;
2. construct the residual network $G_f$;
3. **while** there is a positive flow in $G_f$ **do**
   call **Layered-Network** to construct the layered
   network $L_0$ for $G_f$;
   call **Dinic-Saturation** on $L_0$ to construct a shortest
   saturation flow $f^*$ in $G_f$;
   let $f = f + f^*$ be the new flow in $G$;
   construct the residual network $G_f$;

Figure 2.8: Dinic's algorithm for maximum flow

Define the *capacity*, $cap(v)$, of the vertex $v$ to be

$$cap(v) = \min\left( \sum_{[w,v]\in E_0} cap(w,v), \sum_{[v,u]\in E_0} cap(v,u) \right)$$

That is, $cap(v)$ is the maximum amount of flow we can push through the vertex $v$. For the source $s$ and the sink $t$, we naturally define

$$cap(s) = \sum_{[s,u]\in E_0} cap(s,u) \qquad \text{and} \qquad cap(t) = \sum_{[w,t]\in E_0} cap(w,t)$$

If we start from an arbitrary vertex $v$ and try to push a flow of amount $cap(v)$ through $v$, it may not always be possible. For example, pushing $cap(v) = 10$ units flow through a vertex $v$ may require to push 5 units flow along an edge $(v,w)$, which requires that $cap(w)$ is at least 5. But the capacity of the vertex $w$ may be less than 5, thus we would be blocked at the vertex $w$. However, if we always pick the vertex $w$ in $L_0$ with the smallest capacity, this problem will disappear. In fact, trying to push a flow of amount $cap(w)$, where $w$ has the minimum $cap(w)$, will require no more than $cap(v)$ amount of flow to go through a vertex $v$ for any vertex $v$. Therefore, we can always push the flow all the way to the sink $t$ (assuming we have no deadend vertices). Similarly, we can *pull* this amount $cap(w)$ of flow from the incoming edges of $w$ all the way back to the source $s$. Note

**Algorithm. Karzanov-Initiation**

Input: the layered network $L_0$
Output: the vertex capacity for each vertex

1. **for** each vertex $v \neq s, t$ **do** $in[v] = 0$; $out[v] = 0$;
2. $in[s] = +\infty$; $out[t] = +\infty$;
3. **for** each edge $[u, v]$ in $L_0$ **do**
   $in[v] = in[v] + cap(u, v)$; $out[u] = out[u] + cap(u, v)$;
4. **for** each vertex $v$ in $L_0$ **do** $cap[v] = \min\{in[v], out[v]\}$.

Figure 2.9: Computing the capacity for each vertex

that this process saturates the vertex $w$. Thus, the vertex $w$ can be removed from the layered network $L_0$ in the remaining iterations.

Now we can formally describe Karzanov's algorithm. The first subroutine given in Figure 2.9 computes the capacity for each vertex in the layered network $L_0$.

We will always start with a vertex $v$ with the smallest $cap(v)$ and push a flow $f^v$ of amount $cap(v)$ through it all the way to the sink $t$. This process, called **Push**$(v, f^v)$ and given in Figure 2.10, is similar to the breadth first search process, starting from the vertex $v$. We use the array $fl[w]$ to record the amount of flow requested to push through the vertex $w$, and $fl[w] = 0$ implies that the vertex $w$ has not been seen in the breadth first search process.

We make a few remarks on the algorithm **Push**$(v, f^v)$. First we assume that there is no dead-vertex in the layered network $L_0$. That is, every edge in $L_0$ is on a shortest path from $s$ to $t$. This condition holds when the layered network $L_0$ is built by the algorithm **Layered-Network**. We will keep this condition in Karzanov's main algorithm when vertices and edges are deleted from $L_0$.

The algorithm **Push**$(v, f^v)$, unlike the standard breadth first search, may not search all vertices reachable from $v$. Instead, for each vertex $u$, with a requested flow amount $fl[u]$ through it, the algorithm looks at the out-going edges from $u$ to push the flow of amount $fl[u]$ through these edges. Once this amount of flow is pushed through some of these edges, the algorithm will ignore the rest of the out-going edges from $u$. Note that since the vertex $v$ has the minimum $cap(v)$, and no $fl[u]$ for any vertex $u$ is larger than $cap(v)$, the amount $fl[u]$ can never be larger than $cap(u)$. Thus, the

**Algorithm.** $\mathbf{Push}(v, f^v)$

Input: the layered network $L_0$

1. $Q \leftarrow v$; $\{Q$ is a queue$\}$ $fl[v] = cap(v)$;
2. **while** $Q$ is not empty **do**

2.1. $u \leftarrow Q$; $f_0 = fl[u]$;

2.2. **while** $f_0 > 0$ **do**

pick an out-going edge $[u, w]$ from $u$;

**if** $fl[w] = 0$ and $w \neq t$ **then** $Q \leftarrow w$;

2.2.1. **if** $cap(u, w) \leq f_0$ **then**

$f^v(u, w) = cap(u, w)$; delete the edge $[u, w]$;

$fl[w] = fl[w] + cap(u, w)$; $f_0 = f_0 - cap(u, w)$;

2.2.2. **else** $\{cap(u, w) > f_0\}$

$f^v(u, w) = f_0$; $fl[w] = fl[w] + f_0$;

$cap(u, w) = cap(u, w) - f_0$; $f_0 = 0$;

2.3. **if** $u \neq v$ **then** $cap(u) = cap(u) - fl[u]$;

2.4. **if** $u \neq v$ and $cap(u) = 0$ **then** delete the vertex $u$.

Figure 2.10: Pushing a flow of value $cap(v)$ from $v$ to $t$

amount $fl[u]$ of flow can always be pushed through the vertex $u$.

When a flow $f'$ is pushed along an edge $[u, w]$, we add $f'$ to $fl[w]$ to record this new requested flow through the vertex $w$. Note that when a vertex $u$ is picked from the queue $Q$ in step 2.1, the flow requested along in-coming edges to $u$ has been completely decided. Thus, $fl[u]$ records the correct amount of flow to be pushed through $u$.

Also note that in the subroutine $\mathbf{Push}(v, f^v)$, we neither change the value $cap(v)$ nor remove the vertex $v$ from the layered network $L_0$. This is because the vertex $v$ will be used again in the following $\mathbf{Pull}(v, f^v)$ algorithm.

The algorithm $\mathbf{Pull}(v, f^v)$ is very similar to algorithm $\mathbf{Push}(v, f^v)$. We start from the vertex $v$ and pull a flow $f^v$ of amount $cap(v)$ all the way back to the source vertex $s$. Note that now the breadth first search is on the reversed directions of the edges of the layered network $L_0$. Thus, we will also keep a copy of the reversed layered network $L_0^r$, which is $L_0$ with all edge directions reversed. The reversed layered network $L_0^r$ can be constructed from the layered network $L_0$ in time $O(m)$ (this only needs to be done once for all calls to **Pull**). Moreover, note that the only vertex that can be seen in

**Algorithm.** $\mathbf{Pull}(v, f^v)$
Input: the reversed layered network $L_0^r$

1. $Q \leftarrow v$; $\{Q$ is a queue$\}$ $fl[v] = cap(v)$;
2. **while** $Q$ is not empty **do**
2.1. $u \leftarrow Q$; $f_0 = fl[u]$;
2.2. **while** $f_0 > 0$ **do**
  pick an in-coming edge $[w, u]$ to $u$;
  **if** $fl[w] = 0$ and $w \neq s$ **then** $Q \leftarrow w$;
2.2.1. **if** $cap(w, u) \leq f_0$ **then**
    $f^v(w, u) = cap(w, u)$; delete the edge $[w, u]$;
    $fl[w] = fl[w] + cap(w, u)$; $f_0 = f_0 - cap(w, u)$;
2.2.2. **else** $\{cap(w, u) > f_0\}$
    $f^v(w, u) = f_0$; $fl[w] = fl[w] + f_0$;
    $cap(w, u) = cap(w, u) - f_0$; $f_0 = 0$;
2.3. $cap(u) = cap(u) - fl[u]$;
2.4. **if** $cap(u) = 0$ **then** delete the vertex $u$.

Figure 2.11: Pulling a flow of value $cap(v)$ from $s$ to $v$

both algorithms $\mathbf{Push}(v, f^v)$ and $\mathbf{Pull}(v, f^v)$ is the vertex $v$ — $\mathbf{Push}(v, f^v)$ is working on the vertices "after" $v$ in $L_0$ while $\mathbf{Pull}(v, f^v)$ is working on the vertices "before" $v$ in $L_0$. Therefore, no updating is needed between the call to $\mathbf{Push}(v, f^v)$ and the call to $\mathbf{Pull}(v, f^v)$. The algorithm $\mathbf{Pull}(v, f^v)$ is given in Figure 2.11.

After the execution of the $\mathbf{Pull}(v, f^v)$ algorithm, the vertex $v$ with minimum capacity always gets removed from the layered network $L_0$.

With the subroutines **Push** and **Pull**, Karzanov's algorithm for constructing a shortest saturation flow $f^*$ in the residual network $G_f$ is given in Figure 2.12.

Some implementation details should be explained for the algorithm **Karzanov-Saturation**.

The dynamic deletions of edges and vertices from in the layered network $L_0$ can be recorded by the two dimensional array $cap(\cdot, \cdot)$: we set $cap(u, w) = 0$ when the edge $[u, w]$ is deleted from $L_0$. The actual deletion of the item $[u, w]$ from the adjacency list representation of the layered network $L_0$ or of the reversed layered network $L_0^r$ is done later: when we scan the adjacency list and encounter an item $[u, w]$, we first check if $cap(u, w) = 0$. If so, we

**Algorithm. Karzanov-Saturation**

Input: the layered network $L_0$
Output: a shortest saturation flow $f^*$ in $G_f$

1. call **Karzanov-Initiation** to compute the vertex capacity $cap(v)$ for each vertex $v$;
2. $f^* = 0$;
3. **while** there is a vertex in $L_0$ **do**
   pick a vertex $v$ in $L_0$ with minimum $cap(v)$;
   **if** $v$ is a dead-vertex **then** delete $v$ from $L_0$
   **else** call **Push**$(v, f^v)$; call **Pull**$(v, f^v)$; $f^* = f^* + f^v$

Figure 2.12: Karzanov's algorithm for shortest saturation flow

delete the item (using an extra $O(1)$ time) and move to the next item in the list.

Similarly, we keep a vertex array to record whether a vertex is in the layered network $L_0$. There are two ways to make a vertex $v$ become a dead-vertex: (1) $cap(v) = 0$, which means either all in-coming edges to $v$ or all out-going edges from $v$ are saturated; and (2) $v$ becomes a dead-vertex because of removal of other dead-vertices. For example, suppose that $v$ has only one in-coming edge $[u, v]$ of capacity 10 and one out-going edge $[v, w]$ of capacity 5. Then $cap(v) = 5 \neq 0$. However, if $w$ becomes a dead-vertex (e.g., because all out-going edges from $w$ are saturated) and is deleted, then the vertex $v$ will also become a dead-vertex. For this, we also record the number $in[v]$ of in-coming edges and the number $out[v]$ of out-going edges for each vertex $v$. Once one of $in[v]$ and $out[v]$ becomes 0, we set $cap(v) = 0$ immediately. Since in step 3 of the algorithm **Karzanov-Saturation** we always pick the vertex of minimum $cap(v)$, dead-vertices in $L_0$ will always be picked first. Consequently, when the subroutines **Push** and **Pull** are called in step 3, there must be no dead-vertex in the current layered network $L_0$.

We now analyze the algorithm **Karzanov-Saturation**.

**Lemma 2.2.4** *The algorithm* **Karzanov-Saturation** *takes time* $O(n^2)$.

PROOF. Step 1 takes time $O(e) = O(n^2)$. Since each execution of the **while** loop deletes at least one vertex $v$ from $L_0$ (either because $v$ is a dead-vertex or because of the subroutine call **Pull**$(v, f^v)$), the **while** loop in step 3 is executed at most $n$ times.

The first substep in step 3, finding a vertex $v$ of minimum $cap(v)$ in the layered network $L_0$, takes time $O(n)$. Thus, the total time spent by the algorithm **Karzanov-Saturation** on this substep is bounded by $O(n^2)$.

The analysis for the time spent on the subroutine calls to **Push** and **Pull** is a bit more tricky. Let us first consider the subroutine **Push**$(v, f^v)$. To push a flow of amount $fl[u]$ through a vertex $u$, we take each out-going edge from $u$. If the capacity of the edge is not larger than the amount of flow we request to push (step 2.2.1 in the algorithm **Push**$(v, f^v)$), we saturate and delete the edge; if the capacity of the edge is larger than the amount of flow we request to push (step 2.2.2 in the algorithm **Push**$(v, f^v)$), we let all remaining flow go along that edge and jump out from the **while** loop in Step 2.2 in the algorithm **Push**$(v, f^v)$. Moreover, once an edge gets deleted in the algorithm, the edge will never appear in the layered network $L_0$ for the later calls for **Push** in the algorithm **Karzanov-Saturation**. Thus, each execution of the **while** loop in step 2.2 of the algorithm **Push**$(v, f^v)$, except maybe the last one, deletes an edge from the layered network $L_0$. Hence, the total number of such executions in the whole algorithm **Karzanov-Saturation** cannot be larger than $m$ plus $n$ times the number of calls to **Push**, where $m$ is for the executions of the loop that delete an edge in $L_0$, and $n$ is for the executions of the loop that do not delete edges. Therefore, the total number of executions of the **while** loop in step 2.2 in the algorithm **Push**$(v, f^v)$ for all calls to **Push** in the algorithm **Karzanov-Saturation** is bounded by $O(n^2)$. Since each execution of this **while** loop takes constant time and this part dominates the running time of the algorithm **Push**, we conclude that the total time spent by **Karzanov-Saturation** on the calls to **Push** is bounded by $O(n^2)$. Similarly, the total time spent on the calls to the subroutine **Pull** is also bounded by $O(n^2)$. $\square$

Now if we replace the call to **Dinic-Saturation** in the algorithm **Max-Flow-Dinic** by a call to **Karzanov-Saturation**, we get Karzanov's maximum flow algorithm, which is given in Figure 2.13.

By Theorem 2.2.2, Lemma 2.2.4, and the analysis for the algorithm **Layered-Network**, we get immediately,

**Theorem 2.2.5** *Karzanov's maximum flow algorithm (algorithm* **Max-Flow-Karzanov***) runs in time* $O(n^3)$

**Algorithm. Max-Flow-Karzanov**

Input: a flow network $G$
Output: a maximum flow $f$ in $G$

1. let $f(u,v) = 0$ for all pairs $(u,v)$ of vertices in $G$;
2. construct the residual network $G_f$;
3. **while** there is a positive flow in $G_f$ **do**
   call **Layered-Network** to construct the layered
      network $L_0$ for $G_f$;
   call **Karzanov-Saturation** on $L_0$ to construct a
      shortest saturation flow $f^*$ in $G_f$;
   let $f = f + f^*$ be the new flow in $G$;
   construct the residual network $G_f$;

Figure 2.13: Karzanov's algorithm for maximum flow

## 2.3 Preflow method

The preflow method was proposed more recently by Goldberg and Tarjan [55]. To describe this method, let us start by reviewing Karzanov's maximum flow algorithm. Consider the subroutine **Push**$(v, f^v)$ in Karzanov's algorithm. On each vertex $u$ in the search, we try to push the requested amount $fl[u]$ of flow through the vertex $u$. Since the operation uses only the local neighborhood relation for the vertex $u$ and is independent of the global structure of the underlying flow network, the operation is very efficient. On the other hand, Karzanov's algorithm seems a bit too conservative: it pushes a flow of value $fl[u]$ through the vertex $u$ only when it knows that this amount of flow can be pushed all the way to the sink $t$. Moreover, it pushes flow only along the shortest paths from $s$ to $t$. Can we generalize this efficient operation so that a larger amount of flow can be pushed through each vertex along all possible paths (not just shortest paths) to the sink $t$?

Think of the flow network as a system of water pipes, in which vertices correspond to pipe junctions. Each junction has a position such that water only flows from higher positions to lower positions. In particular, the position of the sink is the lowest, and the position of the source is always higher than that of the sink. For each junction $u$, we have a requested amount $e[u]$ of flow to be pushed through the junction, which at the beginning is supposed to be stored in a private reservoir for the junction $u$. Now if there is a non-saturated pipe $[u, w]$ such that the position of $w$ is lower than $u$, then a certain amount of flow can be pushed along the pipe $[u, w]$. The pushed flow seems to flow to the sink since the sink has the lowest position. In case no further push is possible and there are still junctions with non-empty reservoir, we "lift" the positions for the junctions with non-empty reservoir to make further pushes possible from these junctions.

How do we decide the requested flow $e[u]$ for each junction? According to the principal "higher pressure induces higher speed," we try to be a bit aggressive, and let $e[u]$ be the amount requested from the incoming pipes to $u$, which may be larger than the capacity of $u$. It may eventually turn out that this request is too much for the junction $u$ to let through. In this case, we observe that with the position of the junction $u$ increased, eventually the position of $u$ is higher than the source. Thus, the excess flow $e[u]$ will go backward in the flow network all the way back to the source.

Let us formulate the above idea using the terminologies in the MAXIMUM FLOW problem.

**Definition 2.3.1** Let $G = (V, E)$ be a flow network with source $s$ and sink $t$. A function $f$ on vertex pairs of $G$ is a *preflow* if $f$ satisfies the capacity constraint property, the skew symmetry property (see Definition 2.1.2), and the following *nonnegative excess property*: $\sum_{v \in V} f(v, w) \geq 0$ for all $w \in V - \{s\}$. The amount $\sum_{v \in V} f(v, w)$ is called the *excess flow* into the vertex $w$, denoted $e[w]$.

The excess flow $e[w]$ is the amount of further flow we want to push through the vertex $w$.

The concept of the residual network can be extended to preflows in a flow network. Formally, suppose that $f$ is a preflow in a flow network $G$, then the *residual network* $G_f$ (with respect to $f$) has the same vertex set as $G$, and $[u, v]$ is an edge of capacity $cap_f(u, v) = cap(u, v) - f(u, v)$ in $G_f$ if and only if $cap(u, v) > f(u, v)$.

Note that both the processes described above of pushing along non-saturated edges and of sending excess flow back to the source can be implemented by a single type of push operation on edges in the residual network: if an edge $[u, v]$ is non-saturated then the edge $[u, v]$ also exists in the residual network, and if there is a positive flow request from $s$ to $u$ along a path that should be sent back to the source, then the reversed path from $u$ to $s$ is a path in the residual network.

Each flow network $G$ is also associated with a *height function* $h$ such that for any vertex $u$ of $G$, $h(u)$ is a non-negative integer. To facilitate the analysis of our algorithms, we require that the height function be more

**Algorithm.** $\textbf{Push}(u, w)$

{ Applied only when $(f, h)$ is a preflow scheme, $h(u) = h(w) + 1$, $cap_f(u, w) > 0$, and $e[u] > 0$. }

1. $f_0 = \min\{e[u], cap_f(u, w)\}$;
2. $f(u, w) = f(u, w) + f_0$; $f(w, u) = -f(u, w)$;
3. $e[w] = e[w] + f_0$; $e[u] = e[u] - f_0$.

Figure 2.14: Pushing a flow along the edge $[u, w]$

restricted when it is associated with a preflow, as given in the following definition.

**Definition 2.3.2** Let $G$ be a flow network, $f$ be a preflow in $G$, and $h$ be a height function for $G$. The pair $(f, h)$ is a *preflow scheme* for $G$ if for any edge $[u, w]$ in the residual network $G_f$, we have $h(u) \leq h(w) + 1$.

Now we are ready to describe our first basic operation on a flow network with a preflow scheme.

The operation $\textbf{Push}(u, w)$ is applied to a pair of vertices $u$ and $w$ in a flow network $G$ only when the following three conditions all holds: (1) $h(u) = h(w) + 1$; (2) $cap_f(u, w) > 0$; and (3) $e[u] > 0$. In this case, the **Push** operation pushes as much flow as possible (i.e., the minimum of $cap_f(u, w)$ and $e[u]$) along the edge $[u, w]$, and update the values for $e[u]$, $e[w]$, and $f(u, w)$. In other words, the operation shifts an amount of $\min\{cap_f(u, w), e[u]\}$ excess value, along edge $[u, w]$, from $u$ to $w$. A formal description of the operation $\textbf{Push}(u, w)$ is given in Figure 2.14.

Note that the operation $\textbf{Push}(u, w)$ does not change the height function value. On the other hand, the operation does change the flow value $f(u, w)$, and the excess values of the vertices $u$ and $w$. The following lemma shows that the operation **Push** preserves a preflow scheme.

**Lemma 2.3.1** *Let $(f, h)$ be a preflow scheme for a flow network $G$. Suppose that the operation $\textbf{Push}(u, w)$ is applicable to a pair of vertices $u$ and $w$ in $G$. Then after applying the operation $\textbf{Push}(u, w)$, the new values for $f$ and $h$ still make a preflow scheme.*

PROOF. The $\textbf{Push}(u, w)$ operation only changes values related to vertices $u$ and $w$ and to edge $[u, w]$. For the new value for $f$, (1) the capacity

**Algorithm. Lift**$(v)$

{ Applied only when $(f, h)$ is a preflow scheme, $e[v] > 0$, and for all out-going edges $[v, w]$ from $v$ (there is at least one such an edge), $h(v) < h(w) + 1$. }

1. let $w_0$ be a vertex with the minimum $h(w_0)$ over all $w$ such that $[v, w]$ is an edge in $G_f$;
2. $h(v) = h(w_0) + 1$;

Figure 2.15: Lifting the position of a vertex $v$

constraint is preserved: since $cap_f(u, w) \geq f_0$ and $cap_f(u, w)$ is equal to $cap(u, w)$ minus the old flow value $f(u, w)$, thus, $cap(u, w)$ is not smaller than the old flow value $f(u, w)$ plus $f_0$, which is the new flow value $f(u, w)$; (2) the skew symmetry property is preserved by step 2; and (3) the non-negative excess property is preserved: the excess $e[u]$ of $u$ is decreased by $f_0 \leq e[u]$ and the excess $e[w]$ is in fact increased.

To consider the constraint for the height function, note that the only possible new edge that is created by the **Push**$(u, w)$ operation is edge $[w, u]$. Since the **Push** operation does not change the height function values, we have $h[w] = h[u] - 1$, which is of course not larger than $h[u] + 1$. □

Now we consider the second basic operation, **Lift** on a preflow scheme. The **Lift** operation is applied to a vertex $v$ in a preflow scheme $(f, h)$ when the position of $v$ is too low for the **Push** operation to push a flow through $v$. Therefore, to apply a **Lift** operation on a vertex $v$, the following three conditions should all hold: (1) $e[v] > 0$; (2) there is an out-going edge $[v, w]$ from $v$ in the residual network $G_f$; and (3) for each out-going edge $[v, w]$ from $v$ in $G_f$, we have $h(v) < h(w) + 1$ (note that condition (3) is equivalent to $h(v) \neq h(w) + 1$ since for a preflow scheme $(f, h)$, we always have $h(v) \leq h(w) + 1$). The formal description of the **Lift** operation is given in Figure 2.15.

Note that the operation **Lift**$(v)$ does not change the preflow value.

**Lemma 2.3.2** *Let $(f, h)$ be a preflow scheme for a flow network $G$. Suppose that* **Lift**$(v)$ *is applicable to a vertex $v$ in $G$. Then after applying the operation* **Lift**$(v)$*, the new values for $f$ and $h$ still make a preflow scheme.*

PROOF. Since the operation **Lift**$(v)$ does not change the preflow value, we

**Algorithm. Max-Flow-GT**

Input: a flow network $G$ with source $s$ and sink $t$
Output: a maximum flow $f$ in $G$

1. **for** each vertex $v$ in $G$ **do** $h(v) = 0$; $e[v] = 0$;
2. **for** each pair of vertices $u$ and $w$ in $G$ **do** $f(u, w) = 0$;
3. $h(s) = n$;
4. **for** each out-going edge $[s, v]$ from $s$ **do**
   $f(s, v) = -f(v, s) = cap(s, v)$; $e[v] = cap(s, v)$;
5. **while** there is a vertex $v \neq s, t$ with $e[v] > 0$ **do**
5.1. pick a vertex $v \neq s, t$ with $e[v] > 0$;
5.2. **if Push** is applicable to an edge $[v, w]$ in $G_f$
   **then Push**$(v, w)$ **else Lift**$(v)$.

Figure 2.16: Golberg-Tarjan's algorithm for maximum flow

only need to verify that the new values for the height function $h$ still make a preflow scheme with the preflow $f$. For this, we only need to verify the edges in the residual network $G_f$ that have the vertex $v$ as an end.

For any in-coming edge $[u, v]$ to $v$, before the operation **Lift**$(v)$, we have $h(u) \leq h(v) + 1$. Since the **Lift**$(v)$ increases the height $h(v)$ of $v$ by at least 1, we still have $h(u) \leq h(v) + 1$ after the **Lift**$(v)$ operation.

For each out-going edge $[v, w]$ from $v$, by the choice of the vertex $w_0$, we have $h(w) \geq h(w_0)$. Thus, the new value of $h(v) = h(w_0) + 1$ is not larger than $h(w) + 1$. $\square$

Now we are ready to describe our maximum flow algorithm using the preflow method. The algorithm was developed by Goldberg and Tarjan [55]. The algorithm is given in Figure 2.16.

In the rest of this section, we first prove that the algorithm **Max-flow-GT** correctly constructs a maximum flow given a flow network, then we analyze the algorithm. Further improvement on the algorithm will also be briefly described.

**Lemma 2.3.3** *Let $f$ be a preflow in a flow network $G$. If a vertex $u_0$ has a positive excess $e[u_0] > 0$, then there is a path in the residual network $G_f$ from $u_0$ to the source $s$.*

PROOF. Let $V_0$ be the set of vertices reachable from $u_0$ in the residual network $G_f$. Consider any pair of vertices $v$ and $w$ such that $v \in V_0$ and $w \notin V_0$. Since in $G_f$, the vertex $v$ is reachable from the vertex $u_0$ while the vertex $w$ is not reachable from $u_0$, there is no edge from $v$ to $w$ in $G_f$. Thus, $cap_f(v, w) = 0$, which by the definition implies that in the original flow network $G$ we have $f(v, w) = cap(v, w) \geq 0$, $f(w, v) \leq 0$. Therefore, for any $v \in V_0$ and $w \notin V_0$, we must have $f(w, v) \leq 0$.

Now consider

$$\sum_{v \in V_0} e[v] = \sum_{v \in V_0} \sum_{w \in V} f(w, v) = \sum_{v \in V_0} \sum_{w \in V_0} f(w, v) + \sum_{v \in V_0} \sum_{w \notin V_0} f(w, v)$$

The first term in the right hand side is equal to 0 by the skew symmetry property, and the second term in the right hand side is not larger than 0 since for any $v \in V_0$ and $w \notin V_0$ we have $f(w, v) \leq 0$. Thus, we have

$$\sum_{v \in V_0} e[v] \leq 0 \tag{2.2}$$

Now if the source $s$ is not in the set $V_0$, then since $e[u_0] > 0$ and by the non-negative excess property, for all other vertices $v$ in $V_0$, we have $e[v] \geq 0$, we derive $\sum_{v \in V_0} e[v] > 0$, contradicting to the relation in (2.2).

In conclusion, the source $s$ must by reachable from the vertex $u_0$, i.e., there must be a path from $u_0$ to $s$ in the residual network $G_f$. $\square$

Now we are ready to prove the correctness for the algorithm **Max-flow-GT**.

**Lemma 2.3.4** *Goldberg-Tarjan's maximum flow algorithm (the algorithm* **Max-Flow-GT** *in Figure 2.16) stops with a maximum flow $f$ in the given flow network $G$.*

PROOF. Steps 1-4 initialize the values for the functions $f$ and $h$. It is easy to verify that after these steps, $f$ is a preflow in the flow network $G$. To verify that at this point $(f, h)$ is a preflow scheme, note that $f$ has positive value only on the out-going edges from the source $s$, and $f$ saturates all these egdes. Thus, in the residual network $G_f$, the source $s$ has no out-going edges. Moreover, all vertices have height 0 except the source $s$, which has height $n$. Therefore, if $[u, v]$ is an edge in the residual network $G_f$, then $u \neq s$ and $h(u) = 0$. In consequence, for any edge $[u, v]$ in the residual network $G_f$, we must have $h(u) \leq h(v) + 1$. This shows that at the end of step 4 in the algorithm, $(f, h)$ is a preflow scheme for the flow network $G$.

An execution of the **while** loop in step 5 applies either a **Push** or a **Lift** operation. By Lemma 2.3.1 and Lemma 2.3.2, if $(f, h)$ is a preflow scheme for $G$ before the operations, then the new values for $f$ and $h$ after these operations still make a preflow scheme. Inductively, before each execution of the **while** loop in step 5, the values of $f$ and $h$ make $(f, h)$ a preflow scheme for the flow network $G$.

We must show the validity for step 5.2, i.e., if the operation **Push** does not apply to any out-going edge $[v, w]$ from the vertex $v$, then the operation **Lift** must apply. By step 5.1, we have $e[v] > 0$. By Lemma 2.3.3, there must be an outgoing edge $[v, w]$ from $v$ in the residual network $G_f$. Therefore, if the operation **Push** does not apply to any out-going edge $[v, w]$ from $v$, then we must have $h(v) \neq h(w) + 1$ for any such an edge. Since $(f, h)$ is a preflow scheme, $h(v) \leq h(w) + 1$. Thus, $h(v) \neq h(w) + 1$ implies $h(v) < h(w) + 1$ for all out-going edges $[v, w]$ from $v$. Now this condition ensures that the **Lift**$(v)$ operation must apply.

Now we prove that when the algorithm **Max-Flow-GT** stops, the resulting preflow $f$ is a maximum (regular) flow in the flow network $G$. The algorithm stops when the condition in step 5 fails. That is, for *all* vertices $v \neq s, t$, we have $e[v] = 0$. By the definition, this means that for all vertices $v \neq s, t$, we have $\sum_{w \in V} f(w, v) = 0$. Thus, the function $f$ satisfies the flow conservation property. Since $f$ is a preflow, it also satisfies the capacity constraint property and the skew symmetry property. Thus, when the algorithm stops, the resulting preflow $f$ is actually a regular flow in $G$.

To prove that this flow $f$ is maximum, by Theorem 2.1.4, we only have to prove that there is no path from the source $s$ to the sink $t$ in the residual network. Suppose the opposite that there is a path $P$ from $s$ to $t$ in the residual network $G_f$. Without loss of generality, we assume that the path $P$ is a simple path of length less than $n$. Let $P = \{v_0, v_1, \ldots, v_k\}$, where $v_0 = s$, $v_k = t$ and $k < n$. Since $[v_i, v_{i+1}]$, for $0 \leq i \leq k-1$, are edges in $G_f$ and the pair $(f, h)$ is a preflow scheme for $G$, we must have $h(v_i) \leq h(v_{i+1}) + 1$ for all $1 \leq i \leq k-1$. Therefore,

$$
\begin{aligned}
h(s) = h(v_0) &\leq h(v_1) + 1 \\
&\leq h(v_2) + 2 \\
&\quad \cdots\cdots \\
&\leq h(v_k) + k \\
&= h(t) + k
\end{aligned}
$$

Since $h(s) = n$ and $h(t) = 0$ the above formula implies $n \leq k$ contradicting the assumption $n > k$. Therefore, there is no path from the source $s$ to the sink $t$ in the residual network $G_f$. Consequently, the flow $f$ obtained by the algorithm is a maximum flow in the flow network $G$. $\square$

Now we analyze the algorithm **Max-Flow-GT**. The running time of the algorithm is dominated by step 5, which is in turn dominated by the number of subroutine calls to the operations **Push** and **Lift**. Thus, a bound on the subroutine calls will imply a bound to the running time of the algorithm.

**Lemma 2.3.5** *Let $(f, h)$ be the final preflow scheme obtained by the algorithm* **Max-Flow-GT** *for a given flow network $G$. Then for any vertex $v_0$ of $G$, we have $h(v_0) \leq 2n - 1$.*

PROOF. If the vertex $v_0$ never gets a positive excess $e[v] > 0$, then it is never picked up in step 5.1. Thus, the height of $v$ is never changed. Since the initial height of each vertex in $G$ is at most $n$, the lemma holds.

Now suppose that the vertex $v_0$ does get a positive excess $e[v_0] > 0$ during the execution of the algorithm. Let $(f', h')$ be the last preflow scheme during the execution of the **while** loop in step 5 in which $e[v_0] > 0$. By Lemma 2.3.3, there is a path from $v_0$ to $s$ in the residual network $G_{f'}$. Let $P' = \{v_0, v_1, \ldots, v_k\}$ be a simple path in $G_{f'}$ from $v_0$ to $s$, where $v_k = s$ and $k \leq n-1$. Then since $(f', h')$ is a preflow scheme, we have $h'(v_i) \leq h'(v_{i+1}) + 1$, for all $0 \leq i \leq k-1$. This implies immediately $h'(v_0) \leq h'(s) + k \leq 2n-1$, since the height of the source $s$ is $n$ and is never changed.

Since the execution of the **while** loop on the preflow scheme $(f', h')$ changes the excess $e[v_0]$ on $v_0$ from a positive value to 0, this execution must be a **Push** on an out-going edge from $v_0$, which does not change the height value for $v_0$. After this execution, since $e[v_0]$ remains 0 so $v_0$ is never picked by step 5.1 and its height value is never changed. In consequence, at the end of the algorithm, the height value $h(v_0)$ for the vertex $v_0$ is still not larger than $2n - 1$. $\square$

Now we can derive a bound on the number of calls to the subroutine **Lift** by the algorithm **Max-Flow-GT**.

**Lemma 2.3.6** *The total number of calls to the subroutine* **Lift** *by the algorithm* **Max-Flow-GT** *is bounded by $2n^2 - 8$.*

PROOF. By Lemma 2.3.5, the height of a vertex $v$ cannot be larger than $2n - 1$. Since each call **Lift**$(v)$ increases the height of $v$ by at least 1, the number of calls **Lift**$(v)$ on the vertex $v$ cannot be larger than $2n - 1$. Note that the operation **Lift** does not apply on the source $s$ and the sink $t$. Thus, the total number of calls to **Lift** in the algorithm **Max-Flow-GT** cannot be larger than $(2n-1)(n-2) \le 2n^2 - 8$ (note $n \ge 2$). $\square$

Lemma 2.3.5 can also be used to bound the number of calls to the subroutine **Push**.

**Lemma 2.3.7** *The total number of calls to the subroutine* **Push** *by the algorithm* **Max-Flow-GT** *is bounded by* $O(n^2m)$.

PROOF. A subroutine call to $\mathbf{Push}(u,w)$ is a *saturating push* if it makes $f(u,w) = cap(u,w)$. Otherwise, the subroutine call is called a *non-saturating push*.

We first count the number of saturating pushes.

Suppose that we have a saturating push $\mathbf{Push}(u,w)$ along the edge $[u,w]$ in the residual network $G_f$. Let the value of $h(u)$ at this moment be $h_0$. After this push, there is no edge from $u$ to $w$ in the residual network $G_f$. When can the next saturating push $\mathbf{Push}(u,w)$ from $u$ to $w$ happen again? To make it possible, we first have to make $[u,w]$ an edge again in the residual network $G_f$. The only way to make $[u,w]$ an edge in the residual netowrk is to push a positive flow from $w$ to $u$, i.e., a call $\mathbf{Push}(w,u)$ must apply. In order to apply $\mathbf{Push}(w,u)$, the height of vertex $w$ must be larger than the height of vertex $u$. Therefore, the new value of $h(w)$ must be at least $h_0 + 1$. Now after the call $\mathbf{Push}(w,u)$, a new edge $[u,w]$ is created in the residual network. Now similarly, if we want to apply another call to $\mathbf{Push}(u,w)$ to the edge $[u,w]$ (no matter if it is saturating on non-saturating), the height $h(u)$ must be larger than the new height $h(w)$ of $w$. That is, the new height $h(u)$ is at lest $h_0 + 2$. Therefore, between two consecutive saturating pushes $\mathbf{Push}(u,w)$ along the edge $[u,w]$ in the residual network $G_f$, the height of the vertex $u$ is increased by at least 2. By Lemma 2.3.5, the height of the vertex $u$ is bounded $2n-1$. Thus, the total number of saturating pushes $\mathbf{Push}(u,w)$ for the pair of vertices $u$ and $w$ is bounded by $(2n-1)/2+1 < n+1$. Now note that a push $\mathbf{Push}(u,w)$ applies only when $[u,w]$ is an edge in the residual network $G_f$, which implies that either $[u,w]$ or $[w,u]$ is an edge in the original network $G$. Thus, there are at most $2m$ pairs of vertices $u$ and $w$ on which a saturating push $\mathbf{Push}(u,w)$ can apply. In summary, the total number of saturating pushes in the algorithm **Max-Flow-GT** is bounded by $2m(n+1)$.

Now we count the number of non-saturating pushes.

Let $V_+$ be the set of vertices $u$ in $V - \{s,t\}$ such that $e[u] > 0$. Consider the value $\beta_+ = \sum_{u \in V_+} h(u)$. The value $\beta_+$ is 0 before step 5 of the algorithm

**Max-Flow-GT** starts since at that point all vertices except $s$ have height 0. The value $\beta_+$ is again 0 at the end of the algorithm since at this point all vertices $u \neq s, t$ have $e[u] = 0$. Moreover, the value $\beta_+$ can never be negative during the execution of the algorithm. Now we consider how each of the operations **Push** and **Lift** affects the value $\beta_+$.

If $\mathbf{Push}(u, w)$ is a non-saturating push, then before the operation $u \in V_+$ and $h(u) = h(w) + 1$, and after the operation, the excess $e[u]$ becomes 0 so the vertex $u$ is removed from the set $V_+$. Note that the vertex $w$ may be a new vertex added to the set $V_+$. Thus, the operation subtracts a value $h(u)$ from $\beta_+$, and *may* add a value $h(w) = h(u) - 1$ to $\beta_+$. In any case, the non-saturating push $\mathbf{Push}(u, w)$ decreases the value $\beta_+$ by at least 1.

If $\mathbf{Push}(u, w)$ is a saturating push, then $u$ belongs to the set $V_+$ before the operation but $w$ may be added to the set $V_+$ by the operation. By Lemma 2.3.5, the height $h(w)$ of $w$ is bounded by $2n - 1$. Thus, each saturating push increases the value $\beta_+$ by at most $2n - 1$.

Now consider the $\mathbf{Lift}(v)$ operation. When the operation $\mathbf{Lift}(v)$ applies, the vertex $v$ is in the set $V_+$. Since the height $h(v)$ of $v$ cannot be larger than $2n - 1$, the operation $\mathbf{Lift}(v)$ increases the value $\beta_+$ by at most $2n - 1$. In consequence, the operation $\mathbf{Lift}(v)$ increases the value $\beta_+$ by at most $2n-1$.

By Lemma 2.3.6, the total number of calls to the subroutine **Lift** is bounded by $2n^2 - 8$, and by the first part in this proof, the total number of saturating pushes is bounded by $2m(n + 1)$. Thus, the total value of $\beta_+$ increased by the calls to **Lift** and by the calls to saturating pushes is at most

$$(2n - 1)(2n^2 - 8 + 2m(n + 1)) \leq 4n^3 + 6n^2 m$$

Since each non-saturating push decreases the value $\beta_+$ by at least 1 and the value $\beta_+$ is never less than 0, we conclude that the total number of non-saturating pushes by the algorithm is bounded by $4n^3 + 6n^2 m = O(n^2 m)$.

This completes the proof for the lemma. $\square$

Now we conclude the discussion for the algorithm **Max-Flow-GT**.

**Theorem 2.3.8** *Goldberg and Tarjan's maximum flow algorithm (algorithm* **Max-Flow-GT** *in Figure 2.16) constructs a maximum flow for a given flow network in time $O(n^2 m)$.*

PROOF. The correctness of the algorithm has been given in Lemma 2.3.4. The running time of the algorithm is dominated by step 5, for which we give a detailed analysis.

We keep two 2-dimensional arrays $f[1..n, 1..n]$ and $cap[1..n, 1..n]$ for the flow value and the capacity for the original flow network $G$, respectively, so that the flow value between a pair of vertices, and the capacity of an edge in the residual network $G_f$ can be obtained and modified in constant time. Similarly, we keep the arrays $h[1..n]$ and $e[1..n]$ for the height and excess for vertices in $G$ so that the values can be obtained and modified in constant time.

The residual network $G_f$ is represented by an adjacency list $L_f$ such that for each vertex $v$ in $G_f$, the edges $[v, w]$ with $h(v) = h(w) + 1$ appear in the front of the list $L_f[v]$. Finally, we also keep a list $OF$ for the vertices $u$ in $G$ with $e[u] > 0$.

With these data structures, the condition in step 5 can by checked in constant time (simply check if the list $OF$ is empty), and step 5.1 takes a constant time to pick a vertex $v$ from the list $OF$. Since the edges $[v, w]$ with $h(v) = h(w) + 1$ appear in the front of the list $L_f[v]$, in constant time, we can check if the operation **Push** applies to an out-going edge from $v$.

For each **Push**$(u, w)$ operation, the modification of the flow values and the excess values can be done in constant time. Moreover, if $[w, u]$ was not an edge in $G_f$ (this can be checked by comparing the values $f[w, u]$ and $cap[w, u]$) then the operation **Push**$(u, w)$ creates a new edge $[w, u]$ in the residual network $G_f$. This new edge $[w, u]$ shoud be added to the *end* of the list $L_f[w]$ since $h(w) = h(u) - 1 \neq h(u) + 1$. In conclusion, each **Push** operation takes time $O(1)$. By Lemmma 2.3.7, the total number of **Push** operations executed by the algorithm **Max-Flow-GT** is bounded by $O(n^2 m)$. Thus, the total time the algorithm **Max-Flow-GT** spends on the **Push** operations is bounded by $O(n^2 m)$.

Now consider the **Lift** operation. A **Lift**$(v)$ operation needs to find a vertex $w_0$ with minimum $h(w_0)$ in the list $L_f[v]$, which takes time $O(n)$. After increasing the value of $h(v)$, we need to check each in-coming edge $[u, v]$ to $v$ to see if now $h(u) = h(v) + 1$, and to check each out-going edge $[v, w]$ from $v$ to see if now $h(v) = h(w) + 1$. If so, the edge should be moved to the front of the list for the proper vertex in $L_f$. In any case, all these can be done in time $O(m)$. According to Lemma 2.3.6, the total number of **Lift** operations executed by the algorithm **Max-Flow-GT** is bounded by $2n^2 - 8$. Thus, the total time the algorithm **Max-Flow-GT** spends on the **Lift** operations is also bounded by $O(n^2 m)$.

This concludes that the running time of the algorithm **Max-Flow-GT** is bounded by $O(n^2 m)$. $\square$

We point out that it has been left totally open for the order of the vertices selected by step 5.1 in the algorithm **Max-Flow-GT**, which gives us further opportunity for improving the running time of the algorithm. In fact, with a more careful selection of the vertex $v$ in step 5.1 and a more efficient data structure, it can be shown that running time of the algorithm **Max-Flow-GT** can be improved to $O(nm \log(n^2/m))$. For dense flow networks with $m = \Omega(n^2)$ edges, the bound $O(nm \log(n^2/m))$ is as good as $O(n^3)$, the bound for Karzanov's maximum flow algorithm, while for sparse flow networks with $m = O(n)$ edges, the bound $O(nm \log(n^2/m))$ becomes $O(n^2 \log n)$, much lower than $O(n^3)$. A description of this improvement can be found in [55].

## 2.4 Final remarks

Before closing the chapter, we give several remarks on the MAXIMUM FLOW problem. First, we show that the MAXIMUM FLOW problem is closely related to a graph cut problem. This is given by the classical *Max-Flow-Min-Cut* theorem. Then, we briefly mention the updated status in the research in maximum flow algorithms.

### Max-Flow-Min-Cut theorem

Let $G = (V, E)$ be a directed and positively weighted graph. A *cut* in $G$ is a partition of the vertex set $V$ of $G$ into an orderd pair $(V_1, V_2)$ of two non-empty subsets $V_1$ and $V_2$, i.e., $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \emptyset$. The *weight* of the cut $(V_1, V_2)$ is the sum of the weights of the edges $[u, w]$ with $u \in V_1$ and $w \in V_2$.

The MIN-CUT problem is to find a cut of minimum weight for a given directed and positively weighted graph. Formally,

MIN-CUT $= \langle I_Q, S_Q, f_Q, opt_Q \rangle$

$I_Q$: the set of directed and positively weighted graphs $G$

$S_Q$: $S_Q(G)$ is the set of cuts for $G$

$f_Q$: $f_Q(G, C)$ is equal to the weight of the cut $C$ for $G$

$opt_Q$: min

A more restricted MIN-CUT problem is on flow networks. We say that $(V_1, V_2)$ is a *cut* for a flow network $G$ if $(V_1, V_2)$ is a cut for $G$ when $G$ is regarded as a directed and positively weighted graph, where the edge weight equals the edge capacity in $G$, such that the source $s$ of $G$ is in $V_1$ and the sink $t$ of $G$ is in $V_2$. We now consider the MIN-CUT problem on flow networks.

**Definition 2.4.1** The *capacity* of a cut $(V_1, V_2)$ for a flow network $G$ is the weight of the cut:

$$cap(V_1, V_2) = \sum_{u \in V_1, w \in V_2} cap(v, w)$$

A cut for $G$ is *minimum* if its capacity is the minimum over all cuts for $G$.

Now we are ready to prove the following classical theorem.

**Theorem 2.4.1 (Max-Flow Min-Cut Theorem)** *For any flow network $G = (V, E)$, the value of a maximum flow in $G$ is equal to the capacity of a minimum cut for $G$.*

PROOF. Let $f$ be a flow in the flow network $G$ and let $(V_1, V_2)$ be a cut for $G$. By the definition, we have $|f| = \sum_{w \in V} f(s, w)$. By the flow conservation property, we also have $\sum_{w \in V} f(v, w) = 0$ for all vertices $v \in V_1 - \{s\}$. Therefore, we have

$$\begin{aligned} |f| &= \sum_{w \in V} f(s, w) = \sum_{v \in V_1, w \in V} f(v, w) \\ &= \sum_{v \in V_1, w \in V_1} f(v, w) + \sum_{v \in V_1, w \in V_2} f(v, w) \end{aligned}$$

By the skew symmetry property, $f(v, w) = -f(w, v)$ for all vertices $v$ and $w$ in $V_1$. Thus, the first term in the last expression of the above equation is equal to 0. Thus,

$$|f| = \sum_{v \in V_1, w \in V_2} f(v, w) \tag{2.3}$$

By the capacity constraint property of a flow, we have

$$|f| \leq \sum_{v \in V_1, w \in V_2} cap(v, w) = cap(V_1, V_2)$$

Since this inequality holds for all flows $f$ and all cuts $(V_1, V_2)$, we conclude

$$\max \{|f| : f \text{ is a flow in G}\} \leq \min \{cap(V_1, V_2) : (V_1, V_2) \text{ is a cut for G}\}$$

To prove the other direction, let $f$ be a maximum flow in the network $G$ and let $G_f$ be the residual network. By Theorem 2.1.4, there is no path from the source $s$ to the sink $t$ in the residual network $G_f$. Define $V_1$ to be the set of vertices that are reachable from the source $s$ in the graph $G_f$ and let $V_2$ be the set of the rest vertices. Then, $s \in V_1$ and $t \in V_2$ so $(V_1, V_2)$ is a cut for the flow network $G$. We have

$$cap(V_1, V_2) = \sum_{v \in V_1, w \in V_2} cap(v, w) = \sum_{[v,w] \in E, v \in V_1, w \in V_2} cap(v, w)$$

Consider an edge $[v, w]$ in $G$ such that $v \in V_1$ and $w \in V_2$. Since $[v, w]$ is not an edge in the residual network $G_f$ (otherwise, the vertex $w$ would be reachable from $s$ in $G_f$), we must have $f(v, w) = cap(v, w)$. On the other hand, for $v \in V_1$ and $w \in V_2$ suppose $(v, w)$ is not an edge in $G$, then we have $cap(v, w) = 0$ thus $f(v, w) \leq 0$. However, $f(v, w) < 0$ cannot hold since otherwise we would have $cap(v, w) > f(v, w)$ thus $[v, w]$ would have been an edge in the residual graph $G_f$, contradicting the assumption $v \in V_1$ and $w \in V_2$. Thus, in case $(v, w)$ is not an edge in $G$ and $v \in V_1$ and $w \in V_2$, we must have $f(v, w) = 0$. Combining these discussions, we have

$$ap(V_1, V_2) = \sum_{[v,w] \in E, v \in V_1, w \in V_2} cap(v, w) = \sum_{v \in V_1, w \in V_2} f(v, w)$$

By equation 2.3, the last expression is equal to $|f|$. This proves

$$\max \{|f| : f \text{ is a flow in G}\} \geq \min \{cap(V_1, V_2) : (V_1, V_2) \text{ is a cut for G}\}$$

The proof of the theorem is thus completed. $\square$

The Max-Flow-Min-Cut theorem used to be used to show that a maximum flow in a flow network can be found in finite number of steps (since there are only $2^n - 2$ cuts for a flow network of $n$ vertices). With more efficient maximum flow algorithms being developed, now the theorem can be used in the opposite direction — to find a minimum cut for a flow network. In fact, the proof of Theorem 2.4.1 has described such an algorithm: given a flow network $G = (V, E)$, construct a maximum flow $f$ in $G$, let $V_1$ be the set of vertices reachable from the source $s$ in the residual network $G_f$ and let $V_2 = V - V_1$, then $(V_1, V_2)$ is a minimum cut for the flow network $G$. Thus, the MIN-CUT problem on flow networks can be solved in time $O(n^3)$, if we use, for example, Karzanov's maximum flow algorithm to solve the MAXIMUM FLOW problem.

| Date | Authors | Time Complexity |
|---|---|---|
| 1960 | Edmonds and Karp | $O(nm^2)$ |
| 1970 | Dinic | $O(n^2m)$ |
| 1974 | Karzanov | $O(n^3)$ |
| 1977 | Cherkasky | $O(n^2\sqrt{m})$ |
| 1978 | Galil | $O(n^{5/3}m^{2/3})$ |
| 1980 | Sleator and Tarjan | $O(nm \log n)$ |
| 1986 | Goldberg and Tarjan | $O(nm \log (n^2/m))$ |

Figure 2.17: Maximum flow algorithms

The MIN-CUT problem for general graphs can be solved via the algorithm for the problem for flow networks: given a general directed and positively weighted graph $G$, we fix a vertex $v_1$ in $G$. Now for each other vertex $w$, we construct a flow network $G_w$ that is $G$ with $v_1$ the source and $w$ the sink, and construct another flow network $G_w^r$ that is $G$ with $w$ the source and $v_1$ the sink. Then we find the minimum cuts for the flow networks $G_w$ and $G_w^r$. Since in a minimum cut $(V_1, V_2)$ for $G$ as a general graph, the vertex $v_1$ must be in one of $V_1$ and $V_2$ and some vertex $w$ must be in the other, when we construct the minimum cuts for the flow networks $G_w$ and $G_w^r$ for this particular $w$, we will find a minimum cut for $G$ as a general graph. Thus, the MIN-CUT problem for general graphs can be solved in polynomial time.

We remark that the maximum version of the cut problem, i.e., the MAX-CUT problem that finds a cut of maximum weight in a directed and positively weighted graph, is much more difficult that the MIN-CUT problem. The MAX-CUT problem will be discussed in more detail in later chapters.

## Updated status

Since the introduction of the MAXIMUM FLOW problem by Ford and Fulkerson about thirty years ago, efficient solution of the MAXIMUM FLOW problem has been a very active research area in theoretical computer science. In this chapter, we have discussed two major techniques for maximum flow algorithms. Further extension and improvement on these techniques are studied. The table in Figure 2.17 gives a selected sequence of maximum flow algorithms developed over the past thirty years. For further discussion on maximum flow algorithms, the reader is referred to the survey by Goldberg, Tardos, and Tarjan [54].

# Chapter 3

# Graph Matching

A problem that often arises is an "effective assignment", which optimally pairs up objects based on given requirements. Recall the OPTIMAL COURSE ASSIGNMENT problem introduced in Chapter 1.

> 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.

Similar problems arise in worker/job assignment, person/position decision, boy/girl engagement, and so forth.

These problems have been formulated into a general fundamental problem on graphs that has been widely studied. Given a graph $G$, a *matching* $M$ in $G$ is a subset of edges in $G$ such that no two edges in $M$ share a common end. The GRAPH MATCHING problem is to find a matching given a graph such that the matching has the maximum cardinality. Using our formulation, the GRAPH MATCHING problem is given as a 4-tuple.

> GRAPH MATCHING $= \langle I_Q, S_Q, f_Q, opt_Q \rangle$
>
> $I_Q$: the set of all undirected graphs $G$
>
> $S_Q$: $S_Q(G)$ is the set of all matchings in the graph $G$
>
> $f_Q$: $f_Q(G, M)$ is the number of edges in the matching $M$
>
> $opt_Q$: max

For example, the OPTIMAL COURSE ASSIGNMENT problem can be converted into the GRAPH MATCHING problem as follows. We let each teacher and each course be a vertex in the graph $G$. There is an edge between a teacher and a course if the teacher can teach the course. A solution to the GRAPH MATCHING problem on the graph $G$ corresponds to an optimal course assignment for the school.

The GRAPH MATCHING problem has attracted attention because of its intuitive nature and its wide applicability. Its solution in the general case involves sophisticated and beautiful combinatorial mathematics. In this chapter, we will concentrate on analysis principles and fundamental algorithms for the problem. Further applications of the problem will be mentioned in later chapters.

Throughout this chapter, we will assume that $n$ is the number of vertices and $m$ is the number of edges in the considered graph.

## 3.1 Augmenting paths

We first prove a fundamental theorem for the GRAPH MATCHING problem. Based on this fundamental theorem, a general scheme of developing algorithms for the problem will be presented.

Let $M$ be a matching in a graph $G = (V, E)$. A vertex $v$ is a *matched vertex* (with respect to the matching $M$) if $v$ is an endpoint of an edge in $M$, otherwise, the vertex is an *unmatched vertex*. An edge $e$ is a *matching edge* if $e \in M$ and an *untamed edge* if $e \notin M$.

An important notion for the GRAPH MATCHING problem is that of an augmenting path.

**Definition 3.1.1** Let $M$ be a matching in a graph $G$. An *augmenting path* (with respect to $M$) is a simple path $p = \{u_0, u_1, \ldots u_{2h+1}\}$ of odd length such that the end vertices $u_0$ and $u_{2h+1}$ are unmatched and that the edges $[u_{2i-1}, u_{2i}]$ are matching edges, for $i = 1, \ldots h$.

Figure 3.1 shows a graph $G$ and a matching $M$ in $G$, where heavy lines are for matching edges in $M$ and light lines are for untamed edges not in $M$. The path $\{v_3, u_3, v_4, u_5, v_2, u_2\}$ is an augmenting path in $G$ with respect to $M$.

Note that an augmenting path is relative to a fixed matching $M$. Moreover, by the definition in an augmenting path $p = \{u_0, u_1, \ldots, u_{2h+1}\}$, the edges $[u_{2i-2}, u_{2i-1}]$ are untamed edges, for $i = 1, \ldots h + 1$. The number

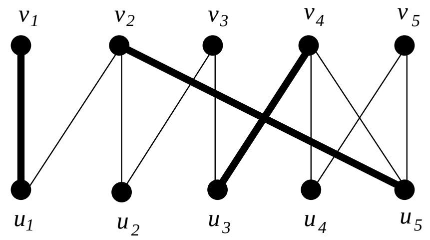



Figure 3.1: Alternating path and augmenting path in a matching

of untamed edges on an augmenting path $p$ is exactly one larger than the number of matching edges on $p$.

The following theorem serves as a fundamental theorem in the study of graph matching and graph matching algorithms.

**Theorem 3.1.1** *Let $G$ be a graph and let $M$ be a matching in $G$. $M$ is the maximum if and only if there is no augmenting path in $G$ with respect to $M$.*

PROOF. Suppose that there is an augmenting path $p = \{u_0, u_1, \ldots, u_{2h+1}\}$ in the graph $G$ with respect to the matching $M$. Consider the set of edges $M' = M \oplus p$, where $\oplus$ is the *symmetric difference* defined by $A \oplus B = (A - B) \cup (B - A)$. In other words, the set $M' = M \oplus p$ is obtained from the set $M$ by first deleting all matching edges on $p$ then adding all untamed edges on $p$. Since the number of matching edges on $p$ is one less than the number of untamed edges on $p$, the number of edges in $M'$ is one more than that in $M$. It is also easy to check that $M'$ is also a matching in $G$: $M' = M \oplus p = (M - p) \cup (p - M)$, for any two edges $e_1$ and $e_2$ in $M'$, (1) if both $e_1$ and $e_2$ are in $M - p$ then they are in $M$ so have no common endpoint because $M$ is a matching; (2) if both $e_1$ and $e_2$ are in $p - M$ then $e_1$ and $e_2$ have no common endpoint because $p$ is an augmenting path; and (3) if $e_1$ is in $M - p$ and $e_2$ is in $p - M$ then $e_1$ cannot have an endpoint on $p$ since the two endpoints of $p$ are unmatched and all other vertices on $p$ are matched by edges on $p$.

Therefore, $M'$ is a matching larger than the matching $M$. This proves that if there is an augmenting path $p$ with respect to $M$, then the matching $M$ cannot be the maximum.

Conversely, suppose that the matching $M$ is not maximum. Let $M_{\max}$ be a maximum matching. Then $|M_{\max}| > |M|$. Consider the graph $G_0 = M_{\max} \oplus M = (M - M_{\max}) \cup (M_{\max} - M)$. No vertex in $G_0$ has degree larger than 2. In fact, if a vertex $v$ in $G_0$ had degree larger than 2, then at least

**Algorithm. Maximum Matching**

Input: an undirected graph $G = (V, E)$
Output: a maximum matching $M$ in $G$

1. $M = \emptyset$;
2. **while** there is an augumenting path in $G$ **do**
   find an augmenting path $p$;
   construct the matching $M = M \oplus p$ with one more edge

Figure 3.2: General algorithm constructing graph matching

two edges incident on $v$ belong to either $M$ or $M_{\max}$, contradicting the fact that both $M$ and $M_{\max}$ are matchings in $G$. Therefore, each component of $G_0$ must be either a simple path, or a simple cycle. In each simple cycle in $G_0$, the number of edges in $M_{\max} - M$ should be exactly the same as the number of edges in $M - M_{\max}$. For each simple path in $G_0$, either the number of edges in $M - M_{\max}$ is the same as the number of edges in $M_{\max} - M$ (in this case, the path has an even length), or the number of edges in $M - M_{\max}$ is one more than the number of edges in $M_{\max} - M$, or the number of edges in $M_{\max} - M$ is one more than the number of edges in $M - M_{\max}$. Since $|M_{\max}| > |M|$, we conclude that there is at least one path $p = \{u_0, u_1, \ldots, u_{2h+1}\}$ in $G_0$ in which the number of edges in $M_{\max} - M$ is one more than the number of edges in $M - M_{\max}$. Note that the endpoint $u_0$ of the path $p$ must be unmatched in $M$. In fact, since $[u_0, u_1] \in M_{\max} - M$, if $u_0$ is matched in $M$ by an edge $e$, we must have $e \neq [u_0, u_1]$. Now since $u_0$ has degree 1 in $G_0$, $e \notin G_0$, $e$ is also contained in $M_{\max}$. This would make the vertex $u_0$ incident on two edges $[u_0, u_1]$ and $e$ in the matching $M_{\max}$. Similar reasoning shows that the vertex $u_{2h+1}$ is also unmatched in $M$. In consequence, the path $p$ is an augmenting path in the graph $G$ with respect to the matching $M$.

This completes the proof. $\square$

Based on Theorem 3.1.1, a general maximum matching algorithm can be derived. The algorithm is given in Figure 3.2. Most algorithms for the Graph Matching problem are essentially based on this basic algorithm.

Since a matching in a graph $G$ of $n$ vertices cannot contain more than $n/2$ edges, the **while** loop in the algorithm **Maximum Matching** will be executed at most $n/2$ times. Therefore, if an augmenting path can be constructed in time $O(t(n))$ when a matching is given, then the Graph

MATCHING problem can be solved in time $O(nt(n))$.

## 3.2 Bipartite graph matching

In this section, we consider the GRAPH MATCHING problem for a class of graphs with a special structure. We illustrate how an augmenting path is constructed for a matching in a graph with this special structure.

**Definition 3.2.1** A graph $G = (V, E)$ is *bipartite* if the vertex set $V$ of $G$ can be partitioned into two disjoint subsets $V = V_1 \cup V_2$ such that every edge in $G$ has one end in $V_1$ and the other end in $V_2$.

The graph in Figure 3.1 is an example of a bipartite graph. The bipartiteness of a graph can be tested using a standard graph search algorithm (depth first search or breadth first search), which tries to color the vertices of the given graph by two colors such that no two adjacent vertices are colored with the same color. Obviously, a graph is bipartite if and only if it can be colored in this way with two colors. Another property of a bipartite graph $G$ is that $G$ does not contain any cycle of odd length.

Many applications can be formulated by matching in bipartite graphs. For example, the OPTIMAL COURSE ASSIGNMENT problem, as well as the worker/job assignment, person/position decision, and boy/girl engagement problems mentioned at beginning of this chapter, can all be formulated by matchings in bipartite graphs.

Now we discuss how an augmenting path can be constructed in a bipartite graph. Let $M$ be a matching in a bipartite graph $G$. The idea of constructing an augmenting path with respect to $M$ is fairly natural: we start from each unmatched vertex $v_0$, and try to find out whether an augmenting path starts from the vertex $v_0$. For this, we perform a process similar to the *breadth first search*, starting from the vertex $v_0$. The vertices encountered in the search process are classified into *odd level vertices* and *even level vertices*, depending upon their distance to the vertex $v_0$ in the search tree. For an even level vertex $v$, the process tries to extend the augmenting path by adding an untamed edge. The vertex $v$ may be incident on several untamed edges and we do not know which is the one we want. Thus, we record all of them — just as in breadth first search we record all unvisited neighbors of the current vertex. For an odd level vertex $w$, the process either concludes with an augmenting path (when $w$ is unmatched) or tries to extend the augmenting path by adding a matching edge (note that if $w$ is a matched vertex then there is a unique matching edge that

**Algorithm. Bipartite Augment**

Input: a bipartite graph $G$ and a matching $M$ in $G$

1. **for** each vertex $v$ **do** $lev(v) = -1$;
2. **for** each unmatched vertex $v_0$ with $lev(v_0) = -1$ **do**
    $lev(v_0) = 0$; $Q = \emptyset$; $Q \leftarrow v_0$; $\{Q$ is a queue$\}$
    **while** $Q \neq \emptyset$ **do**
        $u \leftarrow Q$;
2.1.        **case 1.** $lev(u)$ is even
            **for** each neighbor $w$ of $u$ with $lev(w) = -1$ **do**
2.1.1.            $lev(w) = lev(u) + 1$; $dad(w) = u$; $Q \leftarrow w$;
2.2.        **case 2.** $lev(u)$ is odd
            **if** $u$ is unmatched
2.2.1.        **then** an augmenting path is found; stop.
            **else** let $[u, w]$ be the edge in $M$
2.2.2.            $lev(w) = lev(u) + 1$; $dad(w) = u$; $Q \leftarrow w$;
3. no augmenting path in $G$.

Figure 3.3: Finding an augmenting path in a bipartite graph

is incident on $w$). Note that in case of an odd level vertex $w$, the search process is different from the standard breadth first search: the vertex $w$ may have several unvisited neighbors, but we only record the one that matches $w$ in $M$ and ignore the others.

If the search process starting from $v_0$ fails in finding another unmatched vertex, then it starts from another unmatched vertex and performs the same search process.

A formal description of this search process is given in Figure 3.3.

According to the algorithm, each vertex $v$ is assigned a level number $lev(v)$ such that $v$ has a vertex $dad(v)$ at level $lev(v) - 1$ as its father. In particular, if $lev(v)$ is odd, then the edge $[dad(v), v]$ is an untamed edge while if $lev(v)$ is even then $v$ is the unique child of its father $dad(v)$ and the edge $[dad(v), v]$ is a matching edge. The level is also used to record whether a vertex $v$ has been visited in the process. A vertex $v$ is unvisited if and only if $lev(v) = -1$.

Although the algorithm **Bipartite Augment** looks similar to the standard breadth first search, step 2.2 in the algorithm makes a significant difference. We first study the structure of the subgraph constructed by the algorithm **Bipartite Augment**.

**Lemma 3.2.1** *For each vertex $v_0$ picked at step 2, the algorithm* **Bipartite Augment** *constructs a tree $T(v_0)$ rooted at $v_0$ and sharing no common vertices with other trees. The tree structure is given by the relation $dad(\cdot)$.*

PROOF. To prove the lemma, we only need to show that by the time the vertex $w$ is to be included in the tree $T(v_0)$ in step 2.1.1 or step 2.2.2, the vertex $w$ is still unvisited. This fact is clear when the father $u$ of $w$ is an even level vertex, as described in the algorithm.

The case is much more complicated when the vertex $u$ is an odd level vertex. According to the algorithm, in this case the edge $[u, w]$ is in the matching $M$. Suppose the opposite, that $lev(w) \neq -1$ before the execution of step 2.2.2. We will derive a contradiction as follows. Without loss of generality, assume that $w$ is the first such vertex encountered in the execution of the algorithm.

The vertex $w$ cannot belong to any previously constructed tree $T(v_0')$ — otherwise, the vertex $u$ is either the father of $w$ in $T(v_0')$ (in case $lev(w)$ is even) or the unique child of $w$ in $T(v_0')$ (in case $lev(w)$ is odd). Thus, the vertex $u$ would have also been visited before the construction of the tree $T(v_0)$. This contradicts our assumption that $w$ is the first such vertex.

So we must have $w$ in the same tree $T(v_0)$. Since $u$ is the current vertex in this search process similar to the breadth-first search, the level $lev(w)$ of the vertex $w$ is at most $lev(u) + 1$.

The level $lev(w)$ cannot be $lev(u) + 1$ since otherwise, the vertex $w$ was included in the tree $T(v_0)$ because of another vertex $u' = dad(w)$ at level $lev(u)$ in the tree $T(v_0)$. Since $lev(u)$ is odd, this implies that $[u', w]$ is in the matching $M$, contradicting the assumption that $[u, w]$ is in $M$.

Thus, $lev(w) = lev(u) - k$ for some nonnegative integer $k$. If $k$ is an even number, then let $v$ be the least common ancestor of $w$ and $u$ in the tree $T(v_0)$ (note that by our assumption, the structure constructed so far for $T(v_0)$ is a tree before the edge $[u, w]$ is added). Then the tree path from $v$ to $w$ followed by the edge $[w, u]$ then followed by the tree path from $u$ to $v$ would form a cycle of odd length, contradicting the fact that the graph $G$ is a bipartite graph.

So $lev(w)$ must be an even number and $lev(w) < lev(u)$ since $lev(u)$ is odd. However, an even $lev(w)$ implies that $[dad(w), w]$ is in $M$ thus $dad(w)$ must be $u$, contradicting the fact $lev(w) < lev(u)$.

Therefore, in any case the inequality $lev(w) \leq lev(u) + 1$ is impossible. Consequently, before the execution of step 2.2.2, the vertex $w$ must be unvisited. This completes the proof. □

Now we are ready to show the correctness of the algorithm **Bipartite Augment**.

**Theorem 3.2.2** *On a bipartite graph $G$ and a matching $M$ in $G$, the algorithm* **Bipartite Augment** *stops at step 2.2.1 if and only if there is an augmenting path in $G$ with respect to $M$.*

PROOF. Suppose that the algorithm stops at step 2.2.1. with an odd level vertex $u$ that is unmatched. Since for every even level vertex $w$, the tree edge $[dad(w), w]$ is a matching edge while for every odd level vertex $w$, the tree edge $[dad(w), w]$ is an untamed edge, the tree path from the root $v_0$ to $u$ is an augmenting path.

To prove the other direction, suppose that there is an augmenting path with respect to $M$, we show that the algorithm **Bipartite Augment** must find one and stop at step 2.2.1. Assume the opposite, that the algorithm does not find an augmenting path and stops at step 3.

Then for each vertex $v_0$ picked at step 2, the constructed tree $T(v_0)$ contains no unmatched vertices except $v_0$. Thus, every unmatched vertex will be picked at step 2 of the algorithm and a tree rooted at it is constructed. In particular, for every unmatched vertex $v_0$, we must have $lev(v_0) = 0$. Now assume that $p = \{v_0, v_1, \ldots, v_{2k+1}\}$ is an augmenting path, where both $v_0$ and $v_{2k+1}$ are unmatched vertices.

We show that for the vertex $v_i$, $0 \le i \le 2k+1$, on the augmenting path $p$, the level $lev(v_i)$ has the same parity as $i$. Since $v_0$ is an unmatched vertex, we have $lev(v_0) = 0$. Thus, it suffices to show that $lev(v_i)$ and $lev(v_{i+1})$ have different parity for all $i$.

If the edge $[v_i, v_{i+1}]$ is a matching edge, then by the algorithm and Lemma 3.2.1, the vertex $v_{i+1}$ is either the father of $v_i$ (in case $lev(v_i)$ is even) or the unique child of $v_i$ (in case $lev(v_i)$ is odd) in the same tree. Thus, $lev(v_i)$ and $lev(v_{i+1})$ must have different parity.

Thus, if we let $v_i$ and $v_{i+1}$ be the first pair on $p$ such that $lev(v_i)$ and $lev(v_{i+1})$ have the same parity, then the edge $[v_i, v_{i+1}]$ must be an untamed edge. In other words, $i$ must be an even number. Since $v_i$ and $v_{i+1}$ make the first such a pair, the level number $lev(v_i)$ of $v_i$ is also an even number.

The vertex $v_i$ and $v_{i+1}$ cannot belong to the same tree — otherwise since both $lev(v_i)$ and $lev(v_{i+1})$ are even, the edge $[v_i, v_{i+1}]$ plus the tree paths from $v_i$ and $v_{i+1}$, respectively, to their least common ancestor would form a cycle of odd length, contradicting the fact that the graph $G$ is bipartite.

Suppose that $v_i$ is in the tree $T(v_0')$, then $v_{i+1}$ cannot belong to a tree constructed later than $T(v_0')$ — otherwise, at the time the vertex $v_i$ is processed in step 2.1 of the algorithm, the vertex $v_{i+1}$ is still unvisited. Since $v_i$ is an even level vertex, the vertex $v_{i+1}$ would have been made a child of $v_i$, which would have included $v_{i+1}$ in the same tree $T(v_0')$.

Therefore, the vertex $v_{i+1}$ must belong to a tree $T(v_0'')$ constructed before the tree $T(v_0')$. Since $lev(v_i)$ and $lev(v_{i+1})$ have the same parity, $v_{i+1}$ is an even level vertex. However, according to the algorithm, this would imply that when the vertex $v_{i+1}$ is processed in step 2 of the algorithm, the vertex $v_i$ is still unvisited. Thus, the vertex $v_i$ would have been included in the tree $T(v_0'')$.

Summarizing the above discussion, we conclude that there is no pair $v_i$ and $v_{i+1}$ on the augmenting path $p$ such that $lev(v_i)$ and $lev(v_{i+1})$ have the same parity. In consequence, for each $v_i$ on $p$, $lev(v_i)$ has the same parity as $i$. However, this would imply that the end vertex $v_{2k+1}$ of the augmenting path $p$ is an odd level vertex, contradicting the fact that $v_{2k+1}$ is an unmatched vertex so we must have $lev(v_{2k+1}) = 0$.

This contradiction shows that if there is an augmenting path with respect to the matching $M$, then the algorithm **Bipartite Augment** must stop at step 2.2.1 and produce an augmenting path.

This completes the proof of the lemma. □

Based on Theorem 3.1.1, algorithm **Maximum Matching**, algorithm **Bipartite Augment**, and Theorem 3.2.2, an algorithm can be designed for the GRAPH MATCHING problem on bipartite graphs.

**Theorem 3.2.3** *The* GRAPH MATCHING *problem on bipartite graphs can be solved in time $O(nm)$.*

PROOF. We can use an array $match[1..n]$ to represent a matching in the graph such that $match[i] = j$ if and only if $[i, j]$ is a matching edge. Thus, checking whether an edge is in a matching, adding an edge to a matching, and deleting an edge from a matching can all be done in constant time.

The algorithm **Bipartite Augment** processes each edge in the graph at most twice, one from each end of the edge, and each process takes constant time. Moreover, once the algorithm stops at step 2.2.1, the found augmenting path can be easily constructed by tracing the tree path (via the array $dad(\cdot)$) from the current vertex $u$ to the root of the tree. Therefore, the running time of the algorithm **Bipartite Augment** is bounded by $O(m)$.

Each execution of the **while** loop in step 2 of the algorithm **Maximum Matching** calls the algorithm **Bipartite Augment** to find an augmenting path and constructs a larger matching with one more edge. Since a matching in a graph of $n$ vertices contains no more than $n/2$ edges, we conclude that the algorithm **Maximum Matching** runs in time $O(nm)$ if it uses the algorithm **Bipartite Augment** as a subroutine to find augmenting paths. □

## 3.3 Maximum flow and graph matching

There is an interesting relation between the MAXIMUM FLOW problem and the GRAPH MATCHING problem on bipartite graphs, which leads to a more efficient algorithm for the GRAPH MATCHING problem on bipartite graphs. In this section, we discuss this relation.

Let $G$ be a flow network. A flow $f$ in $G$ is an *integral flow* if $f(u, w)$ is an integer for every pair of vertices $u$ and $w$ in $G$. If an integral flow $f$ is a maximum flow in $G$, then we call $f$ an *integral maximum flow.*

**Lemma 3.3.1** *Let $G$ be a flow network in which all edge capacities are integers. Then there is an integral maximum flow in $G$.*

PROOF. It is easy to see that if all edge capacities in $G$ are integers, then for any integral flow $f$ in $G$, the edge capacities in the residual graph $G_f$ are also all integers.

Consider Dinic's maximum flow algorithm, which is given in Figure 2.8. For the convenience of our reference here and later discussion, we present the algorithm again in Figure 3.4. The algorithm starts with an integral flow $f = 0$. Inductively, suppose that the flow $f$ is integral; thus all edge capacities in the residual network $G_f$ are integers. In an execution of the **while** loop in step 3, the algorithm saturates all shortest paths from the source to the sink in the residual network $G_f$. Since all edge capacities in $G_f$ are integers, the flow $f^*$ constructed by the subroutine **Dinic-Saturation** is also integral. This will make the new flow $f + f^*$ in $G$ integral for the next execution of the **while** loop. In conclusion, Dinic's algorithm ends up with a maximum flow that is integral. □

Given a bipartite graph $B = (V_1 \cup V_2, E)$, where every edge in $E$ has one end in $V_1$ and the other end in $V_2$, we can construct a flow network $G$ by adding a source vertex $s$ and a sink vertex $t$, adding a directed edge from $s$ to each of the vertices in $V_1$, adding a directed edge from each of the vertices in $V_2$ to $t$, giving each original edge in $B$ a direction from $V_1$ to

**Algorithm. Max-Flow-Dinic**

Input: a flow network $G$
Output: a maximum flow $f$ in $G$

1. let $f(u, v) = 0$ for all pairs $(u, v)$ of vertices in $G$;
2. construct the residual network $G_f$;
3. **while** there is a positive flow in $G_f$ **do**
   - 3.1. call **Layered-Network** to construct the layered network $L_0$ for $G_f$;
   - 3.2. call **Dinic-Saturation** on $L_0$ to construct a shortest saturation flow $f^*$ in $G_f$;
   - 3.3. let $f = f + f^*$ be the new flow in $G$;
   - 3.4. construct the residual network $G_f$;

Figure 3.4: Dinic's algorithm for maximum flow

$V_2$, and setting the capacity of each edge in $G$ to 1. See Figure 3.5 for an example of this construction.

The connection of the MAXIMUM FLOW problem and the GRAPH MATCHING problem is given by the following theorem.

**Theorem 3.3.2** *Let $G$ be the flow network constructed from the bipartite graph $B$ as above, and let $f$ be an integral maximum flow in $G$. Then the set of edges $e = [u, w]$ in $G$ satisfying $u \in V_1$, $w \in V_2$, and $f(u, w) = 1$ constitutes a maximum matching in $B$.*

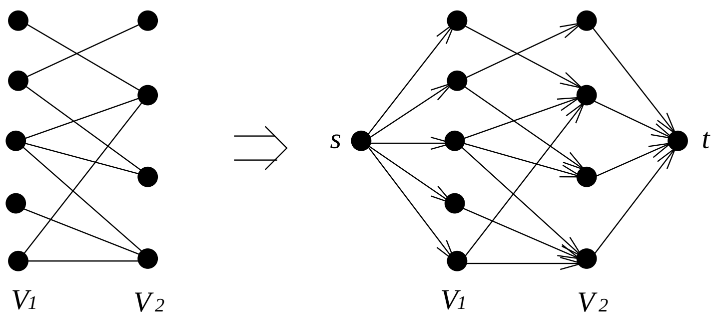



Figure 3.5: From a bipartite graph to a flow network

PROOF. Consider the set $M$ of edges $e = [u, w]$ such that $u \in V_1$, $w \in V_2$, and $f(u, w) = 1$. No two edges in $M$ share a common vertex. For example, if both $[u, w]$ and $[u, w']$ are in the set $M$, then since $f(u, w) = f(u, w') = 1$ and the vertex $u$ has only one incoming edge, whose capacity is 1, we would have $\sum_{v \in V} f(u, v) > 0$, contradicting the flow conservation constraint. Therefore, the set $M$ is a matching in the bipartite graph $B$. Moreover, it is also easy to see that the number of edges $|M|$ in $M$ is equal to the flow value $|f|$.

We show that the matching $M$ is actually a maximum matching in $B$. Let $M' = \{[u_i, w_i] \mid i = 1, \ldots, k\}$ be any matching of $k$ edges in $B$, where $u_i \in V_1$ and $w_i \in V_2$. Then the function $f'$ defined by $f'(s, u_i) = 1$, $f'(u_i, w_i) = 1$, $f'(w_i, t) = 1$ for all $i$, and $f'(u, w) = 0$ for all other pairs $u$ and $w$ is clearly an integral flow in $G$ with flow value $k$. Since $f$ is a maximum flow, we have $k \leq |f| = |M|$. That is, the number of edges in the matching $M'$ cannot be larger than $|M|$. This proves that $M$ is a maximum matching. $\square$

Thus, the GRAPH MATCHING problem on bipartite graphs can be reduced to the MAXIMUM FLOW problem. In particular, since the algorithm **Max-Flow-Dinic** produces an integral maximum flow on flow networks with integral edge capacities, by Theorem 2.2.3, the GRAPH MATCHING problem on bipartite graphs can be solved in time $O(n^2m)$ using the algorithm **Max-Flow-Dinic**.

Observe that $O(n^2m)$ is the worst case time complexity for the algorithm **Max-Flow-Dinic** on general flow networks. It is thus natural to suspect that the algorithm may have a better bound when it is applied to the flow networks of simpler structure such as the one we constructed for a bipartite graph. For this, we first introduce a definition.

**Definition 3.3.1** A flow network $G$ is a *simple flow network* if it satisfies the following three conditions:

1. the capacity of each edge of $G$ is 1;
2. every vertex $v \neq s, t$ either has only one incoming edge or has only one outgoing edge; and
3. there is no pair of vertices $u$ and $w$ such that both $[u, w]$ and $[w, u]$ are edges in $G$.

Clearly, the flow network $G$ constructed above from a bipartite graph is a simple flow network.

**Lemma 3.3.3** *Let $G = (V, E)$ be a simple flow network and let $f$ be an integral flow in $G$. Then the residual graph $G_f$ is also a simple flow network.*

PROOF. Since $G$ is a simple flow network and $f$ is an integral flow, the flow value $f(u,w)$ on an edge $[u,w]$ in $G$ must be either 0 or 1. By this it is easy to verify that in the residual network $G_f$, every edge has capacity exactly 1 and there is no pair $u$ and $w$ such that both $[u,w]$ and $[w,u]$ are edges in $G_f$.

Consider any vertex $w$ in $G$, $w \neq s,t$. Suppose that the vertex $w$ has only one incoming edge $e = [v,w]$.

If $f(v,w) = 0$ then $f(w,u) = 0$ for all $u \in V$ by the flow conservation constraint. Thus, in the residual graph $G_f$, $e$ is still the only incoming edge for the vertex $w$.

If $f(v,w) = 1$ then since $f$ is integral there must be an outgoing edge $[w,u]$ of $w$ such that $f(w,u) = 1$, and for all other outgoing edges $[w,u']$ we must have $f(w,u') = 0$. Therefore, in the residual graph $G_f$, the edge $[v,w]$ disappears and we add another outgoing edge $[w,v]$, and the edge $[w,u]$ disappears and we add a new incoming edge $[u,w]$, which is the unique incoming edge of the vertex $w$ in $G_f$.

The case that the vertex $w$ has only one outgoing edge can be proved similarly.

This completes the proof. □

Therefore, all residual networks $G_f$ in algorithm **Max-Flow-Dinic** are simple flow networks if $G$ is a simple flow network. In particular, the layered network $L_0$ in step 3.1, which is a subgraph of $G_f$, is a simple flow network.

According to the discussion in section 2.2, the layered network in step 3.1 of the algorithm **Max-Flow-Dinic** can be constructed in time $O(m)$. It is also easy to see that steps 3.3 and 3.4 take time at most $O(m)$ (note that the residual network $G_f$ has at most $2m$ edges if the original flow network $G$ has $m$ edges). We show below that step 3.2 of the algorithm can be implemented more efficiently when the layered network $L_0$ is a simple flow network.

**Lemma 3.3.4** *If the layered network $L_0$ is a simple network, then the shortest saturation flow $f^*$ for $G_f$ in step 3.2 of the algorithm* **Max-Flow-Dinic** *can be constructed in time $O(m)$.*

PROOF. We modify the algorithm **Dinic-Saturation** in Figure 2.7 to construct the shortest saturation flow $f^*$. Each time when we search for a path $p$ from $s$ to $t$ in the layered network $L_0$, we keep a queue $q(\cdot)$ for the vertices on $p$. In case we hit a dead end $q(k)$ that is not the sink $t$, we delete the edge $[q(k-1), q(k)]$ from $L_0$ and continue the search from the vertex $q(k-1)$. Once a path $p$ from $s$ to $t$ is found, we can saturate *all* edges in $p$ by a flow of value 1 along the path $p$, thus deleting all edges on $p$ from the layered network $L_0$. This is because the layered network $L_0$ is a simple flow network so all of its edges have capacity 1. Therefore, every edge encountered in this search process will be deleted from $L_0$. Since it takes only constant time to process each edge in this search process, we conclude that in time $O(m)$, all shortest paths in the layered network $L_0$ are saturated. $\square$

According to Lemma 3.3.3 and Lemma 3.3.4, the running time of each execution of the **while** loop in step 3 of the algorithm **Max-Flow-Dinic** can be bounded by $O(m)$ when it is applied to a simple flow network. This together with Theorem 2.2.2 concludes that the running time of the algorithm **Max-Flow-Dinic** on simple flow networks is bounded by $O(nm)$. In particular, this shows that the GRAPH MATCHING problem on bipartite graphs can be solved in time $O(nm)$, which matches the time complexity of the algorithm we developed in Section 3.2 for the problem.

In fact, the algorithm **Max-Flow-Dinic** can do even better on simple flow networks. For this, we first prove a lemma.

**Lemma 3.3.5** *Let $G = (V, E)$ be a simple flow network, and let $f$ be a maximum flow in $G$, let $l$ be the length of the shortest path from the source $s$ to the sink $t$ in $G$. Then $l \leq n/|f| + 1$, where $n$ is the number of vertices in $G$.*

PROOF. Define $V_i$ to be the set of vertices $v$ of distance $dist(v) = i$ from the source $s$ in $G$.

Fix an $i$, $0 \leq i \leq l-2$. Define $L_i$ and $R_i$ by

$$L_i = \bigcup_{j=0}^{i} V_j \qquad \text{and} \qquad R_i = V - L_i$$

That is, $L_i$ is the set of vertices whose distance from the source $s$ is at most $i$ and $R_i$ is the set of vertices whose distance from the source $s$ is larger than $i$. Since $dist(t) = l$ and $0 \leq i \leq l-2$, $(L_i, R_i)$ is a cut of the flow network $G$.

We claim that for any edge $e = [v, w]$ of $G$ such that $v \in L_i$ and $w \in R_i$, we must have $v \in V_i$ and $w \in V_{i+1}$. In fact, if $v \in V_h$ for some $h < i$, then the distance from $s$ to $w$ cannot be larger than $h + 1 \leq i$. This would imply that $w$ is in $L_i$. Thus, $v$ must be in $V_i$. Now the edge $[v, w]$ gives $dist(w) \leq dist(v) + 1 = i + 1$. Now since $w \in R_i$ we must have $dist(w) > i$. Therefore, $dist(w) = i + 1$ and $w \in V_{i+1}$.

Now consider

$$\begin{aligned} |f| &= \sum_{w \in V} f(s, w) = \sum_{v \in L_i, w \in V} f(v, w) \\ &= \sum_{v \in L_i, w \in L_i} f(v, w) + \sum_{v \in L_i, w \in R_i} f(v, w) \\ &= \sum_{v \in L_i, w \in R_i} f(v, w) \end{aligned}$$

The second equality is because of the flow conservation constraint $\sum_{w \in V} f(v, w) = 0$ for all $v \in L_i - \{s\}$ (note $t \notin L_i$), and the fourth equality is because of the skew symmetry constraint $f(v, w) = -f(w, v)$ for all $v$ and $w$ in $L_i$. Note that if $v$ is not in $V_i$ or $w$ is not in $V_{i+1}$ then $[v, w]$ is not an edge in $G$ thus we must have $f(v, w) \leq 0$. This combined with the above equation gives

$$|f| \leq \sum_{v \in V_i, w \in V_{i+1}} f(v, w)$$

Now since $G$ is a simple flow network, there is at most one unit flow through a vertex $v \neq s, t$. Therefore, for $i = 0$ (i.e., $V_i = \{s\}$), we have $|f| \leq |V_1|$, and for $1 \leq i \leq l - 2$, we have $|f| \leq |V_{i+1}|$. Summarizing these $l - 1$ inequalities, we get

$$(l - 1)|f| \leq |V_1| + |V_2| + \cdots + |V_{l-1}| \leq n$$

which gives immediately $l \leq n/|f| + 1$. □

Now we can give a more precise analysis on the number of executions of the **while** loop in the algorithm **Max-Flow-Dinic**.

**Lemma 3.3.6** *On a simple flow network of $n$ vertices, the* **while** *loop in step 3 of the algorithm* **Max-Flow-Dinic** *is executed at most $2\sqrt{n}+1$ times.*

PROOF. Let $f_{\max}$ be a maximum flow in $G$.

If $|f_{\max}| \leq 2\sqrt{n}$, then of course the **while** loop is executed at most $2\sqrt{n}$ times since each execution of the loop increases the flow value by at least 1.

Now assume $|f_{\max}| > 2\sqrt{n}$. Let $h_0$ be the largest integer such that after $h_0$ executions of the **while** loop, the flow $f_0$ constructed in Dinic's algorithm has value $|f_0| \leq |f_{\max}| - \sqrt{n}$. A few interesting facts about $h_0$ are

- after the $(h_0 + 1)$st execution of the **while** loop, the constructed flow has value larger than $|f_{\max}| - \sqrt{n}$;
- the value of the maximum flow in the residual network $G_{f_0}$ is $|f_{\max}| - |f_0| \geq \sqrt{n}$.

By the second fact and Lemma 3.3.5, the distance $dist(t)$ from the source $s$ to the sink $t$ in the residual network $G_{f_0}$ is bounded by $n/\sqrt{n}+1 = \sqrt{n}+1$. By Lemma 2.2.1, each execution of the **while** loop increases the distance $dist(t)$ by at least 1. Since the algorithm starts with $dist(t) > 0$, we conclude that $h_0 \leq \sqrt{n}$.

By the first fact, after the $(h_0+1)$st execution of the **while** loop, the constructed flow $f_1$ has value larger than $|f_{\max}| - \sqrt{n}$. Therefore, with another at most $\sqrt{n}$ executions of the **while** loop, starting from the flow network $G_{f_1}$, the algorithm must reach the maximum flow value $f_{\max}$ because each execution of the **while** loop increases the flow value by at least 1.

In conclusion, the total number of executions of the **while** loop in step 3 of the algorithm **Max-Flow-Dinic** on a simple flow network is bounded by $h_0 + 1 + \sqrt{n} \leq 2\sqrt{n} + 1$. This completes the proof. □

**Theorem 3.3.7** *The algorithm* **Max-Flow-Dinic** *runs in time $O(\sqrt{n}m)$ on a simple flow network of $n$ vertices and $m$ edges.*

PROOF. Follows directly from Lemma 3.3.4 and Lemma 3.3.6. □

**Corollary 3.3.8** *The* GRAPH MATCHING *problem on bipartite graphs can be solved in time $O(\sqrt{n}m)$.*

The bound $O(\sqrt{n}m)$ in Corollary 3.3.8 is still the best known bound for the algorithms solving the GRAPH MATCHING problem on bipartite graphs.

## 3.4 General graph matching

In this section, we discuss the GRAPH MATCHING problem on general graphs.

Unfortunately, the network flow method does not seem to apply easily to the GRAPH MATCHING problem on general graphs. Thus, we come back to the fundamental theorem, Theorem 3.1.1, and the basic algorithm **Maximum Matching** in Figure 3.2 for the GRAPH MATCHING problem.

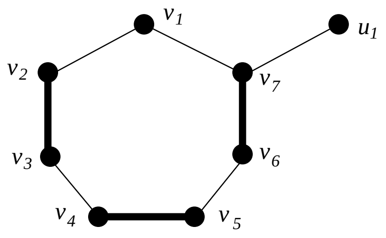



Figure 3.6: **Bipartite Augment** fails to find an existing augmenting path

All known algorithms for the GRAPH MATCHING problem on general graphs are based on Theorem 3.1.1 and the algorithm **Maximum Matching**. The main point here is how an augmenting path can be found when a matching $M$ is given in a graph $G$. For the rest of the discussion, we assume that $G$ is a fixed graph and that $M$ is a fixed matching in $G$.

Let us make a careful examination of the reason the algorithm **Bipartite Augment** in Figure 3.3 may fail in finding an existing augmenting path. Consider the matching in the graph in Figure 3.6, where heavy lines are for matching edges and light lines are for untamed edges. Suppose that we start from the unmatched vertex $v_1$. In level 1 we get two vertices $v_2$ and $v_7$ as children of vertex $v_1$ in the tree $T(v_1)$. Now since both $v_2$ and $v_7$ are odd level vertices, $v_2$ will get $v_3$ as its unique child and $v_7$ will get $v_6$ as its unique child. Note that although vertex $u_1$ is unvisited when we process vertex $v_7$, we ignore the vertex $u_1$ since $v_7$ is an odd level vertex. Now the process continues the construction of the tree $T(v_1)$, making $v_4$ a child of $v_3$ and $v_5$ a child of $v_6$. Now the process gets stuck since neither of the vertices $v_4$ and $v_5$ can use the matching edge $[v_4, v_5]$ to expand the tree $T(v_1)$. Note that this situation cannot happen for a bipartite graph as we proved in Lemma 3.2.1, in which when we process an odd level matched vertex $v$, the vertex matching $v$ in $M$ must be unvisited. Even if we ignore this abnormal phenomena and continue the process, we still do not find an augmenting path — the tree $T(u_1)$ contains a single vertex $u_1$ since the only neighbor $v_7$ of $u_1$ has been visited. On the other hand, there exists an augmenting path $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, u_1\}$ with respect to the matching $M$.

It is clear that the troubles are caused by the odd length cycle $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$. Such a cycle does not exist in a bipartite graph.

Therefore, in order to ensure that the structure we construct starting from an unmatched vertex is a tree, we need to modify the algorithm **Bipartite Augment**. For this, before we add a vertex to the tree, no matter

**Algorithm. Modified BFS**

1. **for** each vertex $v$ **do** $lev(v) = -1$;
2. **for** each unmatched vertex $v$ **do**
   $Q \leftarrow v$; $\{Q$ is a queue$\}$ $lev(v) = 0$;
3. **while** no augmenting path is found and $Q \neq \emptyset$ **do**
   $u \leftarrow Q$;
   **case 1.** $lev(u)$ is even
      **for** each neighbor $w$ of $u$ with $lev(w) = -1$ **do**
         $dad(w) = u$; $lev(w) = lev(u) + 1$; $Q \leftarrow w$;
   **case 2.** $lev(u)$ is odd
      **if** $[u, w] \in M$ and $lev(w) = -1$ **then**
         $dad(w) = u$; $lev(w) = lev(u) + 1$; $Q \leftarrow w$;

Figure 3.7: The modified breadth first search

whether it is a child of an even level vertex or an odd level vertex, we first check that the vertex is unvisited.

Another modification for the algorithm **Bipartite Augment** is based on the following observation. As shown in Figure 3.6, an augmenting path may come from an unvisited and unmatched vertex $u_1$, which cannot be identified if the construction of the tree $T(u_1)$ does not start until the construction of the tree $T(v_1)$ is completed. To resolve this problem, we will perform the breadth first fashion search starting from *all* unmatched vertices simultaneously, and construct the trees rooted at the unmatched vertices in parallel, instead of starting from a single unmatched vertex. Implementation of this modification is simple: we first put all unmatched vertices in the queue $Q$ then perform the breadth first fashion search until either an augmenting path is found or the queue $Q$ is empty. It is easy to see that this search will first construct the first level for all trees rooted at the unmatched vertices, then the second level for all trees, and so on.

We call this search process the *modified breadth first search* process. The formal description of this process is given in Figure 3.7.

The algorithm **Modified BFS** has not described how an augmenting path is found, which is discussed in the rest of this section. We call an edge in the graph $G$ a *cross-edge* is it is not a tree edge in any of the search trees constructed by the algorithm **Modified BFS**.

**Definition 3.4.1** A cross-edge $e$ is a *good cross-edge* if either $e \in M$ and $e$

links two odd level vertices in two different trees, or $e \notin M$ and $e$ links two even level vertices in two different trees.

A good cross-edge commits an augmenting path, as shown by the following lemma.

**Lemma 3.4.1** *If a good cross-edge is given, then an augmenting path with respect to the matching $M$ can be constructed in time $O(n)$.*

PROOF. Let $e = [v_{2s+1}, u_{2t+1}]$ be a good cross-edge such that $e \in M$, $\{v_0, v_1, \ldots, v_{2s+1}\}$ be the tree path in a search tree $T(v_0)$ rooted at the unmatched vertex $v_0$, and $\{u_0, u_1, \ldots, u_{2t+1}\}$ be the tree path in another search tree $T(u_0)$ rooted at the unmatched vertex $u_0$, $v_0 \neq u_0$. According to the algorithm **Modified BFS**, the edges $[v_{2i}, v_{2i+1}]$ and $[u_{2j}, u_{2j+1}]$ are untamed edges, for all $i = 0, \ldots, s$ and $j = 0, \ldots, t$, and the edges $[v_{2i+1}, v_{2i+2}]$ and $[u_{2j+1}, u_{2j+2}]$ are matching edges, for all $i = 0, \ldots, s-1$ and $j = 0, \ldots, t-1$. Therefore, the path

$$\{v_0, v_1, \ldots, v_{2s}, v_{2s+1}, u_{2t+1}, u_{2t}, \ldots, u_1, u_0\}$$

is an augmenting path. Since the augmenting path contains at most $n$ vertices, it can be easily constructed in time $O(n)$ using the array $dad(\cdot)$ given the edge $e = [v_{2s+1}, u_{2t+1}]$.

The case that the good cross-edge is an untamed edge and links two even level vertices can be proved similarly. $\square$

Lemma 3.4.1 suggests how statements can be inserted in the algorithm **Modified BFS** so that an augmenting path can be constructed. This extension is given in Figure 3.8.

It had been believed that the algorithm **Obvious Augment** was sufficient for constructing an augmenting path in a general graph until the following structure was discovered.

**Definition 3.4.2** A cross-edge $e$ is a *bad cross-edge* if either $e \in M$ and $e$ links two odd level vertices in the same search tree, or $e \notin M$ and $e$ links two even level vertices in the same search tree.

The edge $[v_4, v_5] \in M$ in Figure 3.6 is a bad cross-edge. As we have seen, when the bad cross-edge $[v_4, v_5]$ is encountered in our search process, the construction of the tree $T(v_1)$ gets stuck. Moreover, the bad cross-edge cannot be simply ignored because it may "hide" an augmenting path, as we pointed out before.

**Algorithm. Obvious Augment**

1. **for** each vertex $v$ **do** $lev(v) = -1$;
2. **for** each unmatched vertex $v$ **do** $Q \leftarrow v$; $lev(v) = 0$;
3. **while** no augmenting path is found and $Q \neq \emptyset$ **do**
    - $u \leftarrow Q$;
    - **case 1.** $lev(u)$ is even
        - **for** each neighbor $w$ of $u$ **do**
            - **if** $[u, w]$ is a good cross-edge
            - **then** an augmenting path is found; stop.
            - **else if** $lev(w) = -1$ **then**
                - $dad(w) = u$; $lev(w) = lev(u) + 1$; $Q \leftarrow w$;
    - **case 2.** $lev(u)$ is odd; let $[u, w] \in M$;
        - **if** $[u, w]$ is a good cross-edge
        - **then** an augmenting path is found; stop.
        - **else if** $lev(w) = -1$ **then**
            - $dad(w) = u$; $lev(w) = lev(u) + 1$; $Q \leftarrow w$;

Figure 3.8: Finding an augmenting path based on a good cross-edge

A similar situation may occur when an edge $e$ is a bad cross-edge and $e \notin M$. We can also construct a configuration in which the algorithm **Obvious Augment** fails in finding an existing augmenting path.

This discussion motivates the following definition.

**Definition 3.4.3** A *blossom* is a simple cycle consisting of a bad cross-edge $e = [v, v']$ together with the two unique tree paths from $v$ and $v'$ to their least common ancestor $v''$. The vertex $v''$ will be called the *base* of the blossom.

For example, the cycle $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ in Figure 3.6 is a blossom whose base is $v_1$.

**Remark.** There are a number of interesting properties of a blossom. We list those that are related to our later discussion.

- A blossom consists of an odd number of vertices. This is because either both ends $v$ and $v'$ of the bad cross-edge are odd level vertices or both $v$ and $v'$ are even level vertices. Therefore, a bipartite graph cannot contain a blossom.

- Suppose that the cycle $b = \{v_0, v_1, \ldots, v_{2s}, v_0\}$ is a blossom, where $v_0$ is the base, then the edges $[v_{2s}, v_0]$ and $[v_{2i}, v_{2i+1}]$ for all $i = 0, \ldots, s-1$, are untamed edges, and the edges $[v_{2j-1}, v_{2j}]$ for all $j = 1, \ldots, s$ are matching edges.

- If an edge $e_0$ is not contained in a blossom but is incident to a vertex $v$ in the blossom, then the edge $e_0$ cannot be a matching edge unless the incident vertex $v$ is the base of the blossom. This is because each vertex, except the base, in a blossom is incident on a matching edge in the blossom.

Identifying and constructing a blossom is easy, as described in the following lemma.

**Lemma 3.4.2** *During the execution of the algorithm* **Modified BFS**, *a bad cross-edge can be identified in constant time. Given a bad cross-edge, the corresponding blossom can be constructed in time $O(n)$.*

PROOF. For each search tree $T(v)$ starting from an unmatched vertex $v$, we attach a different marker to the vertices of $T(v)$ so that in constant time we can identify whether two given vertices belong to the same tree. This marker plus the level number $lev(v)$ enables us to identify in constant time whether a given edge $e = [v, v']$ is a bad cross-edge. Once a bad cross-edge is found, we can follow the tree edges to find the least common ancestor of $v$ and $v'$. Since a blossom contains at most $n$ vertices, we conclude that the blossom can be constructed in time $O(n)$. □

Thus, blossoms are a structure that may make the algorithm **Obvious Augment** fail. Is there any other structure that can also fool the algorithm? Fortunately, blossoms are the only such structure, as we will discuss below. We start with the following lemma.

**Lemma 3.4.3** *If a matching edge is a cross-edge, then it is either a good cross-edge or a bad cross-edge.*

PROOF. Let $e = [v, v']$ be a cross-edge such that $e$ is in $M$. The vertices $v$ and $v'$ cannot be roots of the search trees since roots of the search trees are unmatched vertices. Let $w$ and $w'$ be the fathers of $v$ and $v'$, respectively. The tree edges $[w, v]$ and $[w', v']$ are untamed edges since $[v, v']$ is a matching edge. Thus, $v$ and $v'$ must be odd level vertices. Now if $v$ and $v'$ belong to different search trees, then the edge $e$ is a good cross-edge, otherwise $e$ is a bad cross-edge. □

According to Lemma 3.4.3, once we encounter a matching edge $e$ that is a cross-edge, either we can construct an augmenting path (in case $e$ is a good cross-edge) or we can construct a blossom (in case $e$ is a bad cross-edge).

**Lemma 3.4.4** *If there is no blossom in the execution of the algorithm* **Modified BFS**, *then there is a good cross-edge if and only if there is an augmenting path.*

PROOF. By Lemma 3.4.1, if there is a good cross-edge, then there is an augmenting path that can be constructed from the good cross-edge in linear time.

Conversely, suppose there is an augmenting path $p = \{u_0, u_1, \ldots, u_{2t+1}\}$. If $t = 0$, then the path $p$ itself is a good cross-edge so we are done. Thus, assume $t > 0$. Let $v_1, \ldots, v_h$ be the roots of the search trees, processed in that order by the algorithm **Modified BFS**. Without loss of generality, we assume $u_0 = v_b$ where $b$ is the smallest index such that $v_b$ is an end of an augmenting path. With this assumption, the vertex $u_1$ is a child of $u_0$ in the search tree $T(u_0)$ rooted at $u_0$ — this is because $u_1$ must be a level 1 vertex. If $u_1$ is not a child of $u_0 = v_b$ then $u_1$ must be a child of an unmatched vertex $v_{b'}$ for some $b' < b$, thus $\{v_{b'}, u_1, u_2, \ldots, u_{2t+1}\}$ would also be an augmenting path, contradicting our assumption that $v_b$ is the first vertex from which an augmenting path starts.

If any matching edge $e$ on $p$ is a cross-edge, then by Lemma 3.4.3, $e$ is either a good cross-edge or a bad cross-edge. Since there is no blossom, $e$ must be a good cross-edge. Again the lemma is proved.

Thus, we assume that the augmenting path $p$ has length larger than 1, no matching edges on $p$ are cross-edges, and $u_1$ is a child of $u_0$ in the search tree $T(u_0)$ rooted at $u_0$.

**Case 1.** Suppose that all vertices on $p$ are contained in the search trees.

Both $u_0$ and $u_{2t+1}$ are even level vertices. Since the path $p$ is of odd length, there must be an index $i$ such that $lev(u_{i-1})$ and $lev(u_i)$ have the same parity. Without loss of generality, assume $i$ is the smallest index satisfying this condition. The edge $[u_{i-1}, u_i]$ must be a cross-edge. Thus, by our assumption, $[u_{i-1}, u_i]$ is not a matching edge.

Suppose that both $u_{i-1}$ and $u_i$ are odd level vertices, then $i \geq 2$. Since $[u_{i-2}, u_{i-1}]$ is a matching edge, $u_{i-2} \neq u_0$. Moreover, by our assumption, $[u_{i-2}, u_{i-1}]$ is a tree edge. Thus, $u_{i-2}$ is an even level vertex. Moreover, since $[u_{i-2}, u_{i-1}]$ is a matching edge, the index $i-2$ is an odd number. Now the partial path

$$p_{i-2} = \{u_0, u_1, \ldots, u_{i-2}\}$$

is of odd length and has both ends being even level vertices. This implies that there is an index $j$ such that $j \leq i-2$ and $lev(u_{j-1})$ and $lev(u_j)$ have the same parity. But this contradicts the assumption that $i$ is the smallest index satisfying this condition.

Thus, $u_{i-1}$ and $u_i$ must be even level vertices. So $[u_{i-1}, u_i]$ is either a good cross-edge or a bad cross-edge. By the assumption of the lemma, there is no blossom. Consequently, $[u_{i-1}, u_i]$ must be a good cross-edge and the lemma is proved for this case.

**Case 2.** Some vertices on $p$ are not contained in any search trees.

Let $u_i$ be the vertex on $p$ with minimum $i$ such that $u_i$ is not contained in any search trees. Then $i \geq 2$.

Suppose $[u_{i-1}, u_i] \in M$. If $u_{i-1}$ is an odd level vertex then $u_i$ would have been made the child of $u_{i-1}$. Thus $u_{i-1}$ is an even level vertex. However, since $u_{i-1}$ cannot be a root of a search tree, $u_{i-1}$ would have matched its father in the search tree, this contradicts the assumption that $u_{i-1}$ matches $u_i$ and $u_i$ is not contained in any search trees.

Thus we must have $[u_{i-1}, u_i] \notin M$. Then $[u_{i-2}, u_{i-1}]$ is in $M$. Thus, the index $i-2$ is an odd number. By our assumption, $[u_{i-2}, u_{i-1}]$ is a tree edge. If $u_{i-1}$ is an even level vertex, then $u_i$ would have been made a child of $u_{i-1}$. Thus, $u_{i-1}$ must be an odd level vertex. Since $[u_{i-2}, u_{i-1}]$ is a tree edge, $u_{i-2}$ is an even level vertex. Now in the partial path of odd length

$$p_{i-2} = \{u_0, u_1, \ldots, u_{i-2}\},$$

all vertices are contained in the search trees, and the two ends are even level vertices. Now the proof goes exactly the same as for **Case 1** — we can find a smaller index $j \leq i-2$ such that $lev(u_{j-1})$ and $lev(u_j)$ have the same parity and $[u_{j-1}, u_j]$ is a good cross-edge.

This completes the proof of the claim. □

By Lemma 3.4.4, if there is an augmenting path and if no bad cross-edge is found (thus no blossom is found), then the algorithm **Obvious Augment** will eventually find a good cross-edge. By Lemma 3.4.1, an augmenting path can be constructed in time $O(n)$ from the good cross-edge. In particular, if the graph is bipartite, then the algorithm **Obvious Augment** will always be able to construct an augmenting path if there exists one, since a bipartite graph contains no odd length cycle, thus no blossom can appear in the search process.

In order to develop an efficient algorithm for the Graph Matching problem on general graphs, we need to resolve the problem of blossoms.

Surprisingly the solution to this problem is not very difficult, based on the following "blossom shrinking" technique.

**Definition 3.4.4** Let $G$ be a graph and $M$ a matching in $G$. Let $B$ be a blossom found in the search process by the algorithm **Modified BFS**. Define $G/B$ to be the graph obtained from $G$ by "shrinking" the blossom $B$. That is, $G/B$ is a graph obtained from $G$ by deleting all vertices (and their incident edges) of the blossom $B$ than adding a new vertex $v_B$ that is connected to all vertices that are adjacent to some vertices in $B$ in the original graph $G$.

It is easy to see that given the graph $G$ and the blossom $B$, the graph $G/B$ can be constructed in linear time.

Since there is at most one matching edge that is incident to but not contained in a blossom, for a matching $M$ in $G$, the edge set $M - B$ is a matching in the graph $G/B$.

The following theorem is due to Jack Edmonds [34], who introduced the concept of blossoms. This theorem plays a crucial role in all algorithms for the GRAPH MATCHING problem on general graphs.

**Theorem 3.4.5 (Edmonds)** *Let $G$ be a graph and $M$ a matching in $G$. Let $B$ be a blossom. Then there is an augmenting path in $G$ with respect to $M$ if and only if there is an augmenting path in $G/B$ with respect to $M - B$.*

PROOF. Suppose that the blossom is $B = \{v_0, v_1, \ldots, v_s, v_0\}$, where $v_0$ is the base. We first show that the existence of an augmenting path in $G/B$ with respect to $M - B$ implies an augmenting path in $G$ with respect to $M$. Let $p_B = \{u_0, u_1, \ldots, u_t\}$ be an augmenting path in $G/B$ with respect to $M - B$ and let $v_B$ be the new vertex in $G/B$ obtained by shrinking $B$.

**Case 1.** If the vertex $v_B$ is not on the path $p_B$, then clearly $p_B$ is also an augmenting path in $G$ with respect to $M$.

**Case 2.** Suppose $v_B = u_t$. Then $v_B$ is an unmatched vertex in the matching $M - B$. Consequently, the base $v_0$ of the blossom $B$ is unmatched in the matching $M$.

If the edge $[u_{t-1}, u_t]$ in $G/B$ corresponds to the edge $[u_{t-1}, v_0]$ in $G$, then the path

$$p = \{u_0, u_1, \ldots, u_{t-1}, v_0\}$$

is an augmenting path in $G$. See Figure 3.9(a) (where heavy lines are for matching edges, light lines are for untamed edges, and curved lines are for paths of arbitrary length).

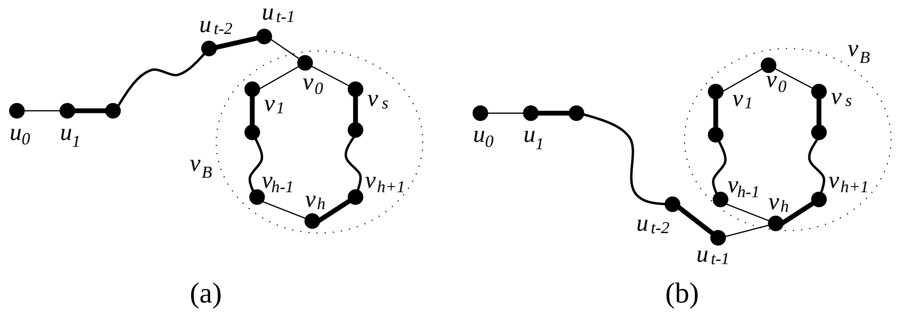



Figure 3.9: The vertex $v_B$ is an end of the augmenting path $p_B$

If the edge $[u_{t-1}, u_t]$ in $G/B$ corresponds to the edge $[u_{t-1}, v_h]$ in $G$, where $v_h$ is not the base of $B$, then one of the edges $[v_{h-1}, v_h]$ and $[v_h, v_{h+1}]$ is in the matching $M$. Without loss of generality, suppose that $[v_h, v_{h+1}]$ is in $M$. Then, the path

$$p = \{u_0, u_1, \ldots, u_{t-1}, v_h, v_{h+1}, \ldots, v_s, v_0\}$$

is an augmenting path in $G$. See Figure 3.9(b).

The case $v_B = u_0$ can be proved similarly.

**Case 3.** Suppose that $v_B = u_d$, where $0 < d < t$. Then without loss of generality, we assume that $[u_{d-1}, u_d]$ is an edge in the matching $M - B$ and $[u_d, u_{d+1}]$ is not in $M - B$ (the case where $[u_{d-1}, u_d]$ is not in $M - B$ but $[u_d, u_{d+1}]$ is in $M - B$ can be proved by reversing the augmenting path $p_B$). The matching edge $[u_{d-1}, u_d]$ in $G/B$ must correspond to a matching edge $[u_{d-1}, v_0]$ in $G$. Let the untamed edge $[u_d, u_{d+1}]$ in $G/B$ correspond to the untamed edge $[v_h, u_{d+1}]$ in $G$.

If $v_h = v_0$, then the path

$$p = \{u_0, \ldots, u_{d-1}, v_0, u_{d+1}, \ldots, u_t\}$$

is an augmenting path in $G$. See Figure 3.10(a).

If $v_h \neq v_0$, then as we proved in **Case 2**, we can assume that $[v_h, v_{h+1}]$ is an edge in $M$. Thus, the path

$$p = \{u_0, \ldots, u_{d-1}, v_0, v_s, v_{s-1}, \ldots, v_{h+1}, v_h, u_{d+1}, \ldots, u_t\}$$

is an augmenting path in $G$. See Figure 3.10(b).

Therefore, given the augmenting path $p_B$ in $G/B$, we are always able to construct an augmenting path $p$ in $G$.

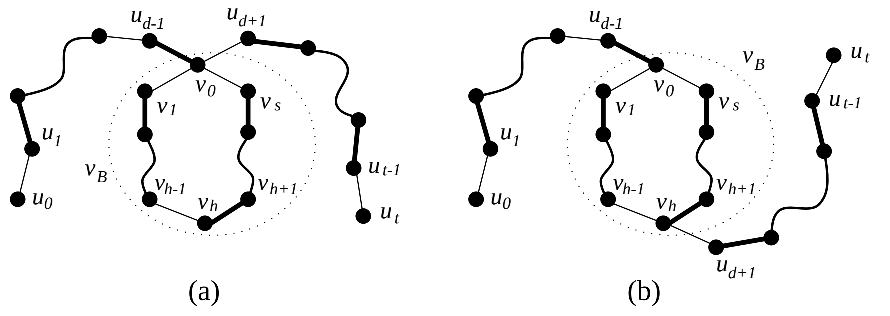



Figure 3.10: The vertex $v_B$ is an interior vertex of the augmenting path $p_B$

The proof for the other direction that the existence of an augmenting path in $G$ with respect to $M$ implies an augmenting path in $G/B$ with respect to $M - B$ is rather complicated based on a case by case analysis. We omit the proof here. □

**Corollary 3.4.6** *Let $G$ be a graph, $M$ be a matching in $G$, and $B$ be a blossom. Given an augmenting path in $G/B$ with respect to $M - B$, an augmenting path in $G$ with respect to $M$ can be constructed in time $O(n)$.*

PROOF. The proof follows directly from the construction given in the proof of Theorem 3.4.5. □

Now the idea is fairly clear for how we can find an augmenting path: given the graph $G$ and the matching $M$, we apply the modified breadth first search, as described in the algorithm **Modified BFS**. If a good cross-edge is found, then an augmenting path with respect to $M$ is constructed in time $O(n)$, according to Lemma 3.4.1. If a bad cross-edge is found, then we construct the corresponding blossom $B$ in time $O(n)$, as described in Lemma 3.4.2. Now according to Theorem 3.4.5, there is an augmenting path with respect to $M$ in $G$ if and only if there is an augmenting path with respect to $M - B$ in $G/B$. Thus, we *recursively* look for an augmenting path with respect to $M - B$ in $G/B$. Once an augmenting path $p_B$ is found in $G/B$, we can construct from $p_B$ an augmenting path $p$ with respect to $M$ in $G$ in time $O(n)$, according to Corollary 3.4.6. Finally, suppose that neither good cross-edge nor bad cross-edge is found in the modified breadth first search given $G$ and $M$. Since no bad cross-edges means no blossoms, by Lemma 3.4.4, no good cross-edges implies no augmenting paths with respect to the matching $M$ in $G$.

This gives a complete procedure that constructs an augmenting path when a matching $M$ in a general graph $G$ is given. We summarize this procedure in Figure 3.11.

**Lemma 3.4.7** *The algorithm* **General Augment** *runs in time $O(nm)$ on a graph of $n$ vertices and $m$ edges.*

PROOF. Step 3 is the modified breadth first search process that examines each edge at most twice. Thus, the total running time spent on step 3 is bounded by $O(m)$.

If a good cross-edge is found in the search process, then by Lemma 3.4.1, the corresponding augmenting path can be constructed in time $O(n)$. Thus, in this case, the algorithm returns an augmenting path with respect to $M$ in time $O(m)$.

If a bad cross-edge is found in the search process, then by Lemma 3.4.2, the corresponding blossom $B$ can be constructed in time $O(n)$. Now the graph $G/B$ and the matching $M - B$ in $G/B$ can be constructed in time $O(m)$. A recursive execution is then applied to the graph $G/B$ and the matching $M - B$. Note that the graph $G/B$ has at most $n - 2$ vertices since each blossom consists of at least 3 vertices. Now if an augmenting path $p_B$ is found in $G/B$, then by Corollary 3.4.6, an augmenting path $p$ with respect to $M$ can be constructed from $p_B$ in time $O(n)$. Therefore, if we let $t(n)$ be the running time of the algorithm **General Augment** on a graph of $n$ vertices and $m$ edges, then we have the following recurrence relation:

$$t(n) \le t(n-2) + O(m)$$

which immediately gives $t(n) = O(nm)$. $\square$

**Theorem 3.4.8** *The* GRAPH MATCHING *problem on general graphs can be solved in time $O(n^2m)$.*

PROOF. We apply the algorithm **Maximum Matching** in Figure 3.2 and use the algorithm **General Augment** as a subroutine to find augmenting paths. By Lemma 3.4.7, given a matching $M$ in a graph $G$, and augmenting path can be found in time $O(nm)$. Since an augmenting path results in a larger matching with one more edge, and there are no more than $n/2$ edges in a matching for a graph of $n$ vertices, the **while** loop in step 2 of the algorithm **Maximum Matching** is executed at most $n/2$ times. In conclusion, the algorithm **Maximum Matching** runs in time $(n^2m)$ and constructs a maximum matching for the given graph. $\square$

```
Algorithm.  General Augment
Input: a graph G and a matching M in G
1.  for each vertex v do lev(v) = -1;
2.  for each unmatched vertex v do Q <- v;  lev(v) = 0;
3.  while no augmenting path is found and Q != ∅ do
        u <- Q;
        case 1. lev(u) is even
            for each neighbor w of u do
                subcase 1a. lev(w) = -1 then
                    dad(w) = u; lev(w) = lev(u) + 1;  Q <- w;
                subcase 1b. [u, w] is a good cross-edge
                    construct the augmenting path p; goto step 5;
                subcase 1c. [u, w] is a bad cross-edge
                    construct the blossom B; goto step 6;
        case 2. lev(u) is odd
            let [u, w] ∈ M;
            subcase 2a. lev(w) = -1 then
                dad(w) = u; lev(w) = lev(u) + 1; Q <- w;
            subcase 2b. [u, w] is a good cross-edge
                construct the augmenting path p; goto step 5;
            subcase 2c. [u, w] is a bad cross-edge
                construct the blossom B; goto step 6;
4.  {at this point, neither good nor bad cross-edge is found.}
      return "no augmenting path"; stop.
5.  {at this point, an augmenting path p has been found.}
      return the augmenting path p; stop.
6.  {at this point, a blossom B has been found.}
      construct the graph G/B and the matching M - B;
      recursively apply the algorithm to G/B and M - B
        to look for an augmenting path in G/B;
      if an augmenting path p_B is found in G/B
      then construct an augmenting path p in G from p_B;
          return the augmenting path p
      else return "no augmenting path".
```

Figure 3.11: Finding an augmenting path in a general graph

We should point out that $O(n^2m)$ is not the best upper bound for the GRAPH MATCHING problem. In fact, a moderate change in the algorithm **General Augment** gives an algorithm of running time $O(n^3)$ for the problem [45]. The basic idea for this change is that instead of actually shrinking the blossoms, we keep track of all vertices in a blossom by "marking" them. A careful bookkeeping technique shows that this can be done in time $O(n)$ per blossom. More careful and thorough techniques and analysis have been developed. The best algorithms developed to date for the GRAPH MATCHING problem on general graphs run in time $O(\sqrt{n}m)$ [95], thus matching the best algorithm known for the problem on bipartite graphs.

## 3.5 Weighted matching problems

A natural extension of the GRAPH MATCHING problem is matchings on weighted graphs. Here we measure a matching in term of its weight instead of the number of edges in the matching. Let $G$ be an undirected and weighted graph in which the weight for each edge $e$ is given by $wt(e)$. Let $M$ be a matching in $G$. The *weight* of $M$ is defined by $wt(M) = \sum_{e \in M} wt(e)$. The matching problem on weighted graphs is formally given as follows.

WEIGHTED MATCHING $= \langle I_Q, S_Q, f_Q, opt_Q \rangle$

$I_Q$: the set of all undirected weighted graphs $G$

$S_Q$: $S_Q(G)$ is the set of all matchings in the graph $G$

$f_Q$: $f_Q(G, M)$ is the weight $wt(M)$ of the matching $M$

$opt_Q$: max

The WEIGHTED MATCHING problem on bipartite graphs is commonly called the ASSIGNMENT problem, in which we have a group of people and a set of jobs such that the weight of an edge $[p, j]$ from a person $p$ to a job $j$ represents the benefit of assigning person $p$ to job $j$, and we are looking for an assignment that maximizes the benefit.

Introducing weights in the matching problem makes the problem much harder. In particular, neither must a maximum weighted matching have the maximum number of edges nor must matchings of maximum number of edges be maximum weighted. Therefore, Theorem 3.1.1 does not directly apply. On the other hand, we will see in the rest of this section that the concepts of augmenting paths, modified breadth first search, and blossoms can still be carried over. Due to the space limit, we will only give a brief introduction to the theory and algorithms for the WEIGHTED MATCHING problem. For more thorough and detailed descriptions, readers are referred to more specialized literature [87, 102, 112].

### 3.5.1 Theorems and algorithms

Without loss of generality, we can assume that the weighted graph $G$, which is an instance of the WEIGHTED MATCHING problem, has no edges of weight less than or equal to 0. In fact, any matching $M$ in $G$ minus the edges of weight not larger than 0 gives a matching in $G$ whose weight is at least as large as $wt(M)$. Again we fix a weighted graph $G$, and let $n$ and $m$ be the number of vertices and the number of edges in $G$, respectively.

**Definition 3.5.1** *For each $k \geq 0$ such that there is a matching of $k$ edges in the graph $G$, let $M_k$ be a matching of $k$ edges in $G$ such that $wt(M_k)$ is the maximum over all matchings of $k$ edges in $G$. If there is no matching of $k$ edges in $G$, then we define $M_k = \emptyset$.*

Therefore, for some $k$, the matching $M_k$ is a maximum weighted matching in the graph $G$. We firest characterize the index $k$ such that $M_k$ makes a maximum weighted matching in $G$.

**Lemma 3.5.1** *If the index $k$ satisfies $wt(M_k) \geq wt(M_{k-1})$ and $wt(M_k) \geq wt(M_{k+1})$, then $M_k$ is a maximum weighted matching in the graph $G$.*

PROOF. Assume the opposite, that the matching $M_k$ is not a maximum weighted matching in $G$. Let $M_{\max}$ be a maximum weighted matching in $G$. Consider the graph $G_0 = M_k \oplus M_{\max}$. As we explained in the proof for Theorem 3.1.1, each vertex in the graph $G_0$ has degree at most 2. Thus, each connected component of $G_0$ is either a simple path or a simple cycle, and in each connected component of $G_0$, the number of edges in $M_k$ and the number of edges in $M_{\max}$ differ by at most 1.

Since $wt(M_{\max}) > wt(M_k)$, there must be one connected component $C_0$ in the graph $G_0$ such that

$$\sum_{e \in C_0 \cap M_k} wt(e) < \sum_{e \in C_0 \cap M_{\max}} wt(e)$$

It is easy to verify that $M' = M_k \oplus C_0$ is also a matching in $G$. Moreover, we have $wt(M') > wt(M_k)$. Now a contradiction is derived: (1) if $C_0$ contains the same number of edges in $M_k$ and in $M_{\max}$, then the matching $M'$ has exactly $k$ edges, contradicting the assumption that $M_k$ has the maximum weight over all matchings of $k$ edges in $G$; (2) if $C_0$ contains one more edge in $M_k$ than in $M_{\max}$, then $M'$ is a matching of $k-1$ edges, contradicting the assumption that $wt(M_k) \geq wt(M_{k-1})$; and (3) if $C_0$ contains one more edge in $M_{\max}$ than in $M_k$, then $M'$ is a matching of $k+1$ edges, contradicting the assumption that $wt(M_k) \geq wt(M_{k+1})$. $\square$

Therefore, the problem remaining is to find the matchings $M_k$, for $k \geq 0$. The following theorem indicates how the matching $M_{k+1}$ can be constructed from the matching $M_k$. For an augmenting path $p$ with respect to the matching $M_k$, we define the *differential weight* of $p$, denoted $dw(p)$, to be the difference $\sum_{e \in p - M_k} wt(e) - \sum_{e \in p \cap M_k} wt(e)$.

**Theorem 3.5.2** *Let $p$ be an augmenting path with respect to $M_k$ such that $dw(p)$ is the maximum over all augmenting paths with respect to $M_k$. Then $M' = M_k \oplus p$ is a matching of $k+1$ edges in $G$ such that $wt(M')$ is the maximum over all matchings of $k+1$ edges in $G$.*

PROOF. Since there is at least one augmenting path with respect to $M_k$, the matching $M_{k+1}$ is not empty. Moreover, note that $M_k \oplus p$ is a matching of $k+1$ edges in $G$ whose weight is $wt(M_k) + dw(p)$. Thus, $wt(M_k) + dw(p) \leq wt(M_{k+1})$.

Consider the graph $G_0 = M_k \oplus M_{k+1}$, in which each connected component is either a simple path or a simple cycle. Moreover, in each connected component in $G_0$, the number of edges in $M_k$ and the number of edges in $M_{k+1}$ differ by at most 1.

We say that a connected component $C$ of the graph $G_0$ is "balanced" if the number of edges in $C \cap M_k$ is equal to that in $C \cap M_{k+1}$, is "$M_k$-dominating" if the number of edges in $C \cap M_k$ is 1 more than that in $C \cap M_{k+1}$, and is "$M_{k+1}$-dominating" if the number of edges in $C \cap M_k$ is 1 less than that in $C \cap M_{k+1}$. Since the number of edges in the matching $M_{k+1}$ is 1 more that the number of edges in the matching $M_k$, the number of $M_{k+1}$-dominating components is exactly one larger than the number of $M_k$-dominating components in $G_0$. We pair arbitrarily each $M_k$-dominating component with a $M_{k+1}$-dominating component, leaving one $M_{k+1}$-dominating component unpaired.

For each balanced component $C_0$ in $G_0$, we must have $dw(C_0) = 0$. In fact, if $dw(C_0) < 0$, then $M_{k+1} \oplus C_0$ is a matching of $k+1$ edges that has a weight larger than $wt(M_{k+1})$, contradicting the definition of the matching $M_{k+1}$, and if $dw(C_0) > 0$, then $M_k \oplus C_0$ is a matching of $k$ edges that has weight larger than $wt(M_k)$, contradicting the definition of the matching $M_k$.

**Algorithm. Weighted Matching**

Input: an undirected weighted graph $G = (V, E)$
Output: a maximum weighted matching $M$ in $G$

1. $M_0 = \emptyset$; $k = 0$;
2. **while** there is an augumenting path w.r.t $M_k$ **do**
   find an augmenting path $p$ of maximum differential weight;
   let $M_{k+1} = M_k \oplus p$;
   **if** $wt(M_k) \geq wt(M_{k+1})$
   **then** stop: $M_k$ is the maximum weighted matching
   **else** $k = k + 1$;

Figure 3.12: An algorithm constructing a maximum weighted matching

For each $M_k$-dominating component $C_k$ that is paired with an $M_{k+1}$-dominating component $C_{k+1}$, we first note that the number of edges in $M_k \cap (C_k \cup C_{k+1})$ is equal to the number of edges in $M_{k+1} \cap (C_k \cup C_{k+1})$. Now if $dw(C_k) + dw(C_{k+1}) < 0$, then $M_{k+1} \oplus C_k \oplus C_{k+1}$ is a matching of $k+1$ edges that has weight larger than $wt(M_{k+1})$, contradicting the definition of the matching $M_{k+1}$, and if $dw(C_k) + dw(C_{k+1}) > 0$, then $M_k \oplus C_k \oplus C_{k+1}$ is a matching of $k$ edges that has weight larger than $wt(M_k)$, contradicting the definition of the matching $M_k$. Thus, we must have $dw(C_k) + dw(C_{k+1}) = 0$.

Therefore, if we let $C_s$ be the only unpaired $M_{k+1}$-dominating component in $G_0$, we must have $\sum_{C \in G_0} dw(C) = dw(C_s)$. By the definition, $\sum_{C \in G_0} dw(C) = wt(M_{k+1}) - wt(M_k)$. Since $C_s$ is an augmenting path with respect to $M_k$, by our choice of the augmenting path $p$, we have $dw(p) \geq dw(C_s)$. Therefore, $wt(M_k) + dw(p) \geq wt(M_{k+1})$. Combining this with the result we obtained in the beginning of this proof, we conclude that $wt(M_k) + dw(p) = wt(M_{k+1})$ and that $M_k \oplus p$ is a matching of $k + 1$ edges such that $wt(M_k \oplus p)$ is the maximum over all matchings of $k + 1$ edges in $G$. $\square$

Lemma 3.5.1 and Theorem 3.5.2 suggest the algorithm described in Figure 3.12 for solving the WEIGHTED MATCHING problem. Theorem 3.5.2 guarantees that the constructed matching $M_k$ of $k$ edges for each $k$ has the maximum weight over all matchings of $k$ edges in $G$, and Lemma 3.5.1 ensures that when the algorithm stops, the matching $M_k$ is the maximum weighted matching in the graph $G$.

According to the algorithm **Weighted Matching**, the problem WEIGHTED MATCHING is reduced to the problem of finding an augmenting path of maximum differential weight. If the graph $G$ is bipartite, then finding an augmenting path of maximum differential weight can be reduced to solving the SHORTEST PATH problem, which was given in Section 1.1 and can be solved using the well-known Dijkstra's algorithm [28][1] However, finding an augmenting path of maximum differential weight in a general graph is, though still possible [34], much harder because of, not surprisingly, the existence of the blossom structure.

A different approach has been adopted in the literature. The approach is based on the duality theory of linear programming, more specifically on the primal-dual method. Due to the space limit here, we are unable to give the details of this approach. Readers are referred to the excellent discussion in [87, 102, 112]. Instead, we state the best solution to this problem as follows.

**Theorem 3.5.3** *The* WEIGHTED MATCHING *problem on general weighted graphs can be solved in time $O(n^3)$.*

Theorem 3.5.3 is the best known bound for the WEIGHTED MATCHING problem on general graphs. With more efficient data structures and more careful manipulation on blossoms, an algorithm of time complexity

$$O(nm \log\log\log_{m/n+1} n + n^2 \log n)$$

has been developed [46], which is much faster than the $O(n^3)$ time algorithm when applied to sparse graphs with only $O(n)$ edges.

### 3.5.2 Minimum perfect matchings

The WEIGHTED MATCHING problem has a number of interesting variations that have nice applications in computational optimization. In this subsection, we discuss one of these variations, which will be used in our later discussion.

Let $M$ be a matching in a graph $G$. The matching $M$ is *perfect* if every vertex in $G$ is matched. In other words, the matching $M$ contains exactly $n/2$ edges in $G$ if $G$ has $n$ vertices. The MIN PERFECT MATCHING problem is to look for a *minimum* weighted perfect matching in a given weighted graph, formally defined as follows.

[1]Dijkstra's algorithm solving the SHORTEST PATH problem is very similar to Prim's greedy algorithm **PRIM** presented in Section 1.3.1 for solving the MINIMUM SPANNING TREE problem. Readers are encouraged to work on this algorithm without looking at the reference.

MIN PERFECT MATCHING $= \langle I_Q, S_Q, f_Q, opt_Q \rangle$

$I_Q$: the set of all undirected weighted graphs $G$

$S_Q$: $S_Q(G)$ is the set of all perfect matchings in the graph $G$

$f_Q$: $f_Q(G, M)$ is the weight $wt(M)$ of the matching $M$

$opt_Q$: min

Of course, not all graphs have perfect matchings. In particular, a graph of odd number of vertices has no perfect matchings. Therefore, the MIN PERFECT MATCHING problem is only defined on weighted graphs that have perfect matchings. Since whether a given graph $G$ has perfect matchings can be detected in time $O(\sqrt{n}m)$ by applying the algorithm **Maximum Matching** in Figure 3.2 to the graph $G$, regarded as an unweighted graph, we always assume in the following discussion that the given graph $G$ has perfect matchings.

We show how the MIN PERFECT MATCHING problem is reduced to the WEIGHTED MATCHING problem. Let $G$ be an instance of the MIN PERFECT MATCHING problem. For each edge $e$ in $G$, let $wt(e)$ be the weight of $e$ in $G$. We construct a new weighted graph $G'$ as follows. Let $w_0$ be the maximum $|wt(e)|$ over all edges $e$ of $G$. The graph $G'$ has the same vertex set and edge set as $G$. For each edge $e$ in $G'$, the weight $wt'(e)$ of $e$ in $G'$ is equal to $wt'(e) = (m + n)w_0 - wt(e)$.

**Lemma 3.5.4** *Let $M'$ be a maximum weighted matching in the graph $G'$. Then the same set of edges in $M'$ constitutes a minimum perfect matching in the graph $G$.*

PROOF. By our assumption, the graph $G$ has perfect matchings. Thus, the number $n$ of vertices of $G$ is an even number.

First we show that any imperfect matching in $G'$ has weight strictly less than that of any perfect matching in $G'$. Let $M_i$ be an imperfect matching in $G'$ and let $M_p$ be a perfect matching $G'$. We have

$$\begin{aligned} wt'(M_i) &= \sum_{e \in M_i} wt'(e) = \sum_{e \in M_i} ((m+n)w_0 - wt(e)) \\ &\leq \sum_{e \in M_i} (m+n+1)w_0 \\ &\leq (\frac{n}{2} - 1)(m+n+1)w_0 \end{aligned}$$

where the first inequality is because of $w_0 \geq |wt(e)|$ for any edge $e$, and the last inequality is because $M_i$ is imperfect thus contains at most $\frac{n}{2} - 1$ edges. Also

$$\begin{aligned} wt'(M_p) &= \sum_{e \in M_p} wt'(e) = \sum_{e \in M_p} ((m+n)w_0 - wt(e)) \\ &= \sum_{e \in M_p} ((m+n-1)w_0 + (w_0 - wt(e))) \\ &\geq \sum_{e \in M_p} (m+n-1)w_0 \\ &= \frac{n}{2}(m+n-1)w_0 \end{aligned}$$

where the first inequality is because $w_0 - wt(e) \geq 0$ for all edges $e$. Since $\frac{n}{2}(m+n-1) > (\frac{n}{2}-1)(m+n+1)$, we derive that the perfect matching $M_p$ has weight strictly larger than the weight of the imperfect matching $M_i$. In consequence, every maximum weighted matching in the graph $G'$ must be a perfect matching.

Now for any perfect matching $M_p$ in the graph $G'$ we have

$$\begin{aligned} wt'(M_p) &= \sum_{e \in M_p} wt'(e) = \sum_{e \in M_p} ((m+n)w_0 - wt(e)) \\ &= \frac{n}{2}(m+n)w_0 - \sum_{e \in M_p} wt(e) \\ &= \frac{n}{2}(m+n)w_0 - wt(M_p) \end{aligned}$$

Thus, for any two perfect matchings $M_p$ and $M_p'$ in $G'$, $wt'(M_p) \geq wt'(M_p')$ if and only if $wt(M_p) \leq wt(M_p')$. In conclusion, the maximum weighted matching $M'$ in the graph $G'$ must constitute a minimum weighted perfect matching in the graph $G$. $\square$

Combining Theorem 3.5.3 and Lemma 3.5.4, we obtain

**Theorem 3.5.5** *The* Min Perfect Matching *problem can be solved in time $O(n^3)$.*

# Chapter 4

# Linear Programming

Recall that a general instance of the Linear Programming problem is described as follows.

Linear Programming

$$
\begin{array}{rc}
\text{minimize} & c_1x_1 + \cdots + c_nx_n \\
\text{subject to} & \\
& a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \geq a_1 \\
& \cdots\cdots \\
& a_{r1}x_1 + a_{r2}x_2 + \ldots + a_{rn}x_n \geq a_r \\
& b_{11}x_1 + b_{12}x_2 + \ldots + b_{1n}x_n \leq b_1 \\
& \cdots\cdots \\
& b_{s1}x_1 + b_{s2}x_2 + \ldots + b_{sn}x_n \leq b_s \\
& d_{11}x_1 + d_{12}x_2 + \ldots + d_{1n}x_n = d_1 \\
& \cdots\cdots \\
& d_{t1}x_1 + d_{t2}x_2 + \ldots + d_{tn}x_n = d_t
\end{array}
\tag{4.1}
$$

where $c_i$, $a_{ji}$, $a_j$, $b_{ki}$, $b_k$, $d_{li}$, and $d_l$ are all given real numbers, for $1 \leq i \leq n$, $1 \leq j \leq r$, $1 \leq k \leq s$, and $1 \leq l \leq t$; and $x_i$, $1 \leq i \leq n$, are unknown variables.

The Linear Programming problem is characterized, as the name implies, by linear functions of the unknown variables: the objective function is linear in the unknown variables, and the constraints are linear equalities or linear inequalities in the unknown variables.

For many combinatorial optimization problems, the objective function and the constraints on a solution to an input instance are linear, i.e., they can be formulated by linear equalities and linear inequalities. Therefore, optimal solutions for these combinatorial optimization problems can be derived from optimal solutions for the corresponding instance in the LINEAR PROGRAMMING problem. This is one of the main reasons why the LINEAR PROGRAMMING problem receives so much attention.

For example, consider the MAXIMUM FLOW problem. Let $G$ be an instance of the MAXIMUM FLOW problem. Thus, $G$ is a flow network. Without loss of generality, we can assume that the vertices of $G$ are named by the integers 1, 2, ..., $n$, where 1 is the source and $n$ is the sink. Each pair of vertices $i$ and $j$ in $G$ is associated with an integer $c_{ij}$, which is the capacity of the edge $[i, j]$ in $G$ (if there is no edge from $i$ to $j$, then $c_{ij} = 0$). To formulate the instance $G$ of the MAXIMUM FLOW problem into an instance of the LINEAR PROGRAMMING problem, we introduce $n^2$ unknown variables $f_{ij}$, $1 \leq i, j \leq n$, where the variable $f_{ij}$ is for the amount of flow from vertex $i$ to vertex $j$. By the definition of flow in a flow network, the flow value $f_{ij}$ must satisfy the capacity constraint, the skew symmetry constraint, and the flow conservation constraint. These constraints can be easily formulated into linear relations:

$$\begin{array}{rll} \text{capacity constraint:} & f_{ij} \leq c_{ij} & \text{for all } 1 \leq i, j \leq n \\ \text{skew symmetry:} & f_{ij} = -f_{ji} & \text{for all } 1 \leq i, j \leq n \\ \text{flow conservation:} & \sum_{j=1}^{n} f_{ij} = 0 & \text{for } i \neq 1, n \end{array}$$

Finally, the MAXIMUM FLOW problem is to maximize the flow value, which by definition is given by $f_{11} + f_{12} + \cdots + f_{1n}$. This is equivalent to minimizing the value $-f_{11} - f_{12} - \cdots - f_{1n}$. Therefore, the instance $G$ of the MAXIMUM FLOW problem has been formulated into an instance of the LINEAR PROGRAMMING problem as follows.

$$\begin{array}{rll} \text{minimize} & -f_{11} - f_{12} - \cdots - f_{1n} & \\ \text{subject to} & & \\ & f_{i,j} \leq c_{ij} & \text{for all } 1 \leq i, j \leq n \\ & f_{ij} + f_{ji} = 0 & \text{for all } 1 \leq i, j \leq n \\ & \sum_{j=1}^{n} f_{ij} = 0 & \text{for } i \neq 1, n \end{array}$$

An efficient algorithm for the LINEAR PROGRAMMING problem implies an efficient algorithm for the MAXIMUM FLOW problem.

In this chapter, we introduce the basic concepts and efficient algorithms for the LINEAR PROGRAMMING problem. We start by introducing the basic concepts and preliminaries for the LINEAR PROGRAMMING problem. An algorithm, the "simplex method", is then described. The simplex method is, though not a polynomial time bounded algorithm, very fast for most practical instances of the LINEAR PROGRAMMING problem. We will also discuss the idea of the *dual* LINEAR PROGRAMMING problem, which can be used to solve the original LINEAR PROGRAMMING problem more efficiently than by simply applying the simplex method to the original problem. Finally, polynomial time algorithms for the LINEAR PROGRAMMING problem will be briefly introduced.

We assume in this chapter the familiarity of the fundamentals of linear algebra. In particular, we assume that the readers are familiar with the definitions of vectors, matrices, linear dependency and linear independency, and know how a system of linear equations can be solved. All these can be found in any introductory book in linear algebra. Appendix C provides a quick review for these concepts. To avoid confusions, we will use little bold letters such as $\mathbf{x}$ and $\mathbf{c}$ for vectors, and use capital bold letters such as $\mathbf{A}$ and $\mathbf{B}$ for matricies. For a vector $\mathbf{x}$ and a real number c, we write $\mathbf{x} \geq c$ if all elements in $\mathbf{x}$ are larger than or equal to $c$.

## 4.1 Basic concepts

First note that in the constraints in a general instance in (4.1) of the LINEAR PROGRAMMING problem, there is no strict inequalities. Mathematically, any bounded set defined by linear equalities and non-strict linear inequalities is a "compact set" in the Euclidean space, in which the objective function can always achieve its optimal value, while strict linear inequalities define a non-compact set in which the objective function may not be able to achieve its optimal value. For example, consider the following instance:

$$\begin{array}{ll} \text{minimize} & -x_1 - x_2 \\ \text{subject to} & \\ & x_1 + x_2 + x_3 < 1 \\ & x_1 \geq 0, \;\; x_2 \geq 0, \;\; x_3 \geq 0 \end{array}$$

The set $S$ defined by the constraints $x_1 + x_2 + x_3 < 1$, $x_1 \geq 0$, $x_2 \geq 0$, and $x_3 \geq 0$ is certainly bounded. However, no vector $(x_1, x_2, x_3)$ in $S$ can make the objective function $-x_1 - x_2$ to achieve the minimum value: for any

$\epsilon > 0$, we can find a vector $(x_1, x_2, x_3)$ in the set $S$ that makes the objective function $-x_1 - x_2$ to have value less than $-1 + \epsilon$ but no vector in the set $S$ can make the objective function $-x_1 - x_2$ to have value less than or equal to $-1$.

Now we show how a general instance in (4.1) of the LINEAR PROGRAMMING problem can be converted into a simpler form.

The *standard form* for the LINEAR PROGRAMMING problem is given in the following format

$$\begin{array}{ll} \text{minimize} & c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ \text{subject to} & \\ & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ & a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ & \cdots\cdots \\ & a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m \\ & x_1 \geq 0, \quad x_2 \geq 0, \quad \ldots, \quad x_n \geq 0 \end{array} \tag{4.2}$$

The general form in (4.1) of the LINEAR PROGRAMMING problem can be converted into the standard form in (4.2) through the following steps.

1. **Eliminating $\geq$ inequalities**

   Each inequality $a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n \geq a_i$ is replaced by the equivalent inequality $(-a_{i1})x_1 + (-a_{i2})x_2 + \ldots + (-a_{in})x_n \leq (-a_i)$.

2. **Eliminating $\leq$ inequalities**

   Each inequality $b_{j1}x_1 + b_{j2}x_2 + \ldots + b_{jn}x_n \leq b_j$ is replaced by the equality $b_{j1}x_1 + b_{j2}x_2 + \ldots + b_{jn}x_n + y_j = b_j$ by introducing a new *slack variable* $y_j$ with the constraint $y_j \geq 0$.

3. **Eliminating unconstrained variables**

   For each variable $x_i$ such that the constraint $x_i \geq 0$ is not present, introduce two new variables $u_i$ and $v_i$ satisfying $u_i \geq 0$ and $v_i \geq 0$, and replace the variable $x_i$ by $u_i - v_i$.

The above transformation rules are not strict. For example, the $\geq$ inequalities can also be eliminated using a "surplus variable". Moreover, sometimes a simple linear transformation may be more convenient and more effective than the above transformations. We illustrate these transformations and other possible transformations by an example. Consider the instance in (4.3) for the LINEAR PROGRAMMING problem.

$$\begin{aligned} \text{minimize} \quad & 2x_1 + x_2 - 3x_3 \\ \text{subject to} \quad & \\ & 2x_1 - x_2 - 7x_3 \geq 5 \\ & 2x_2 - x_3 = 3 \\ & x_2 \geq 2 \end{aligned} \tag{4.3}$$

We apply the first rule to convert the first constraint $2x_1 - x_2 - 7x_3 \geq 5$ into $-2x_1 + x_2 + 7x_3 \leq -5$. Then we apply the second rule and introduce a new slack variable $x_4$ with constraint $x_4 \geq 0$ to get an equality $-2x_1 + x_2 + 7x_3 + x_4 = -5$.

For the constraint $x_2 \geq 2$, we could also convert it into an equality using the first and second rules. However, we can also perform a simple linear transformation as follows. Let $x_2' = x_2 - 2$ and replace in (4.3) the variable $x_2$ by $x_2' + 2$. This combined with the transformations on the first constraint will convert the instance (4.3) into the form

$$\begin{aligned} \text{minimize} \quad & 2x_1 + x_2' - 3x_3 \\ \text{subject to} \quad & \\ & -2x_1 + x_2' + 7x_3 + x_4 = -7 \\ & 2x_2' - x_3 = -1 \\ & x_2' \geq 0, \quad x_4 \geq 0 \end{aligned} \tag{4.4}$$

Note that after the linear transformation $x_2 = x_2' + 2$, the objective function $2x_1 + x_2 - 3x_3$ should have become $2x_1 + x_2' - 3x_3 + 2$. However, minimizing $2x_1 + x_2' - 3x_3 + 2$ is equivalent to minimizing $2x_1 + x_2' - 3x_3$.

Now we need to remove the unconstrained variables in the instance (4.4). For the unconstrained variable $x_1$, by the third rule, we introduce two new variables $x_1'$ and $x_1''$ with constraints $x_1' \geq 0$ and $x_1'' \geq 0$, and replace in (4.4) $x_1$ by $x_1' - x_1''$. We obtain

$$\begin{aligned} \text{minimize} \quad & 2x_1' - 2x_1'' + x_2' - 3x_3 \\ \text{subject to} \quad & \\ & -2x_1' + 2x_1'' + x_2' + 7x_3 + x_4 = -7 \\ & 2x_2' - x_3 = -1 \\ & x_1' \geq 0, \quad x_1'' \geq 0 \ \ x_2' \geq 0, \quad x_4 \geq 0 \end{aligned} \tag{4.5}$$

The unconstrained variable $x_3$ could also be eliminated using the same rule. But it can also be eliminated using a simple linear transformation. For this, we observe the constraint $2x_2' - x_3 = -1$ so $x_3 = 2x_2' + 1$. Thus, replacing $x_3$ in (4.5) by $2x_2' + 1$, we obtain the following standard form for the LINEAR PROGRAMMING problem.

$$\begin{aligned} &\text{minimize} && 2x_1' - 2x_1'' - 5x_2' \\ &\text{subject to} && \\ & && -2x_1' + 2x_1'' + 15x_2' + x_4 = -14 \\ & && x_1' \geq 0, \;\; x_1'' \geq 0 \;\; x_2' \geq 0 \;\; x_4 \geq 0 \end{aligned} \tag{4.6}$$

It is easy to verify that if we solve the instance (4.6) and obtain an optimal solution $(x_1', x_1'', x_2', x_4)$, then we can construct an optimal solution $(x_1, x_2, x_3)$ for the instance (4.3), where $x_1 = x_1' - x_1''$, $x_2 = x_2' + 2$, and $x_3 = 2x_2' + 1$.

Note that the transformations do not result in an instance whose size is much larger than the original instance. In fact, to eliminate an inequality, we need to introduce at most one new variable $y$ plus a new constraint $y \geq 0$, and to eliminate an unconstrained variable we need to introduce at most two new variables $u$ and $v$ plus two new constraints $u \geq 0$ and $v \geq 0$. Therefore, if the original instance consists of $n$ variables and $m$ constraints, then the corresponding instance in the standard form consists of at most $2n + m$ variables and $2n + 2m$ constraints.

Therefore, without loss of generality, we can always assume that a given instance of the LINEAR PROGRAMMING problem is in the standard form. Using our 4-tuple formulation, the LINEAR PROGRAMMING problem can now be formulated as follows.

LINEAR PROGRAMMING $= \langle I_Q, S_Q, f_Q, opt_Q \rangle$, where

- $I_Q$ is the set of triples $(\mathbf{b}, \mathbf{c}, \mathbf{A})$, where $\mathbf{b}$ is an $m$-dimensional vector of real numbers, $\mathbf{c}$ is an $n$-dimensional vector of real numbers, and $\mathbf{A}$ is an $m \times n$ matrix of real numbers, for some integers $n$ and $m$;
- for an instance $\alpha = (\mathbf{b}, \mathbf{c}, \mathbf{A})$ in $I_Q$, the solution set $S_Q(\alpha)$ consists of the set of $n$-dimensional vectors $\mathbf{x}$ that satisfy the constraints $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{x} \geq 0$;
- for a given input instance $\alpha \in I_Q$ and a solution $\mathbf{x} \in S_Q(\alpha)$, the objective function value is defined to be the inner product $\mathbf{c}^T\mathbf{x}$ of the vectors $\mathbf{c}$ and $\mathbf{x}$;

- $opt_Q$ is min.

We make a further assumption that the $m \times n$ matrix $\mathbf{A}$ in an instance of the LINEAR PROGRAMMING problem has its $m$ rows linearly independent, which also implies that $m \leq n$. This assumption can be justified as follows. If the $m$ rows of the matrix $\mathbf{A}$ are not linearly independent, then either the constraint $\mathbf{Ax} = \mathbf{b}$ is contradictory, in which case the instance obviously has no solution, or there are redundancy in the constraint. The redundancy in the constraint can be eliminated using standard linear algebra techniques such as the well-known Gaussian Elimination algorithm.

Under these assumptions, we can assume that there are $m$ columns in the matrix $\mathbf{A}$ that are linearly independent. Without loss of generality, suppose that the first $m$ columns of $\mathbf{A}$ are linearly independent and let $\mathbf{B}$ be the nonsingular $m \times m$ submatrix of $\mathbf{A}$ such that $\mathbf{B}$ consists of the first $m$ columns of $\mathbf{A}$. Let $\mathbf{x_B} = (x_1, x_2, \ldots, x_m)^T$ be the $m$-dimensional vector that consists of the first $m$ unknown variables in the vector $\mathbf{x}$. Since the matrix $\mathbf{B}$ is nonsingular, the equation

$$\mathbf{Bx_B} = \mathbf{b}$$

has a unique solution $\mathbf{x}^0_\mathbf{B} = \mathbf{B}^{-1}\mathbf{b}$. If we let $\mathbf{x}^0 = (\mathbf{x}^0_\mathbf{B}, \overbrace{0, \ldots, 0}^{n-m})$, then obviously, $\mathbf{x}^0$ is a solution to the system $\mathbf{Ax} = \mathbf{b}$. If the vector $\mathbf{x}^0$ happens to also satisfy the constraint $\mathbf{x}^0 \geq 0$, then $\mathbf{x}^0$ is a solution to the instance of the LINEAR PROGRAMMING problem

$$\begin{array}{cc} \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \\ & \mathbf{Ax} = \mathbf{b} \quad \text{and} \quad \mathbf{x} \geq 0 \end{array} \tag{4.7}$$

This introduces a very important class of solutions to an instance of the LINEAR PROGRAMMING problem, formally defined as follows.

**Definition 4.1.1** A vector $\mathbf{x}^0 = (x^0_1, x^0_2, \ldots, x^0_n)^T$ satisfying $\mathbf{Ax}^0 = \mathbf{b}$ and $\mathbf{x}^0 \geq 0$ is a *basic solution* if there are $m$ indices $1 \leq i_1 < i_2 < \ldots < i_m \leq n$ such that the $i_1$th, $i_2$th, ..., $i_m$th columns of the matrix $\mathbf{A}$ are linearly independent, and $x^0_i = 0$ for all $i \notin \{i_1, \ldots, i_m\}$. These $m$ columns of the matrix $\mathbf{A}$ will be called the *basic columns* for $\mathbf{x}^0$.

Note that we did not exclude the possibility that $x^0_{i_j} = 0$ for some index $i_j$ in the basic solution $\mathbf{x}^0$. If any element $x^0_{i_j} = 0$ in the above basic solution $\mathbf{x}^0$, the basic solution $\mathbf{x}^0$ is called a *degenerate basic solution*.

The following theorem is fundamental in the study of the LINEAR PROGRAMMING problem.

**Theorem 4.1.1** *Let* $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ *be an instance for the* LINEAR PROGRAMMING *problem. If the solution set* $S_Q(\alpha)$ *is not empty, then* $S_Q(\alpha)$ *contains a basic solution. Moreover, if the objective function* $\mathbf{c}^T\mathbf{x}$ *achieves the minimum value at a vector* $\mathbf{x}^0$ *in* $S_Q(\alpha)$*, then there is a basic solution* $\mathbf{x}_b^0$ *in* $S_Q(\alpha)$ *such that* $\mathbf{c}^T\mathbf{x}_b^0 = \mathbf{c}^T\mathbf{x}^0$.

PROOF. Suppose that $S_Q(\alpha) \neq \emptyset$. Let $\mathbf{x}_b = (x_1, x_2, \ldots, x_n)^T$ be a solution to the instance $\alpha$ such that $\mathbf{x}_b$ has the maximum number of 0 elements over all solutions in $S_Q(\alpha)$. We show that $\mathbf{x}_b$ must be a basic solution.

For convenience, we suppose that the first $p$ elements $x_1$, $x_2$, …, $x_p$ in $\mathbf{x}_b$ are larger than 0 and all other elements in $\mathbf{x}_b$ are 0. Let the $n$ column vectors of the matrix $\mathbf{A}$ be $\mathbf{a}_1$, $\mathbf{a}_2$, …, $\mathbf{a}_n$. Then the equality $\mathbf{A}\mathbf{x}_b = \mathbf{b}$ can be written as $x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_n\mathbf{a}_n = \mathbf{b}$. Since $x_{p+1} = \cdots x_n = 0$, this equality is equivalent to

$$x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_p\mathbf{a}_p = \mathbf{b} \tag{4.8}$$

If $\mathbf{x}_b$ is not a basic solution, then the column vectors $\mathbf{a}_1$, …, $\mathbf{a}_p$ are linearly dependent. Thus, there are $p$ real numbers $y_1$, …, $y_p$ such that at least one $y_i$ is positive and that

$$y_1\mathbf{a}_1 + y_2\mathbf{a}_2 + \cdots + y_p\mathbf{a}_p = \mathbf{0} \tag{4.9}$$

where $\mathbf{0}$ denotes the $m$-dimensional vector $(0, 0, \ldots, 0)^T$. Let $\epsilon$ be a constant. Subtract $\epsilon$ times the equality (4.9) from the equality (4.8), we get

$$(x_1 - \epsilon y_1)\mathbf{a}_1 + (x_2 - \epsilon y_2)\mathbf{a}_2 + \cdots + (x_p - \epsilon y_p)\mathbf{a}_p = \mathbf{b} \tag{4.10}$$

Equality (4.10) holds for any constant $\epsilon$. Since at least one $y_i$ is positive, the value $\epsilon_0 = \min\{x_i/y_i \mid y_i > 0\}$ is well-defined and $\epsilon_0 > 0$ (note that $x_i > 0$ for all $1 \leq i \leq p$). Again for convenience, suppose that $y_p > 0$ and $\epsilon_0 = x_p/y_p$. With this choice of $\epsilon_0$, we have $x_i - \epsilon_0 y_i \geq 0$ for all $1 \leq i \leq p$. Thus, in equality (4.10), if we let $z_i = x_i - \epsilon_0 y_i$ for all $1 \leq i \leq p$, we will get

$$z_1\mathbf{a}_1 + z_2\mathbf{a}_2 + \cdots + z_{p-1}\mathbf{a}_{p-1} = \mathbf{b}$$

and

$$z_1 \geq 0, \quad z_2 \geq 0, \quad \ldots, \quad z_{p-1} \geq 0$$

Now if we let $\mathbf{z} = (z_1, z_2, \ldots, z_{p-1}, 0, 0, \ldots, 0)^T$ be the $n$-dimensional vector with the last $n - p + 1$ elements all equal to 0, we will get

$$\mathbf{Az} = \mathbf{b} \quad \text{and} \quad \mathbf{z} \geq 0$$

Thus, $\mathbf{z}$ is a solution to the instance $\alpha$ and $\mathbf{z}$ has at least $n-p+1$ elements equal to 0. However, this contradicts our assumption that the vector $\mathbf{x}_b$ is a solution to the instance $\alpha$ with the maximum number of 0 elements over all solutions to $\alpha$. This contradiction shows that the vector $\mathbf{x}_b$ must be a basic solution to $\alpha$.

This proves that if $S_Q(\alpha)$ is not empty, then $S_Q(\alpha)$ contains at least one basic solution.

Now suppose that there is a solution $\mathbf{x}^0$ in $S_Q(\alpha)$ such that $\mathbf{c}^T\mathbf{x}^0$ is the minimum over all solutions in $S_Q(\alpha)$. We pick from $S_Q(\alpha)$ a solution $\mathbf{x}_b^0 = (x_1^0, \ldots, x_n^0)$ such that $\mathbf{c}^T\mathbf{x}_b^0 = \mathbf{c}^T\mathbf{x}^0$ and $\mathbf{x}_b^0$ has the maximum number of 0 elements over all solutions $\mathbf{x}$ in $S_Q(\alpha)$ satisfying $\mathbf{c}^T\mathbf{x} = \mathbf{c}^T\mathbf{x}^0$. We show that $\mathbf{x}_b^0$ is a basic solution.

As we proceeded before, we assume that the first $p$ elements in $\mathbf{x}_b^0$ are positive and all other elements in $\mathbf{x}_b^0$ are 0. If $\mathbf{x}_b^0$ is not a basic solution, then we can find $p$ real numbers $y_1, \ldots, y_p$ in which at least one $y_i$ is positive such that

$$(x_1^0 - \epsilon y_1)\mathbf{a}_1 + (x_2^0 - \epsilon y_2)\mathbf{a}_2 + \cdots + (x_p^0 - \epsilon y_p)\mathbf{a}_p = \mathbf{b}$$

for any constant $\epsilon$. Now if we let $\mathbf{y} = (y_1, y_2, \ldots, y_p, 0, \ldots, 0)^T$ be the $n$-dimensional vector with the last $n-p$ elements equal to 0, then $\mathbf{A}(\mathbf{x}_b^0 - \epsilon\mathbf{y})$ $= \mathbf{b}$ for any $\epsilon$. Since $x_i > 0$ for $1 \leq i \leq p$ and $x_j = y_j = 0$ for $j > p$, we have $\mathbf{x}_b^0 - \epsilon\mathbf{y} \geq 0$ for small enough (positive or negative) $\epsilon$. Thus, for any small enough $\epsilon$, $\mathbf{z}_\epsilon = \mathbf{x}_b^0 - \epsilon\mathbf{y} \geq 0$ is a solution to the instance $\alpha$. Now consider the objective function value $\mathbf{c}^T\mathbf{z}_\epsilon$. We have

$$\mathbf{c}^T\mathbf{z}_\epsilon = \mathbf{c}^T\mathbf{x}_b^0 - \epsilon\mathbf{c}^T\mathbf{y}$$

We claim that we must have $\mathbf{c}^T\mathbf{y} = \mathbf{0}$. In fact, if $\mathbf{c}^T\mathbf{y} \neq \mathbf{0}$, then pick a proper small $\epsilon$, we will have $\epsilon\mathbf{c}^T\mathbf{y} > 0$. But this implies that the value $\mathbf{c}^T\mathbf{z}_\epsilon =$ $\mathbf{c}^T\mathbf{x}_b^0 - \epsilon\mathbf{c}^T\mathbf{y}$ is smaller than $\mathbf{c}^T\mathbf{x}_b^0$, and $\mathbf{z}_\epsilon$ is a solution in $S_Q(\alpha)$, contradicting our assumption that $\mathbf{x}_b^0$ minimizes the value $\mathbf{c}^T\mathbf{x}$ over all solutions $\mathbf{x}$ in $S_Q(\alpha)$.

Thus, we must have $\mathbf{c}^T\mathbf{y} = \mathbf{0}$. In consequence, $\mathbf{c}^T\mathbf{z}_\epsilon = \mathbf{c}^T\mathbf{x}_b^0$ for any $\epsilon$. Now if we let $\epsilon_0 = \min\{x_i/y_i \mid y_i > 0\}$, and let $\mathbf{z}_0 = \mathbf{x}_b^0 - \epsilon_0\mathbf{y}$, then we have $\mathbf{c}^T\mathbf{z}_0 = \mathbf{c}^T\mathbf{x}_b^0 = \mathbf{c}^T\mathbf{x}^0$, $\mathbf{Az}_0 = \mathbf{b}$, $\mathbf{z}_0 \geq 0$, and $\mathbf{z}_0$ has at least $n-p+1$

elements equal to 0. However, this contradicts our assumption that $\mathbf{x}_b^0$ is a solution in $S_Q(\alpha)$ with the maximum number of 0 elements over all solutions $\mathbf{x}$ satisfying $\mathbf{c}^T\mathbf{x} = \mathbf{c}^T\mathbf{x}^0$. This contradiction shows that the vector $\mathbf{x}_b^0$ must be a basic solution.

This completes the proof of the theorem. $\square$

Theorem 4.1.1 reduces the problem of finding an optimal solution for an instance of the LINEAR PROGRAMMING problem to the problem of finding an optimal basic solution for the instance. According to the theorem, if the instance has an optimal solution, then the instance must have an optimal solution that is a basic solution. Note that in general there are infinitely many solutions to a given instance while the number of basic solutions is always finite — it is bounded by the number of ways of choosing $m$ columns from the $n$ columns of the matrix $\mathbf{A}$. Moreover, all these basic solutions can be constructed systematically: pick every $m$ columns from the matrix $\mathbf{A}$, check if they are linearly independent. In case the $m$ columns are linearly independent, a unique $m$-dimensional vector $\mathbf{x} = \mathbf{B}^{-1}\mathbf{b}$ can be constructed using standard linear algebra techniques, where $\mathbf{B}$ is the submatrix consisting of the $m$ columns of $\mathbf{A}$. Now if this vector $\mathbf{x}$ also satisfies $\mathbf{x} \geq 0$, then we can expand $\mathbf{x}$ into an $n$-dimensional vector $\mathbf{x}_0$ by inserting properly $n - m$ 0's. The vector $\mathbf{x}_0$ is then the basic solution with these $m$ linearly independent columns as basic columns.

Algorithmically, however, there can be still too many basic solutions for us to search for the optimal one — the number of ways of choosing $m$ columns from the $n$ columns of the matrix $\mathbf{A}$ is $\binom{n}{m}$, which is of order $\Theta(n^m)$. In the next section, we introduce the simplex method, which provides a more effective way to search for an optimal basic solution among all basic solutions.

Theorem 4.1.1 has an interesting interpretation from the view of geometry. Given an instance $\alpha$ of the LINEAR PROGRAMMING problem, each solution $\mathbf{x}$ to $\alpha$ can be regarded as a point in the $n$-dimensional Euclidean space $\mathcal{E}^n$. Thus, the solution set $S_Q(\alpha)$ of $\alpha$ is a subset in the Euclidean space $\mathcal{E}^n$. In fact, $S_Q(\alpha)$ is a *convex set* in $\mathcal{E}^n$ in the sense that for any two points $\mathbf{x}$ and $\mathbf{y}$ in $S_Q(\alpha)$, the entire line segment connecting $\mathbf{x}$ and $\mathbf{y}$ is also in $S_Q(\alpha)$. An example of convex sets in 3-dimensional Euclidean space $\mathcal{E}^3$ is a convex polyhedron. An *extreme point* in a convex set $S$ is a point that is not an interior point of any line segment in $S$. For example, each vertex in a convex polyhedron $P$ in $\mathcal{E}^3$ is an extreme point of $P$. It can be formally proved that the basic solutions in $S_Q(\alpha)$ correspond exactly to the extreme points in $S_Q(\alpha)$. From this point of review, Theorem 4.1.1 claims that if

$S_Q(\alpha)$ is not empty then $S_Q(\alpha)$ has at least one extreme point, and that if a point in $S_Q(\alpha)$ achieves the optimal objective function value, then some extreme point in $S_Q(\alpha)$ should also achieve the optimal objective function value.

## 4.2 The simplex method

Theorem 4.1.1 claims that in order to solve the LINEAR PROGRAMMING problem, we only need to concentrate on basic solutions. This observation motivates the classical *simplex method*. Essentially, the simplex method starts with a basic solution, and repeatedly moves from a basic solution to a better basic solution until the optimal basic solution is achieved. Three immediate questions are suggested by this approach:

1. How do we find the first basic solution?
2. How do we move from one basic solution to a better basic solution? and
3. How do we realize that an optimal basic solution has been achieved?

We first discuss the solutions to the second and the third questions. A solution to the first question can be easily obtained when the solutions to the second and the third are available.

Many arguments in the LINEAR PROGRAMMING problem are substantially simplified upon the introduction of the following assumption.

> **Nondegeneracy Assumption.** For an instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ of the LINEAR PROGRAMMING problem, we assume that all basic solutions to $\alpha$ are nondegenerate.

This assumption is invoked throughout our development of the simplex method, since when it does not hold the simplex method can break down if it is not suitably amended. This assumption, however, should be regarded as one made primarily for convenience, since all arguments can be extended to include degeneracy, and the simplex method itself can be easily modified to account for it. After the whole system of methods is established, we will mention briefly how the situation of degeneracy is handled.

In the following discussion, we will fix an instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ of the LINEAR PROGRAMMING problem, where $\mathbf{b}$ is an $m$-dimensional vector, $\mathbf{c}$ is an $n$-dimensional vector, $m \leq n$, and $\mathbf{A}$ is an $m \times n$ matrix whose $m$ rows are linearly independent. Let the $n$ column vectors of the matrix $\mathbf{A}$ be $\mathbf{a}_1$, $\mathbf{a}_2$, ..., $\mathbf{a}_n$.

### How to move to a neighbor basic solution

Let $\mathbf{x}$ be a basic solution to the instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ such that the $i_1$th, $i_2$th, ..., $i_m$th elements in $\mathbf{x}$ are positive and all other elements in $\mathbf{x}$ are 0. Let $\mathbf{x}'$ be another basic solution to $\alpha$ such that the $i'_1$th, $i'_2$th, ..., $i'_m$th elements in $\mathbf{x}'$ are positive and all other elements in $\mathbf{x}'$ are 0. The basic solution $\mathbf{x}'$ is a *neighbor basic solution to* $\mathbf{x}$ if the index sets $\{t_1, \ldots, t_m\}$ and $\{t'_1, \ldots, t'_m\}$ have $m-1$ indices in common. For a give basic solution $\mathbf{x}$, the simplex method looks at neighbor basic solutions to $\mathbf{x}$ and tries to find one that is better than the current basic solution $\mathbf{x}$.

For the convenience of our discussion, we will suppose that the basic solution $\mathbf{x}$ has the first $m$ elements being positive:

$$\mathbf{x} = (x_1, \ldots, x_m, 0, \ldots, 0) \tag{4.11}$$

Since $\mathbf{x}$ is a basic solution to the instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$, we have

$$x_1\mathbf{a}_1 + x_2\mathbf{x}_2 + \cdots + x_m\mathbf{a}_m = \mathbf{b} \tag{4.12}$$

By the definition, the $m$ $m$-dimensional vectors $\mathbf{a}_1$, ..., $\mathbf{a}_m$ are linearly independent. Therefore, every column vector $\mathbf{a}_q$ of the matrix $\mathbf{A}$ can be represented as a linear combination of the vectors $\mathbf{a}_1$, ..., $\mathbf{a}_m$:

$$\mathbf{a}_q = y_{1q}\mathbf{a}_1 + y_{2q}\mathbf{a}_2 + \cdots + y_{mq}\mathbf{a}_m \quad \text{for } \ q = 1, \ldots, n \tag{4.13}$$

or

$$y_{1q}\mathbf{a}_1 + y_{2q}\mathbf{a}_2 + \cdots + y_{mq}\mathbf{a}_m - \mathbf{a}_q = \mathbf{0} \tag{4.14}$$

where $\mathbf{0}$ is the $m$-dimensional vector with all elements equal to 0. Let $\epsilon$ be a constant. Substract $\epsilon$ times the equality (4.14) from the equality (4.12),

$$(x_1 - \epsilon y_{1q})\mathbf{a}_1 + (x_2 - \epsilon y_{2q})\mathbf{a}_2 + \cdots + (x_m - \epsilon y_{mq})\mathbf{a}_m + \epsilon\mathbf{a}_q = \mathbf{b} \tag{4.15}$$

The equality (4.15) holds for all constant $\epsilon$. In particular, when $\epsilon = 0$, it corresponds to the basic solution $\mathbf{x}$ and for $\epsilon$ being a small positive number, it corresponds to a non-basic solution (note that by the Nondegeneracy Assumption, $x_i > 0$ for $1 \leq i \leq m$). Now if we let $\epsilon$ be increased from 0, then the coefficient of the vector $\mathbf{a}_q$ in the equality (4.15) is increased, and the coefficients of the other vectors $\mathbf{a}_i$, $i \neq q$, in the equality (4.15) are either increased (when $y_{iq} < 0$), unchanged (when $y_{iq} = 0$), or decreased (when $y_{iq} > 0$). Therefore, if there is a positive $y_{iq}$, then we can let $\epsilon$ be the smallest positive number that makes $x_p - \epsilon y_{pq} = 0$ for some $p$, $1 \le p \le m$. This $\epsilon$ corresponds to the value

$$\epsilon_0 = x_p/y_{pq} = \min\{x_i/y_{iq} \mid y_{iq} > 0 \text{ and } 1 \le i \le m\}$$

Note that with this value $\epsilon_0$, all coefficients in the equality (4.15) are non-negative, the coefficient of $\mathbf{a}_q$ is positive, and the coefficient of $\mathbf{a}_p$ becomes 0. Therefore, in this case, the vector

$$\begin{aligned} \mathbf{x}' &= (x_1 - \epsilon_0 y_{1q}, \ldots, x_{p-1} - \epsilon_0 y_{p-1,q}, 0, x_{p+1} - \epsilon_0 y_{p+1,q}, \ldots, \\ &\qquad \ldots, x_m - \epsilon_0 y_{mq}, 0, \ldots, 0, \epsilon_0, 0, \ldots, 0) \end{aligned} \tag{4.16}$$

satisfies $\mathbf{A}\ \mathbf{x}' = \mathbf{b}$ and $\mathbf{x}' \ge 0$, and has at most $m$ nonzero elements, where the element $\epsilon_0$ in $\mathbf{x}'$ is at the $q$th position. These $m$ possibly nonzero elements in $\mathbf{x}'$ correspond to the $m$ columns $\mathbf{a}_1, \ldots, \mathbf{a}_{p-1}, \mathbf{a}_{p+1}, \ldots, \mathbf{a}_m, \mathbf{a}_q$ of the matrix $\mathbf{A}$. By our assumption $y_{pq} > 0$, thus by equality (4.14), we have

$$\begin{aligned} \mathbf{a}_p &= (-y_{1q}/y_{pq})\mathbf{a}_1 + \cdots + (-y_{p-1,q}/y_{pq})\mathbf{a}_{p-1} + (-y_{p+1,q}/y_{pq})\mathbf{a}_{p+1} + \\ &\qquad + \cdots + (-y_{mq}/y_{pq})\mathbf{a}_m + (1/y_{pq})\mathbf{a}_q \end{aligned} \tag{4.17}$$

That is, the vector $\mathbf{a}_p$ can be represented by a linear combination of the vectors $\mathbf{a}_1, \ldots, \mathbf{a}_{p-1}, \mathbf{a}_{p+1}, \ldots, \mathbf{a}_m, \mathbf{a}_q$. Since the vectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ are linearly independent, Equality (4.17) implies that the vectors $\mathbf{a}_1, \ldots, \mathbf{a}_{p-1}, \mathbf{a}_{p+1}, \ldots, \mathbf{a}_m, \mathbf{a}_q$ are linearly independent (see Appendix C). Hence, the vector $\mathbf{x}'$ is in fact a basic solution to the instance $\alpha = (\mathbf{b}, \mathbf{c}, \mathbf{A})$. Moreover, $\mathbf{x}'$ is a neighbor basic solution to the basic solution $\mathbf{x}$.

Let us consider how each column vector $\mathbf{a}_i$ of the matrix $\mathbf{A}$ is represented by a linear combination of this new group of linearly independent vectors $\mathbf{a}_1, \ldots, \mathbf{a}_{p-1}, \mathbf{a}_{p+1}, \ldots, \mathbf{a}_m, \mathbf{a}_q$. By equality (4.13), we have

$$\mathbf{a}_i = y_{1i}\mathbf{a}_1 + y_{2i}\mathbf{a}_2 + \cdots + y_{mi}\mathbf{a}_m \tag{4.18}$$

Replace $\mathbf{a}_p$ in (4.18) by the expression in (4.17) and reorganize the equality, we get

$$\begin{aligned} \mathbf{a}_i &= (y_{1i} - y_{pi}y_{1q}/y_{pq})\mathbf{a}_1 + \cdots + (y_{p-1,i} - y_{pi}y_{p-1,q}/y_{pq})\mathbf{a}_{p-1} \\ &\qquad + (y_{pi}/y_{pq})\mathbf{a}_q + (y_{p+1,i} - y_{pi}y_{p+1,q}/y_{pq})\mathbf{a}_{p+1} + \\ &\qquad + \cdots + (y_{mi} - y_{pi}y_{mq}/y_{pq})\mathbf{a}_m \end{aligned} \tag{4.19}$$

Thus, the column $\mathbf{a}_q$ replaces the column $\mathbf{a}_p$ and becomes the $p$th basic column for the basic solution.

The above transformation from the basic solution $\mathbf{x}$ to the neighbor basic solution $\mathbf{x}'$ can be conveniently managed in the form of a tableau. For the basic solution $\mathbf{x} = (x_1, \ldots, x_m, 0, \ldots, 0)$, and suppose that the last $n - m$ columns $\mathbf{a}_q$, $m + 1 \le q \le n$, of the matrix $\mathbf{A}$ are given by the linear combinations of the columns $\mathbf{a}_1$, $\mathbf{a}_2$, ..., $\mathbf{a}_m$ in equality (4.13), then the tableau corresponding to the basic solution $\mathbf{x}$ is given as

| $\mathbf{a}_1$ | $\cdots$ | $\mathbf{a}_p$ | $\cdots$ | $\mathbf{a}_m$ | $\mathbf{a}_{m+1}$ | $\cdots$ | $\mathbf{a}_q$ | $\cdots$ | $\mathbf{a}_n$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $\cdots$ | 0 | $\cdots$ | 0 | $y_{1,m+1}$ | $\cdots$ | $y_{1q}$ | $\cdots$ | $y_{1n}$ | $x_1$ |
| $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ |
| 0 | $\cdots$ | 1 | $\cdots$ | 0 | $y_{p,m+1}$ | $\cdots$ | $y_{pq}$ | $\cdots$ | $y_{pn}$ | $x_p$ |
| $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ |
| 0 | $\cdots$ | 0 | $\cdots$ | 1 | $y_{m,m+1}$ | $\cdots$ | $y_{mq}$ | $\cdots$ | $y_{mn}$ | $x_m$ |

In order to move from the basic solution $\mathbf{x}$ to the neighbor basic solution $\mathbf{x}'$ by replacing the column $\mathbf{a}_p$ by the column $\mathbf{a}_q$ (assume that $y_{pq} > 0$ and $x_p/y_{pq}$ is the minimum $x_i/y_{iq}$ over all $i$ such that $1 \le i \le m$ and $y_{iq} > 0$), we only need to perform the following row transformations on the tableau: (1) divide the $p$th row of the tableau by $y_{pq}$; and (2) for each row $j$, $j \neq p$, subtract $y_{jq}$ times the $p$th row from the $j$th row. After these transformations, the tableau becomes

| $\mathbf{a}_1$ | $\cdots$ | $\mathbf{a}_p$ | $\cdots$ | $\mathbf{a}_m$ | $\mathbf{a}_{m+1}$ | $\cdots$ | $\mathbf{a}_q$ | $\cdots$ | $\mathbf{a}_n$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $\cdots$ | $y'_{1p}$ | $\cdots$ | 0 | $y'_{1,m+1}$ | $\cdots$ | 0 | $\cdots$ | $y'_{1n}$ | $x'_1$ |
| $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ |
| 0 | $\cdots$ | $y'_{pp}$ | $\cdots$ | 0 | $y'_{p,m+1}$ | $\cdots$ | 1 | $\cdots$ | $y'_{pn}$ | $x'_p$ |
| $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ |
| 0 | $\cdots$ | $y'_{mp}$ | $\cdots$ | 1 | $y'_{m,m+1}$ | $\cdots$ | 0 | $\cdots$ | $y'_{mn}$ | $x'_m$ |

Thus, the $q$th column in the tableau, which corresponds to column $\mathbf{a}_q$, now becomes a vector whose $p$th element is 1 and all other elements are 0.

Consider the $p$th column $\mathbf{a}_p$ in the tableau. We have

$$y'_{pp} = 1/y_{pq} \tag{4.20}$$

and

$$y'_{jp} = -y_{jq}/y_{pq} \quad \text{for } 1 \le i \le m \text{ and } j \neq p \tag{4.21}$$

By equality (4.17), we get

$$\mathbf{a}_p = y'_{1p}\mathbf{a}_1 + \cdots + y'_{p-1,p}\mathbf{a}_{p-1} + y'_{pp}\mathbf{a}_q + y'_{p+1,p}\mathbf{a}_{p+1} + \cdots + y'_{mp}\mathbf{a}_m$$

Therefore, the $p$th column in the new tableau gives exactly the coefficients of the linear combination for the column $\mathbf{a}_p$ in terms of the new basic columns $\mathbf{a}_1, \ldots, \mathbf{a}_{p-1}, \mathbf{a}_q, \mathbf{a}_{p+1}, \ldots, \mathbf{a}_m$.

For the $i$th column $\mathbf{a}_i$ in the tableau, where $m+1 \leq i \leq n$ and $i \neq q$, we have

$$y'_{pi} = y_{pi}/y_{pq} \tag{4.22}$$

and

$$y'_{ji} = y_{ji} - y_{jq}y_{pi}/y_{pq} \quad \text{for } 1 \leq j \leq m \text{ and } j \neq p \tag{4.23}$$

By eqality (4.19), the $i$th column in the tableau gives exactly the coefficients of the linear combination for the column $\mathbf{a}_i$ in terms of the new basic columns $\mathbf{a}_1, \ldots, \mathbf{a}_{p-1}, \mathbf{a}_q, \mathbf{a}_{p+1}, \ldots, \mathbf{a}_m$.

Finally, let us consider the last column in the tableau. We have

$$x'_p = x_p/y_{pq} = \epsilon_0$$

and

$$x'_j = x_j - y_{jq}x_p/y_{pq} = x_j - \epsilon_0 y_{jq} \quad \text{for } 1 \leq j \leq m \text{ and } j \neq p$$

Thus, the last column of the tableau gives exactly the values for the new basic solution $\mathbf{x}'$.

Therefore, the row transformations performed on the tableau for the basic solution $\mathbf{x}$ result in the tableau for the new basic solution $\mathbf{x}'$.

We should point out that in the above discussion, the basic columns for a basic solution are not ordered by their indices. Instead, they are ordered by the positions of the element 1 in the corresponding columns in the tableau. For example, the column $\mathbf{a}_q$ becomes the $p$th basic column because in the $q$th column of the new tableau, the $p$th element is 1 and all other elements are 0. Hence, the $p$th row in the new tableau corresponds to the coefficients for the column $\mathbf{a}_q$, i.e., $y'_{pi}$ is the coefficient of $\mathbf{a}_q$ in the linear combination for $\mathbf{a}_i$ in terms of $\mathbf{a}_1, \ldots, \mathbf{a}_{p-1}, \mathbf{a}_q, \mathbf{a}_{p+1}, \ldots, \mathbf{a}_m$, and $x'_p$ is the value of the $q$th element in the basic solution $\mathbf{x}'$.

In general case, suppose that we have a basic solution $\mathbf{x}$ in which the $i_1$th, $i_2$th, $\ldots$, $i_m$th elements $x_{i_1}, x_{i_2}, \ldots, x_{i_m}$ are positive, and the tableau $\mathcal{T}$ for $\mathbf{x}$ such that (1) for each $j$, $1 \leq j \leq m$, the $i_j$th column of $\mathcal{T}$ has the $j$th element equal to 1 and all other elements equal to 0; (2) for each $j$,

$1 \le j \le m$, the $j$th element in the last column of $\mathcal{T}$ is $x_{i_j}$; and (3) for each $i$, $1 \le i \le n$, the $i$th column of $\mathcal{T}$ is $(y_{1i}, y_{2i}, \ldots, y_{mi})^T$ if the $i$th column $\mathbf{a}_i$ of the matrix $\mathbf{A}$ is represented by the linear combination of the columns $\mathbf{a}_{i_1}$, $\mathbf{a}_{i_2}$, …, $\mathbf{a}_{i_m}$ as

$$\mathbf{a}_i = y_{1i}\mathbf{a}_{i_1} + y_{2i}\mathbf{a}_{i_2} + \cdots + y_{mi}\mathbf{a}_{i_m}$$

In order to replace the column $\mathbf{a}_{i_p}$ in the basic solution $\mathbf{x}$ by a new column $\mathbf{a}_q$ to obtain a new basic solution $\mathbf{x}'$, we first require that the element $y_{pq}$ in the $q$th column of the tableau $\mathcal{T}$ be positive, and that $x_p/y_{pq}$ be the minimum over all $x_j/y_{jq}$ with $y_{jq} > 0$. With these conditions satisfied, perform the following row transformation on the tableau $\mathcal{T}$: (1) divide the $p$th row by $y_{pq}$; and (2) for each $j$, $1 \le j \le m$ and $j \ne p$, subtract $y_{jq}$ times the $p$th row from the $j$th row. The resulting tableau by these row transformations is exactly the tableau for the new basic solution $\mathbf{x}'$ obtained by adding the $q$th column and deleting the $i_p$th column from the basic solution $\mathbf{x}$.

**Example 4.2.1** Consider the following instance $\alpha = (\mathbf{b}, \mathbf{c}, \mathbf{A})$ of the LINEAR PROGRAMMING problem

$$\begin{aligned} \text{minimize} \quad & x_6 \\ \text{subject to} \quad & 3x_1 + 5x_2 + x_3 = 24 \\ & 4x_1 + 2x_2 + x_4 = 16 \\ & x_1 + x_2 - x_5 + x_6 = 3 \\ & x_1, x_2, x_3, x_4, x_5, x_6 \ge 0 \end{aligned} \tag{4.24}$$

Let the six column vectors of the matrix $\mathbf{A}$ be $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$, $\mathbf{a}_4$, $\mathbf{a}_5$, $\mathbf{a}_6$. The vector $\mathbf{x}$= $(0, 0, 24, 16, 0, 3)$ is obviously a basic solution to the instance $\alpha$ with the basic columns $\mathbf{a}_3$, $\mathbf{a}_4$, and $\mathbf{a}_6$. The other columns of $\mathbf{A}$ can be represented by linear combinations of the columns $\mathbf{a}_3$, $\mathbf{a}_4$, and $\mathbf{a}_6$ as follows.

$$\begin{aligned} \mathbf{a}_1 &= 3\mathbf{a}_3 + 4\mathbf{a}_4 + \mathbf{a}_6 \\ \mathbf{a}_2 &= 5\mathbf{a}_3 + 2\mathbf{a}_4 + \mathbf{a}_6 \\ \mathbf{a}_5 &= -\mathbf{a}_6 \end{aligned}$$

Thus, the tableau for the basic solution $\mathbf{x}$ is

| $\mathbf{a}_1$ | $\mathbf{a}_2$ | $\mathbf{a}_3$ | $\mathbf{a}_4$ | $\mathbf{a}_5$ | $\mathbf{a}_6$ | |
|---|---|---|---|---|---|---|
| 3 | 5 | 1 | 0 | 0 | 0 | 24 |
| 4 | 2 | 0 | 1 | 0 | 0 | 16 |
| 1 | 1 | 0 | 0 | −1 | 1 | 3 |

It should not be surprising that the tableau for the basic solution $\mathbf{x}$ consists of the columns of the matrix $\mathbf{A}$ plus the vecter $\mathbf{b}$. This is because that the three columns $\mathbf{a}_3$, $\mathbf{a}_4$, and $\mathbf{a}_6$ are three linearly independent unit vectors in the 3-dimensional Euclidean space $\mathcal{E}^3$.

Now suppose that we want to construct a new basic solution by replacing a column for the basic solution $\mathbf{x}$ by the second column $\mathbf{a}_2$ of the matrix $\mathbf{A}$. All elements in the second column of the tableau are positive. Thus, we only need to check the ratios. We have

$$x_1/y_{12} = 24/5 = 4.8 \qquad x_2/y_{22} = 16/2 = 8 \qquad x_3/y_{32} = 3/1 = 3$$

Thus, we will replace the column $\mathbf{a}_6$ by the column $\mathbf{a}_2$ (note that the 3rd row of the tableau corresponds to the 3rd basic column for $\mathbf{x}$, which is $\mathbf{a}_6$). Dividing the third row of the tableau by $y_{32}$ does not change the tableau since $y_{32} = 1$. Then we subtract from the second row by 2 times the third row, and subtract from the first row by 5 times the third row. We obtain the final tableau

| $\mathbf{a}_1$ | $\mathbf{a}_2$ | $\mathbf{a}_3$ | $\mathbf{a}_4$ | $\mathbf{a}_5$ | $\mathbf{a}_6$ | |
|---|---|---|---|---|---|---|
| $-2$ | 0 | 1 | 0 | 5 | $-5$ | 9 |
| 2 | 0 | 0 | 1 | 2 | $-2$ | 10 |
| 1 | 1 | 0 | 0 | $-1$ | 1 | 3 |

The new basic solution $\mathbf{x}'$ corresponds to the columns $\mathbf{a}_3$, $\mathbf{a}_4$, and $\mathbf{a}_2$ (again note that though $\mathbf{a}_2$ has the smallest index, it is the 3rd basic column for $\mathbf{x}'$). The value of $\mathbf{x}'$ can be read directly from the last column of the tableau, which is $\mathbf{x}' = (0, 3, 9, 10, 0, 0)$. The coefficients of the linear combinations of the columns $\mathbf{a}_1$, $\mathbf{a}_5$, and $\mathbf{a}_6$ in terms of the columns $\mathbf{a}_2$, $\mathbf{a}_3$, and $\mathbf{a}_4$ can also read directly from the tableau:

$$\begin{aligned}
\mathbf{a}_1 &= -2\mathbf{a}_3 + 2\mathbf{a}_4 + \mathbf{a}_2 = \mathbf{a}_2 - 2\mathbf{a}_3 + 2\mathbf{a}_4 \\
\mathbf{a}_5 &= 5\mathbf{a}_3 + 2\mathbf{a}_4 - \mathbf{a}_2 = -\mathbf{a}_2 + 5\mathbf{a}_3 + 2\mathbf{a}_4 \\
\mathbf{a}_6 &= -5\mathbf{a}_3 - 2\mathbf{a}_4 + \mathbf{a}_2 = \mathbf{a}_2 - 5\mathbf{a}_3 - 2\mathbf{a}_4
\end{aligned}$$

All these can be verified easily in the original instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$.

### How to move to a better neighbor basic solution

We have described how the basic solution $\mathbf{x} = (x_1, \ldots, x_m, 0, \ldots, 0)$ for the instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ of the LINEAR PROGRAMMING problem can be converted to a neighbor basic solution

$$\begin{aligned}\mathbf{x}' &= (x_1 - \epsilon_0 y_{1q}, \ldots, x_{p-1} - \epsilon_0 y_{p-1,q}, 0, x_{p+1} - \epsilon_0 y_{p+1,q}, \ldots, \\ &\qquad \ldots, x_m - \epsilon_0 y_{mq}, 0, \ldots, 0, \epsilon_0, 0, \ldots, 0)\end{aligned}$$

by replacing the column $\mathbf{a}_p$ by the column $\mathbf{a}_q$, where $y_{pq} > 0$ and $\epsilon_0 = x_p/y_{pq}$ is the minimum over all $x_i/y_{iq}$ with $y_{iq} > 0$. Since we want to minimize the value of the objective function $\mathbf{c}^T\mathbf{x}$, we would like to have $\mathbf{x}'$ to give a smaller objective function value. Consider the objective function values on these two basic solutions:

$$\mathbf{c}^T\mathbf{x} = c_1x_1 + c_2x_2 + \cdots + c_mx_m$$

and

$$\begin{aligned}\mathbf{c}^T\mathbf{x}' &= c_1(x_1 - \epsilon_0 y_{1q}) + \cdots + c_{p-1}(x_{p-1} - \epsilon_0 y_{p-1,q}) + \\ &\qquad + c_{p+1}(x_{p+1} - \epsilon_0 y_{p+1,q}) + \cdots + c_m(x_m - \epsilon_0 y_{mq}) + c_q\epsilon_0 \\ &= (\sum_{j=1}^{m} c_jx_j) - c_px_p + \epsilon_0(c_q - \sum_{j=1}^{m} c_jy_{jq}) + \epsilon_0 c_p y_{pq}\end{aligned}$$

Since $\epsilon_0 = x_p/y_{pq}$, we have $\epsilon_0 c_p y_{pq} = c_p x_p$. Thus, the last equality gives

$$\mathbf{c}^T\mathbf{x}' = \mathbf{c}^T\mathbf{x} + \epsilon_0(c_q - \sum_{j=1}^{m} c_jy_{jq})$$

Thus, the basic solution $\mathbf{x}'$ gives a better (i.e., smaller) objective function value $\mathbf{c}^T\mathbf{x}'$ than $\mathbf{c}^T\mathbf{x}$ if and only if $\epsilon_0(c_q - \sum_{j=1}^{m} c_jy_{jq}) < 0$. This thus gives us a guideline for choosing a column to construct a better neighbor basic solution. The constant $c_q - \sum_{j=1}^{m} c_jy_{jq}$ plays such a central role in the development of the simplex method, it is convenient to introduce somewhat abbreviated notation for it. Denote by $r_q$ the constant $c_q - \sum_{j=1}^{m} c_jy_{jq}$ for $1 \leq q \leq n$, and call them the *reduced cost coefficients.* The above discussion gives us the following lemma.

**Lemma 4.2.1** *Let* $\mathbf{x}$ *and* $\mathbf{x}'$ *be the basic solutions as given above. The basic solution* $\mathbf{x}'$ *gives a better (i.e., smaller) objective function value* $\mathbf{c}^T\mathbf{x}'$ *than* $\mathbf{c}^T\mathbf{x}$ *if and only if the reduced cost coefficient*

$$r_q = c_q - \sum_{j=1}^{m} c_jy_{jq}$$

*is less than* 0.

| $\mathbf{a}_1$ | $\cdots$ | $\mathbf{a}_p$ | $\cdots$ | $\mathbf{a}_m$ | $\mathbf{a}_{m+1}$ | $\cdots$ | $\mathbf{a}_q$ | $\cdots$ | $\mathbf{a}_n$ | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | $\cdots$ | 0 | $\cdots$ | 0 | $y_{1,m+1}$ | $\cdots$ | $y_{1q}$ | $\cdots$ | $y_{1n}$ | $x_1$ |
| $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ |
| 0 | $\cdots$ | 1 | $\cdots$ | 0 | $y_{p,m+1}$ | $\cdots$ | $y_{pq}$ | $\cdots$ | $y_{pn}$ | $x_p$ |
| $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ | | $\cdot$ | | $\cdot$ | $\cdot$ |
| 0 | $\cdots$ | 0 | $\cdots$ | 1 | $y_{m,m+1}$ | $\cdots$ | $y_{mq}$ | $\cdots$ | $y_{mn}$ | $x_m$ |
| 0 | $\cdots$ | 0 | $\cdots$ | 0 | $r_m + 1$ | $\cdots$ | $r_q$ | $\cdots$ | $r_n$ | $z_0$ |

Figure 4.1: The general tableau format for the basic solution $\mathbf{x}$

The nice thing is that the reduced cost coefficients $r_i$ as well as the objective function value $\mathbf{c}^T\mathbf{x}$ can also be made a row in the tableau for the basic solution $\mathbf{x}$ and calculated by formal row transformations of the tableau. For this, we create a new row, the $(m+1)$st row, in the tableau so that the element corresponding to the vector $\mathbf{a}_i$ in this row is $r_i$ (note that by the formula if $1 \leq i \leq m$ then $r_i = 0$), and the element in the last column of this row is the value $z_0 = -\mathbf{c}^T\mathbf{x}$. The new tableau format is given in Figure 4.1.

Now suppose that we replace the basic column $\mathbf{a}_p$ for the basic solution $\mathbf{x}$ by the column $\mathbf{a}_q$ to construct the basic solution $\mathbf{x}'$. Then in addition to the row transformations described before to obtain the coefficients $y'_{ji}$ and $\mathbf{x}'$, we also (after dividing the $p$th two by $y_{pq}$) subtract $r_q$ times the $p$th row from the $(m+1)$st row. We verify that this row transformation converts the $(m+1)$st row to give exactly the reduced cost coefficients and the objective function value for the new basic solution $\mathbf{x}'$.

By the above described procedure, the new value $r'_i$ of the $i$th element in the $(m+1)$st row in the tableau is

$$
\begin{aligned}
r'_i &= r_i - r_q y_{pi}/y_{pq} \\
&= (c_i - \sum_{j=1}^{m} c_j y_{ji}) - (c_q - \sum_{j=1}^{m} c_j y_{jq}) y_{pi}/y_{pq} \\
&= c_i - \sum_{j=1}^{m} c_j y_{ji} - c_q y_{pi}/y_{pq} + \sum_{j=1}^{m} c_j y_{jq} y_{pi}/y_{pq} \\
&= c_i - \sum_{j=1}^{m} c_j (y_{ji} - y_{jq} y_{pi}/y_{pq}) - c_q y_{pi}/y_{pq}
\end{aligned}
$$

By equalities (4.22) and (4.23), and note $y'_{pi} = y_{pi} - y_{pq}y_{pi}/y_{pq} = 0$, we get

$$\begin{aligned} r'_i &= c_i - \sum_{j=1}^{m} c_j y'_{ji} - c_q y'_{pi} \\ &= c_i - (c_1 y'_{1i} + \cdots + c_{p-1} y'_{p-1,i} + c_{p+1} y'_{p+1,i} + \cdots + c_m y'_{mi} + c_q y'_{pi}) \end{aligned}$$

Therefore, the value $r'_i$ is exactly the reduced cost coefficient for the column $\mathbf{a}_i$ in the new basic solution $\mathbf{x}'$.

Consider the new value $z'_0$ in the last column of the $(m+1)$st row. By the procedure, the new value is equal to

$$\begin{aligned} z'_0 &= z_0 - r_q x_p / y_{pq} \\ &= -\mathbf{c}^T\mathbf{x} - (c_q - \sum_{j=1}^{m} c_j y_{jq}) x_p / y_{pq} \\ &= -\sum_{j=1}^{m} c_j x_j + \sum_{j=1}^{m} c_j y_{jq} x_p / y_{pq} - c_q x_p / y_{pq} \end{aligned}$$

Let $\epsilon_0 = x_p / y_{pq}$ and note that $x_p - \epsilon_0 y_{pq} = 0$, we get

$$\begin{aligned} z'_0 &= -\sum_{j=1}^{m} c_j x_j + \sum_{j=1}^{m} c_j \epsilon_0 y_{jq} - c_q \epsilon_0 \\ &= -(\sum_{j=1}^{m} c_j (x_j - \epsilon_0 y_{jq}) + c_q \epsilon_0) \\ &= -(c_1(x_1 - \epsilon_0 y_{1q}) + \cdots + c_{p-1}(x_{p-1} - \epsilon_0 y_{p-1,q}) + \\ &\qquad + c_{p+1}(x_{p+1} - \epsilon_0 y_{p+1,q}) + \cdots + c_m(x_m - \epsilon_0 y_{mq}) + c_q \epsilon_0) \end{aligned}$$

According to equality (4.16), $z'_0$ gives exactly the value $-\mathbf{c}^T\mathbf{x}'$

Therefore, after the row transformations of the tableau, the $(m+1)$st row of the tableau gives exactly the reduced cost coefficients and the objective function value for the new basic solution $\mathbf{x}'$.

**Example 4.2.2.** Let us reconsider the instance $\alpha = (\mathbf{b}, \mathbf{c}, \mathbf{A})$ of the LINEAR PROGRAMMING problem given in Example 4.2.1.

$$\begin{array}{ll} \text{minimize} & x_6 \\ \text{subject to} & 3x_1 + 5x_2 + x_3 = 24 \\ & 4x_1 + 2x_2 + x_4 = 16 \\ & x_1 + x_2 - x_5 + x_6 = 3 \\ & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{array}$$

**Algorithm.** **TableauMove**$(\mathcal{T}, p, q)$

Input: an $(m+1) \times (n+1)$ tableau $\mathcal{T}$,
$1 \le p \le m$, $1 \le q \le n$, $\mathcal{T}[p,q] \ne 0$

1. $y_{pq} = \mathcal{T}[p,q]$;
   **for** $i = 1$ **to** $n+1$ **do** $\mathcal{T}[p,i] = \mathcal{T}[p,i]/y_{pq}$;
2. **for** $(1 \le j \le m+1)$ and $(j \ne p)$ **do**
   $y_{jq} = \mathcal{T}[j,q]$;
   **for** $i = 1$ **to** $n+1$ **do** $\mathcal{T}[j,i] = \mathcal{T}[j,i] - \mathcal{T}[p,i] * y_{jq}$;

Figure 4.2: Tableau transformation

The extended tableau for the basic solution $\mathbf{x}= (0,0,24,16,0,3)$, which has an objective function value 3, is as follows.

| $\mathbf{a}_1$ | $\mathbf{a}_2$ | $\mathbf{a}_3$ | $\mathbf{a}_4$ | $\mathbf{a}_5$ | $\mathbf{a}_6$ | |
|---|---|---|---|---|---|---|
| 3 | 5 | 1 | 0 | 0 | 0 | 24 |
| 4 | 2 | 0 | 1 | 0 | 0 | 16 |
| 1 | 1 | 0 | 0 | $-1$ | 1 | 3 |
| $-1$ | $-1$ | 0 | 0 | 1 | 0 | $-3$ |

If we replace the column $\mathbf{a}_6$ by the column $\mathbf{a}_2$ (note $r_2 < 0$), then after the row transformations, we obtain the tableau

| $\mathbf{a}_1$ | $\mathbf{a}_2$ | $\mathbf{a}_3$ | $\mathbf{a}_4$ | $\mathbf{a}_5$ | $\mathbf{a}_6$ | |
|---|---|---|---|---|---|---|
| $-2$ | 0 | 1 | 0 | 5 | $-5$ | 9 |
| 2 | 0 | 0 | 1 | 2 | $-2$ | 10 |
| 1 | 1 | 0 | 0 | $-1$ | 1 | 3 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Thus, we obtained an improved basic solution $\mathbf{x}' = (0,3,9,10,0,0)$ that has an objective function value 0.

We summarize the above discussion on tableau transformations into the algorithm given in Figure 4.2. Thus, suppose that $\mathcal{T}$ is the tableau for the basic solution $\mathbf{x}$ with $\mathcal{T}[m+1,q] < 0$ and let $p$ be the index such that $\mathcal{T}[p,q] > 0$ and the ratio $\mathcal{T}[p,n+1]/\mathcal{T}[p,q]$ is the minimum over all ratios $\mathcal{T}[j,n+1]/\mathcal{T}[j,q]$ with $\mathcal{T}[j,q] > 0$, then according to Lemma 4.2.1, the algorithm **TableauMove**$(\mathcal{T}, p, q)$ will result in the tableau for a neighbor basic solution $\mathbf{x}'$, by replacing the $p$th basic column for $\mathbf{x}$ by the $q$th column, such that $\mathbf{c}^T\mathbf{x}' < \mathbf{c}^T\mathbf{x}$.

## When an optimal solution is achieved

Suppose that we have the basic solution $\mathbf{x} = (x_1, \ldots, x_m, 0, \ldots, 0)$ and the tableau $\mathcal{T}$ in Figure 4.1 for $\mathbf{x}$. By Lemma 4.2.1, if there is a column $q$ in $\mathcal{T}$ such that $r_q < 0$ and there is a positive element $y_{pq}$ in the $q$th column, then we can perform the row transformations to obtain a better basic solution $\mathbf{x}'$ with $\mathbf{c}^T\mathbf{x}'$, thus achieving an improvement. What if no such a column $q$ exists in the tableau?

If no such a column exists in the tableau, then either we have $r_q \geq 0$ for all $q$, $1 \leq q \leq n$, or we have $r_q < 0$ for some $q$ but $y_{pq} \leq 0$ for all $p$, $1 \leq p \leq m$. We consider these two cases separately below.

**CASE 1.** all values $r_q \geq 0$.

We prove that in this case, the basic solution $\mathbf{x}$ is an optimal solution.

Let $\mathbf{x}' = (x'_1, x'_2, \ldots, x'_n)$ be any solution to the instance $\alpha = (\mathbf{b}, \mathbf{c}, \mathbf{A})$. Thus, we have $x'_i \geq 0$, $1 \leq i \leq n$, and

$$x'_1\mathbf{a}_1 + x'_2\mathbf{a}_2 + \cdots + x'_n\mathbf{a}_n = \mathbf{b} \tag{4.25}$$

Since $\mathbf{x} = (x_1, \ldots, x_m, 0, \ldots, 0)$ is a basic solution, each column $\mathbf{a}_i$ of $\mathbf{A}$ can be represented by a linear combination of $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$:

$$\mathbf{a}_i = \sum_{j=1}^{m} y_{ji}\mathbf{a}_j, \qquad m+1 \leq i \leq n \tag{4.26}$$

Replace each $\mathbf{a}_i$ in (4.25), $m+1 \leq i \leq n$, by the equality (4.26), we obtain

$$\begin{aligned} x'_1\mathbf{a}_1 + \cdots + x'_m\mathbf{a}_m + x'_{m+1}\sum_{j=1}^{m} y_{j,m+1}\mathbf{a}_j + & \\ + x'_{m+2}\sum_{j=1}^{m} y_{j,m+2}\mathbf{a}_j + \cdots + x'_n\sum_{j=1}^{m} y_{jn}\mathbf{a}_j &= \mathbf{b} \end{aligned}$$

Regrouping the terms gives

$$\begin{aligned} (x'_1 + \sum_{i=m+1}^{n} x'_i y_{1i})\mathbf{a}_1 + (x'_2 + \sum_{i=m+1}^{n} x'_i y_{2i})\mathbf{a}_2 + & \\ + \cdots + (x'_m + \sum_{i=m+1}^{n} x'_i y_{mi})\mathbf{a}_m &= \mathbf{b} \end{aligned} \tag{4.27}$$

Since $\mathbf{x} = (x_1, \ldots, x_m, 0, \ldots, 0)$ is a basic solution, we also have

$$x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_m\mathbf{a}_m = \mathbf{b} \tag{4.28}$$

Compare equalities (4.27) and (4.28). Since the columns $\mathbf{a}_1, \ldots, \mathbf{a}_m$ are linearly independent, the vector $\mathbf{b}$ has a unique representation in terms of the linear combination of $\mathbf{a}_1, \ldots, \mathbf{a}_m$. Thus, we must have

$$\begin{aligned}
x_1 &= x_1' + \sum_{i=m+1}^{n} x_i' y_{1i} \\
x_2 &= x_2' + \sum_{i=m+1}^{n} x_i' y_{2i} \\
&\cdots\cdots \\
x_m &= x_m' + \sum_{i=m+1}^{n} x_i' y_{mi}
\end{aligned}$$

Bringing these values for $x_1, x_2, \ldots, x_m$ to the inner product $\mathbf{c}^T\mathbf{x}$, we get

$$\begin{aligned}
\mathbf{c}^T\mathbf{x} &= \sum_{j=1}^{n} c_j x_j = \sum_{j=1}^{m} c_j x_j \\
&= \sum_{j=1}^{m} c_j (x_j' + \sum_{i=m+1}^{n} x_i' y_{ji}) \\
&= \sum_{j=1}^{m} c_j x_j' + \sum_{j=1}^{m} \sum_{i=m+1}^{n} c_j x_i' y_{ji} \\
&= \sum_{j=1}^{m} c_j x_j' + \sum_{i=m+1}^{n} x_i' (\sum_{j=1}^{m} c_j y_{ji}) \\
&= \sum_{j=1}^{n} c_j x_j' - \sum_{i=m+1}^{n} c_i x_i' + \sum_{i=m+1}^{n} x_i' (\sum_{j=1}^{m} c_j y_{ji}) \\
&= \mathbf{c}^T\mathbf{x}' - \sum_{i=m+1}^{n} x_i' (c_i - \sum_{j=1}^{m} c_j y_{ji}) \\
&= \mathbf{c}^T\mathbf{x}' - \sum_{i=m+1}^{n} x_i' r_i
\end{aligned}$$

Since $\mathbf{x}'$ is a solution to the instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$, $x_i' \geq 0$ for $m+1 \leq i \leq n$, and by our assumption, $r_q \geq 0$ for $m+1 \leq q \leq n$, we get

$$\mathbf{c}^T\mathbf{x} = \mathbf{c}^T\mathbf{x}' - \sum_{i=m+1}^{n} x_i' r_i \leq \mathbf{c}^T\mathbf{x}';$$

Since $\mathbf{x}'$ is an arbitrary solution to the instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$, we conclude that $\mathbf{x}$ is an optimal solution to $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$. This conclusion is summarized in the following lemma.

**Lemma 4.2.2** *Let $\mathbf{x}$ be a basic solution to the instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ with the tableau given in Figure 4.1. If all reduced cost coefficients $r_q \geq 0$, $1 \leq q \leq n$, then $\mathbf{x}$ is an optimal solution.*

**Example 4.2.3.** Recall that in Example 4.2.2, we obtained the basic solution $\mathbf{x}' = (0, 3, 9, 10, 0, 0)$, which has the objective function value 0, such that all reduced cost coefficients are larger than or equal to 0 (see the last tableau in Example 4.2.2). By Lemma 4.2.2, the solution $\mathbf{x}'$ is an optimal solution to the given instance.

**CASE 2.** There is a $q$ such that $r_q < 0$ but no element in the $q$th column of the tableau is positive.

In this case, consider the equalities

$$y_{1q}\mathbf{a}_1 + y_{2q}\mathbf{a}_2 + \cdots + y_{mq}\mathbf{a}_m - \mathbf{a}_q = 0$$

and

$$x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \cdots + x_m\mathbf{a}_m = \mathbf{b}$$

Subtract from the second equality by $\epsilon$ times the first equality, where $\epsilon$ is any positive number, we get

$$(x_1 - \epsilon y_{1q})\mathbf{a}_1 + (x_2 - \epsilon y_{2q})\mathbf{a}_2 + \cdots + (x_m - \epsilon y_{mq})\mathbf{a}_m + \epsilon\mathbf{a}_q = \mathbf{b}$$

Since $x_i > 0$ for $1 \leq i \leq m$, and $y_{jq} \leq 0$ for all $1 \leq j \leq m$, we have $x_j - \epsilon y_{jq} > 0$ for all $1 \leq j \leq m$. Thus,

$$\mathbf{x}_\epsilon = (x_1 - \epsilon y_{1q}, \ldots, x_m - \epsilon y_{mq}, 0, \ldots, 0, \epsilon, 0, \ldots, 0)$$

where the element $\epsilon$ is in the $q$th position, is a solution to $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ for all positive value $\epsilon$.

Consider the objective function value $\mathbf{c}^T\mathbf{x}_\epsilon$ on the solution $\mathbf{x}_\epsilon$, we have

$$\begin{aligned}
\mathbf{c}^T\mathbf{x}_\epsilon &= \sum_{j=1}^{m} c_j(x_j - \epsilon y_{jq}) + c_q\epsilon \\
&= \sum_{j=1}^{m} c_j x_j + \epsilon(c_q - \sum_{j=1}^{m} c_j y_{jq}) \\
&= \mathbf{c}^T\mathbf{x} + \epsilon r_q
\end{aligned}$$

By our assumption, $r_q < 0$ and $\epsilon$ can be any positive number, the value $\mathbf{c}^T\mathbf{x}_\epsilon$ can be arbitrarily small, i.e., the instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ has no optimal solution. We summarize this in the following lemma.

**Lemma 4.2.3** *Let* $\mathbf{x}$ *be a basic solution to the instance* $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ *with the tableau given in Figure 4.1. If there is a reduced cost coefficients* $r_q < 0$*, and all elements in the qth column of the tableau are less than or equal to 0, then the objective function can have arbitrarily small value and the instance* $\alpha$ *has no optimal solution.*

### Degeneracy

It is possible that in the course of the simplex method described above, a degenerate basic solution occurs. Often they can be handled as a non-degenerate basic solution. However, it is possible that after a new column $\mathbf{a}_q$ is selected to replace a current basic column $\mathbf{a}_p$, the ratio $x_p/y_{pq}$ is 0, implying that the basic column $\mathbf{a}_p$ is the one to go out. This means that the new variable $x_q$ will come into the new basic solution at value 0, the objective function value will not decrease, and the new basic solution will also be degenerate. Conceivably, this process could continue for a series of steps and even worse, some degenerate basic solution may repeat in the series, leading to an endless process without being able to achieve an optimal solution. This situation is called *cycling*.

Degeneracy often occurs in large-scale real-world problems. However, cycling in such instances is very rare. Methods have been developed to avoid cyclings. In practice, however, such procedures are found to be unnecessary. When degenerate solutions are encountered, the simplex method generally does not enter cycling. However, anticycling procedures are simple, and many codes incorporate such a procedure for the sake of safety.

### How to obtain the first basic solution

Lemmas 4.2.1, 4.2.2, and 4.2.3 completely describe how we can move from a basic solution to a better basic solution and when an optimal basic solution is achieved. To describe the simplex method completely, the only thing remaining is how the first basic solution can be obtained.

A basic solution is sometimes immediately available from an instance of the LINEAR PROGRAMMING problem. For example, suppose that the instance of the LINEAR PROGRAMMING problem is given in the form

$$\text{minimize} \qquad c_1x_1 + c_2x_2 + \cdots c_nx_n$$

subject to

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\le b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\le b_2 \\
\cdots\cdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\le b_m \\
x_1 \ge 0,\ x_2 \ge 0,\ \ldots,\ x_n \ge 0
\end{aligned}
$$

with $b_i \ge 0$ for all $i$. Then in the elimination of the $\le$ signs we introduce $m$ slack variables $y_1, \ldots, y_m$ and convert it into the standard form

$$
\begin{array}{ll}
\text{minimize} & c_1x_1 + c_2x_2 + \cdots c_nx_n \\
\text{subject to} &
\end{array}
$$

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n + y_1 &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n + y_2 &= b_2 \\
\cdots\cdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n + y_m &= b_m \\
x_1 \ge 0,\ x_2 \ge 0,\ \ldots,\ x_n \ge 0 \\
y_1 \ge 0,\ y_2 \ge 0,\ \ldots,\ y_m \ge 0
\end{aligned}
$$

Obviously, the $(n+m)$-dimensional vector $(0, \ldots, 0, b_1, b_2, \ldots, b_m)$ is a basic solution to this new instance, from which the simplex method can be initiated. In fact, this method can be applied to general instances for the LINEAR PROGRAMMING problem, as described below.

Given an instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ of the LINEAR PROGRAMMING problem, by multiplying an equality by $-1$ when necessary, we can always assume that $\mathbf{b} \ge 0$. In order to find a solution to $\alpha$, consider the auxiliary instance $\alpha'$ for the LINEAR PROGRAMMING problem

$$
\begin{array}{ll}
\text{minimize} & y_1 + y_2 + \cdots + y_m \\
\text{subject to} & \mathbf{A}\mathbf{x} + \mathbf{y} = \mathbf{b} \\
& \mathbf{x},\ \mathbf{y} \ge 0
\end{array}
\tag{4.29}
$$

where $\mathbf{y} = (y_1, y_2, \ldots, y_m)$ is an $m$-dimensional vector of *artificial variables*. Note that the $(n+m)$-dimensional vector $\mathbf{w} = (0, 0, \ldots, 0, b_1, b_2, \ldots, b_m)$ is clearly a basic solution to the instance $\alpha'$ in (4.29). If there is a solution $(x_1, \ldots, x_n)$ to the instance $\alpha$, then it is clear that the instance $\alpha'$ in (4.29) has an optimal solution $(x_1, \ldots, x_n, 0, \ldots, 0)$ with optimal objective function value 0. On the other hand, if the instance $\alpha$ has no solution, then the optimal objective function value for the instance $\alpha'$ is larger than 0 (note that the solution set $S_Q(\alpha')$ for the instance $\alpha'$ is always nonempty).

Now starting with the basic solution $\mathbf{w} = (0, 0, \ldots, 0, b_1, b_2, \ldots, b_m)$ for the instance $\alpha'$, we can apply the simplex method to find an optimal solution for $\alpha'$. Note that the tableau for the basic solution $\mathbf{w}$ is also immediate — the first $m$ rows of the tableau have the form $[\mathbf{A}, \mathbf{I}, \mathbf{b}]$, where $\mathbf{I}$ is the $m$-dimensional identity matrix, the reduced cost coefficient $r_j$ for $1 \leq j \leq n$ is equal to $-v_j$, where $v_j$ is the sum of the elements in the $j$th column in the tableau, and the last element in the $(m+1)$st row is equal to $-b_1 - \cdots - b_m$.

Suppose that the simplex method finds an optimal basic solution $\mathbf{w}_0 = (w_1, w_2, \ldots, w_{n+m})$ for the instance $\alpha'$ in (4.29). If $\mathbf{w}_0$ does not have objective function value 0, then the original instance $\alpha$ has no solution. If $\mathbf{w}_0$ has objective function value 0, then we must have $w_j = 0$ for all $n + 1 \leq j \leq n + m$. In the second case, we let $\mathbf{x} = (w_1, w_2, \ldots, w_n)$. We claim that the vector $\mathbf{x}$ is a basic solution for the instance $\alpha$. First of all, it is clear that $\mathbf{x} \geq 0$ and $\mathbf{Ax} = \mathbf{b}$. Moreover, suppose that $w_{i_1}$, $w_{i_2}$, …, $w_{i_k}$ are the positive elements in $\mathbf{x}$, then they are also positive elements in $\mathbf{w}_0$. Thus, the columns $\mathbf{a}_{i_1}$, $\mathbf{a}_{i_2}$, …, $\mathbf{a}_{i_k}$ of the matrix $\mathbf{A}$ are basic columns for $\mathbf{w}_0$ thus are linearly independant. Thus, if we extend the $k$ columns $\mathbf{a}_{i_1}$, $\mathbf{a}_{i_2}$, …, $\mathbf{a}_{i_k}$ of the matrix $\mathbf{A}$ (arbitrarily) into $m$ linearly independent columns of $\mathbf{A}$, then the solution $\mathbf{x}$ is a basic solution with these $m$ linearly independent columns as its basic columns. In case $k = m$, $\mathbf{x}$ is a non-degenerate basic solution, and in case $k < m$, $\mathbf{x}$ is a degenerate basic solution.

To summarize, we use artificial variables to attack a general instance of the LINEAR PROGRAMMING problem. Our approach is a *two-phase method*. This method consists of the first phase in which artificial variables are introduced to construct an auxiliary instance $\alpha'$ with an obvious starting basic solution, and an optimal solution $\mathbf{w}_0$ for $\alpha'$ is constructed using the simplex method; and the second phase in which, a basic solution $\mathbf{x}$ for the original instance $\alpha$ is constructed from the vector $\mathbf{w}_0$ obtained in the first phase, and an optimal solution for $\alpha$ is constructed using the simplex method.

### Putting all these together

We summarize the procedures described so far and formulate them into the complete simplex method. See Figure 4.3. For a given column $q$ in a tableau $\mathcal{T}$ of $m+1$ rows and $n+1$ columns, an element $\mathcal{T}[p, q]$ of $\mathcal{T}$ is said to have the *minimum ratio* in the $q$th column if $\mathcal{T}[p, q] > 0$ and the ratio $\mathcal{T}[p, n+1]/\mathcal{T}[p, q]$ is the minimum over all ratios $\mathcal{T}[j, n+1]/\mathcal{T}[j, q]$ with $1 \leq j \leq n$ and $\mathcal{T}[j, q] > 0$.

Note that in Phase I, step 3, we do not have to check whether the $q$th column of the tableau $\mathcal{T}_1$ has an element with minimum ratio — it must have one. This is because if it does not have one, then by Lemma 4.2.3, the objective function of the instance $\alpha'$ would have had arbitrary small value. On the other hand, 0 is obviously a lower bound for the objective function values for the instance $\alpha'$.

The correctness of the algorithm **Simplex Method** is guaranteed by Lemmas 4.2.1, 4.2.2, and 4.2.3. In particular, under the Nondegeneracy Assumption, each procedure call **TableauMove**$(\mathcal{T}, p, q)$ results in a basic solution with a smaller objective function value. Since the number of basic solutions is finite, and since no basic solution repeats because of the strictly decreasing objective function values, the algorithm **Simplex Method** will eventually stop, either with an optimal solution to the instance $\alpha$, or with a claim that there is no optimal solution to the instance $\alpha$. In the case of degeneracy, incorporated with an anticycling procedure, we can also guarantee that the algorithm **Simplex Method** terminates in a finite number of steps with a correct conclusion.

Moving from one basic solution to a neighbor basic solution, using the tableau format, can be easily done in time $O(nm)$. Thus, the time complexity of the algorithm **Simplex Method** depends on how many basic solution moves are needed to achieve an optimal basic solution. Extensive experience with the simplex method applied to problems from various fields, and having various of the number $n$ of variables and the number $m$ of constraints, has indicated that the method can be expected to converge to an optimal solution in $O(m)$ basic solution moves. Therefore, practically, the algorithm **Simplex Method** is pretty fast. However and unfortunately, there are instances for the LINEAR PROGRAMMING problem for which the algorithm **Simplex Method** requires a large number of basic solution moves. These instances show that the algorithm **Simplex Method** is not a polynomial time algorithm.

## 4.3 Duality

Associated with every instance (**b**, **c**, **A**) of the LINEAR PROGRAMMING problem is a corresponding *dual instance*. Both instances are constructed from the vectors **b** and **c** and the matrix **A** but in such a way that if one of these instances is one of a maximization problem then the other is a minimization problem, and that the optimal objective function values of the instances, if finite, are equal. The variables of the dual instance are

**Algorithm. Simplex Method**

Input: an instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ with $\mathbf{b} \geq 0$

Phase I.

1. construct a new instance $\alpha'$: $\mathbf{Ax} + \mathbf{y} = \mathbf{b}; \quad \mathbf{x},\ \mathbf{y} \geq 0$
2. construct the basic solution $\mathbf{w}$ for the instance $\alpha'$ and the tableau
   $\mathcal{T}_1[1..m+1, 1..m+n+1]$ for $\mathbf{w}$;
3. **while** $\mathcal{T}_1[m+1, q] < 0$ for some $1 \leq q \leq n+m$ **do**
   let $\mathcal{T}_1[p, q]$ have the minimum ratio in column $q$;
   call **TableauMove**$(\mathcal{T}_1, p, q)$;
4. **if** $\mathcal{T}_1[m+1, m+n+1] \neq 0$
   **then** stop: the instance $\alpha$ has no solution;

Phase II.

5. let $\mathcal{T}_2[1..m+1, 1..n+1]$ be $\mathcal{T}_1$ with the $(n+1)$st $\ldots$, $(n+m)$th
   columns deleted;
   { $\mathcal{T}_2$ is the tableau for a basic solution $\mathbf{x}$ for $\alpha$. }
   $\mathcal{T}_2[m+1, n+1] = -\mathbf{c}^T\mathbf{x}$;
6. **while** $\mathcal{T}_2[m+1, q] < 0$ for some $1 \leq q \leq n$ **do**
   **if** no element in the $q$th column of $\mathcal{T}_2$ is positive
   **then** stop: the instance $\alpha$ has no optimal solution
   **else** let $\mathcal{T}_2[p, q]$ have the minimum ratio in column $q$;
   call **TableauMove**$(\mathcal{T}_2, p, q)$;
7. stop: the tableau $\mathcal{T}_2$ gives an optimal solution $\mathbf{x}$ to $\alpha$.

Figure 4.3: The Simplex Method algorithm

also intimately related to the calculation of the reduced cost coefficients in the simplex method. Thus, a study of duality sharpens our understanding of the simplex method and motivates certain alternative solution methods. Indeed, the simultaneous consideration of a problem form both the primal and dual viewpoints often provides significant computational advantage.

### Dual instance

We first depart from our usual strategy of considering instance in the standard form, since the duality relationship is most symmetric for instances expressed solely in terms of inequalities.

Given an instance $\alpha$ of the LINEAR PROGRAMMING problem

$$\begin{array}{ll} & \underline{\text{Primal Instance } \alpha} \\ \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b}, \ \ \mathbf{x} \geq \mathbf{0} \end{array} \tag{4.30}$$

where $\mathbf{A}$ is an $m \times n$ matrix, $\mathbf{c}$ is an $n$-dimensional vector, $\mathbf{b}$ is an $m$-dimensional vector, and $\mathbf{x}$ is an $n$-dimensional vector of variables, the corresponding dual instance $\alpha'$ is of the form

$$\begin{array}{ll} & \underline{\text{Dual Instance } \alpha'} \\ \text{maximize} & \mathbf{y}^T\mathbf{b} \\ \text{subject to} & \mathbf{y}^T\mathbf{A} \leq \mathbf{c}, \ \ \mathbf{y} \geq \mathbf{0} \end{array} \tag{4.31}$$

where $\mathbf{y}$ is an $m$-dimensional vector of variables.

The pair $(\alpha, \alpha')$ of instances is called the *synmmetric form* of duality. We explain below how the symmetric form of duality can be used to define the dual of any instance of the LINEAR PROGRAMMING problem. We first note that the role of primal and dual can be reversed. In fact, if the dual instance $\alpha'$ is transformed, by multiplying the objective function and the constraints by $-1$ so that it has the format of the primal instance in (4.30) (but is still expressed in terms of $\mathbf{y}$), then its corresponding dual will be equivalent to the original instance $\alpha$ in the format given in (4.30).

Consider an instance of the LINEAR PROGRAMMING in the standard form

$$\begin{array}{ll} & \underline{\text{Primal Instance } \alpha} \\ \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0} \end{array} \tag{4.32}$$

Write it in the equivalent form

$$\begin{array}{rc} \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b},\ -\mathbf{A}\mathbf{x} \geq -\mathbf{b},\ \mathbf{x} \geq \mathbf{0} \end{array}$$

which is now in the format of the primal instance in (4.30), with coefficient matrix

$$\begin{bmatrix} \mathbf{A} \\ \text{-}\mathbf{A} \end{bmatrix}$$

Now we will let $\mathbf{z}$ be the $(2m)$-dimensional vector of variables for the dual instance, and write $\mathbf{z}$ as $\mathbf{z}^T = (\mathbf{u},\ \mathbf{v})^T$ where both $\mathbf{u}$ and $\mathbf{v}$ are $m$-dimensional vectors of variables, the corresponding dual instance has the format

$$\begin{array}{rc} \text{maximize} & \mathbf{u}^T\mathbf{b} - \mathbf{v}^T\mathbf{b} \\ \text{subject to} & \mathbf{u}^T\mathbf{A} - \mathbf{v}^T\mathbf{A} \leq \mathbf{c}^T,\ \mathbf{u},\ \mathbf{v} \geq \mathbf{0} \end{array}$$

Letting $\mathbf{y} = \mathbf{u} - \mathbf{v}$ we simplify the representation of the dual problem into the following format

$$\begin{array}{rc} & \underline{\text{Dual Instance } \alpha'} \\ \text{maximize} & \mathbf{y}^T\mathbf{b} \\ \text{subject to} & \mathbf{y}^T\mathbf{A} \leq \mathbf{c}^T \end{array} \tag{4.33}$$

The pair $(\alpha, \alpha')$ in (4.32) and (4.33) gives the *asymmetric form* of the duality relation. In this form the dual vector $\mathbf{y}$ is not restricted to be nonegative.

Similar transformation can be worked out for any instance of the LINEAR PROGRAMMING problem by first converting the primal instance into the format in (4.30), calculating the dual, and then simplifying the dual to account for a special structure.

## The Duality Theorem

So far the relation between a primal instance $\alpha$ and its dual instance $\alpha'$ for the LINEAR PROGRAMMING problem has been simply a formal definition. In the following, we reveal a deeper connection between a primal instance and its dual. This connection will enable us to solve the LINEAR PROGRAMMING problem more efficiently than by simply applying the simplex method.

**Lemma 4.3.1** *Let $\mathbf{x}$ be a solution to the primal instance $\alpha$ in (4.32) and let $\mathbf{y}$ be a solution to the dual instance in (4.33). Then $\mathbf{c}^T\mathbf{x} \geq \mathbf{y}^T\mathbf{b}$.*

PROOF. Since $\mathbf{x}$ is a solution to the instance $\alpha$, we have $\mathbf{y}^T\mathbf{b} = \mathbf{y}^T\mathbf{A}\mathbf{x}$. Now since $\mathbf{y}$ is a solution to the dual instance $\alpha'$, $\mathbf{y}^T\mathbf{A} \leq \mathbf{c}^T$. Note that $\mathbf{x} \geq \mathbf{0}$, thus, $\mathbf{y}^T\mathbf{A}\mathbf{x} \leq \mathbf{c}^T\mathbf{x}$. This gives $\mathbf{c}^T\mathbf{x} \geq \mathbf{y}^T\mathbf{b}$. □

Note that the instance $\alpha$ in (4.32) looks for a minimum value $\mathbf{c}^T\mathbf{x}$ while the instance $\alpha'$ in (4.33) looks for a maximum value $\mathbf{y}^T\mathbf{b}$. Thus, Lemma 4.3.1 shows that a solution to one problem yields a finite bound on the objective function value for the other problem. In particular, this lemma can be used to test whether the primal instance or the dual instance has solution. We say that the primal instance $\alpha$ has an *unbounded* solution if for any negative value $-M$ there is a solution $\mathbf{x}$ to $\alpha$ such that $\mathbf{c}^T\mathbf{x} \leq -M$, and that the dual instance $\alpha'$ has an *unbounded* solution if for any positive value $M$ there is a solution $\mathbf{y}$ to $\alpha'$ such that $\mathbf{y}^T\mathbf{b} \geq M$.

**Theorem 4.3.2** *If the primal instance $\alpha$ in* (4.32) *has an unbounded solution then the dual instance $\alpha'$ in* (4.33) *has no solution, if the dual instance $\alpha'$ has an unbounded solution then the primal instance has no solution.*

PROOF. Suppose that $\alpha$ has an unbounded solution but $\alpha'$ has a solution $\mathbf{y}$. Fix $\mathbf{y}$. Given any negative number $-M$, by defintion, there is a solution $\mathbf{x}$ to $\alpha$ such that $\mathbf{c}^T\mathbf{x} \leq -M$. By Lemma 4.3.1, $\mathbf{c}^T\mathbf{x} \geq \mathbf{y}^T\mathbf{b}$. Thus, $\mathbf{y}^T\mathbf{b} \leq -M$. But this is impossible since $-M$ can be any negative number.

The second statement can be proved similarly. □

If $\mathbf{c}^T\mathbf{x} = \mathbf{y}^T\mathbf{b}$ for a solution $\mathbf{x}$ to $\alpha$ and a solution $\mathbf{y}$ to $\alpha'$, then by Lemma 4.3.1, $\mathbf{x}$ must be an optimal solution for the instance $\alpha$ and $\mathbf{y}$ must be an optimal solution for the instance $\alpha'$. The following theorem indicates that, in fact, this is a necessary and sufficient condition for $\mathbf{x}$ to be an optimal solution for $\alpha$ and for $\mathbf{y}$ to be an optimal solution for $\alpha'$.

**Theorem 4.3.3** *Let $\mathbf{x}$ be a solution to the primal instance $\alpha$ in* (4.32). *Then $\mathbf{x}$ is an optimal solution to $\alpha$ if and only if $\mathbf{c}^T\mathbf{x} = \mathbf{y}^T\mathbf{b}$ for some solution $\mathbf{y}$ to the dual instance $\alpha'$ in* (4.33).

PROOF. Suppose that $\mathbf{c}^T\mathbf{x} = \mathbf{y}^T\mathbf{b}$. By Lemma 4.3.1, for every solution $\mathbf{x}'$ to the primal instance $\alpha$, we have $\mathbf{c}^T\mathbf{x}' \geq \mathbf{y}^T\mathbf{b} = \mathbf{c}^T\mathbf{x}$. Thus, $\mathbf{x}$ is an optimal solution to the primal instance $\alpha$.

Conversely, suppose that $\mathbf{x}$ is an optimal solution to the primal instance $\alpha$. We show that there is a solution $\mathbf{y}$ to the dual instance $\alpha'$ such that $\mathbf{c}^T\mathbf{x} = \mathbf{y}^T\mathbf{b}$.

Since all optimal solutions to the primal instance $\alpha$ give the same objective function value, we can assume, without loss of generality, that the solution $\mathbf{x}$ is a basic solution to the instance $\alpha$. Furthermore, we assume for convenience that $\mathbf{x} = (x_1, x_2, \ldots, x_m, 0, \ldots, 0)^T$, and that the first $m$ columns $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$ of the matrix $\mathbf{A}$ are the basic columns for $\mathbf{x}$. Then the tableau for the basic solution $\mathbf{x}$ is of the following form.

| $\mathbf{a}_1$ | $\mathbf{a}_2$ | $\cdots$ | $\mathbf{a}_m$ | $\mathbf{a}_{m+1}$ | $\mathbf{a}_{m+2}$ | $\cdots$ | $\mathbf{a}_n$ | |
|---|---|---|---|---|---|---|---|---|
| 1 | 0 | $\cdots$ | 0 | $y_{1,m+1}$ | $y_{1,m+2}$ | $\cdots$ | $y_{1n}$ | $x_1$ |
| 0 | 1 | $\cdots$ | 0 | $y_{2,m+1}$ | $y_{2,m+2}$ | $\cdots$ | $y_{2n}$ | $x_2$ |
| · | · | | · | · | · | | · | · |
| 0 | 0 | $\cdots$ | 1 | $y_{m,m+1}$ | $y_{m,m+2}$ | $\cdots$ | $y_{mn}$ | $x_m$ |
| 0 | 0 | $\cdots$ | 0 | $r_{m+1}$ | $r_{m+2}$ | $\cdots$ | $r_n$ | $z_0$ |

where for each $j$, $m + 1 \le j \le n$, we have

$$\mathbf{a}_j = \sum_{i=1}^{m} y_{ij}\mathbf{a}_i \quad \text{and} \quad r_j = c_j - \sum_{i=1}^{m} c_i y_{ij}$$

Since $\mathbf{x}$ is an optimal solution, by Lemmas 4.2.1 and 4.2.2, we must have $r_j \ge 0$ for all $m + 1 \le j \le n$. That is,

$$c_j \ge \sum_{i=1}^{m} c_i y_{ij} \quad \text{for } m + 1 \le j \le n \tag{4.34}$$

Let $\mathbf{B} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m]$ be the $m \times m$ nonsingular submatrix of the matrix $\mathbf{A}$ and let $\mathbf{B}^{-1}$ be the inverse matrix of $\mathbf{B}$. Note that for each $i$, $1 \le i \le m$, $\mathbf{B}^{-1}\mathbf{a}_i$ is the $i$th unit vector of dimension $m$ (i.e., the $m$-dimensional vector whose $i$th element is 1 while all other elements are 0):

$$\mathbf{B}^{-1}\mathbf{a}_i = (0, \ldots, 0, 1, 0, \ldots, 0)^T, \quad i = 1, \ldots, m \tag{4.35}$$

Therefore, for $j = m + 1, \ldots, n$, we have

$$\mathbf{B}^{-1}\mathbf{a}_j = \mathbf{B}^{-1}\sum_{i=1}^{m} y_{ij}\mathbf{a}_i = \sum_{i=1}^{m} y_{ij}\mathbf{B}^{-1}\mathbf{a}_i = (y_{1j}, y_{2j}, \ldots, y_{mj})^T \tag{4.36}$$

The last equality is because of the equality (4.35).

Now we let $\mathbf{y}^T = (c_1, c_2, \ldots, c_m)\mathbf{B}^{-1}$. Then $\mathbf{y}$ is an $m$-dimensional vector. We show that $\mathbf{y}$ is a solution to the dual instance $\alpha'$ and satisfies the condition $\mathbf{c}^T\mathbf{x} = \mathbf{y}^T\mathbf{b}$.

First consider $\mathbf{y}^T\mathbf{A}$. We have

$$\begin{aligned}
\mathbf{y}^T\mathbf{A} &= \mathbf{y}^T[\mathbf{B}, \mathbf{a}_{m+1}, \ldots, \mathbf{a}_n] \\
&= (c_1, \ldots, c_m)\mathbf{B}^{-1}[\mathbf{B}, \mathbf{a}_{m+1}, \ldots, \mathbf{a}_n] \\
&= (c_1, \ldots, c_m)[\mathbf{I}, \mathbf{B}^{-1}\mathbf{a}_{m+1}, \ldots, \mathbf{B}^{-1}\mathbf{a}_n] \\
&= (c_1, \ldots, c_m, c'_{m+1}, \ldots, c'_n)
\end{aligned}$$

where (note the equality (4.36))

$$c'_j = (c_1, \ldots, c_m)\mathbf{B}^{-1}\mathbf{a}_j = (c_1, \ldots, c_m)(y_{1j}, \ldots, y_{mj})^T = \sum_{i=1}^{m} c_i y_{ij}$$

for $j = m+1, \ldots, n$. By the inequality (4.34), we have $c'_j \leq c_j$ for $j = m+1, \ldots, n$. This thus proves $\mathbf{y}^T\mathbf{A} \leq \mathbf{c}^T$. That is, $\mathbf{y}$ is a solution to the dual instance $\alpha'$ in (4.33).

Finally, we have

$$\mathbf{y}^T\mathbf{b} = (c_1, \ldots, c_m)\mathbf{B}^{-1}\mathbf{b} = (c_1, \ldots, c_m)(x_1, \ldots, x_m)^T = \sum_{i=1}^{m} c_i x_i = \mathbf{c}^T\mathbf{x}$$

Therefore, $\mathbf{y}$ is a solution to the dual instance $\alpha'$ that satisfies $\mathbf{c}^T\mathbf{x} = \mathbf{y}^T\mathbf{b}$. This completes the proof of the theorem. $\square$

### The dual simplex method

Suppose that we have an instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ (in the standard form) of the LINEAR PROGRAMMING problem. Often available is a vector $\mathbf{x}$ such that $\mathbf{x}$ satisfies $\mathbf{A}\mathbf{x} = \mathbf{b}$ but not $\mathbf{x} \geq \mathbf{0}$. Moveover, $\mathbf{x}$ "optimizes" the objective function value $\mathbf{c}^T\mathbf{x}$ in the sense that no reduced cost efficient is negative (cf. Lemma 4.2.2). Such a situation may arise, for example, if a solution to a certain instance $\beta = (\mathbf{b}',\ \mathbf{c},\ \mathbf{A})$ of the LINEAR PROGRAMMING problem is calculated and then the instance $\alpha$ is constructed such that $\alpha$ differs from $\beta$ only by the vector $\mathbf{b}$. In such situations a basic solution to the dual instance $\alpha'$ of the instance $\alpha$ is available and hence, based on Theorem 4.3.3, it may be desirable to approach the optimal solution for the instance $\alpha$ in such a way as to optimize the dual instance $\alpha'$.

Rather than constructing a tableau for the dual instance $\alpha'$, it is more efficient to work on the dual instance from the tableau for the primal instance $\alpha$. The complete technique based on this idea is the *dual simplex method.* In terms of the primal instance $\alpha$, it operates by maintaining the optimality condition of the reduced cost coefficients while working toward a solution $\mathbf{x}$ to $\alpha$ that satisfies $\mathbf{x} \geq \mathbf{0}$. In terms of the dual instance $\alpha'$, however, it maintains a solution to $\alpha'$ while working toward optimality.

Formally, let the primal instance $\alpha$ be of the form

$$\begin{array}{ll} & \underline{\text{Primal Instance } \alpha} \\ \text{minimize} & \mathbf{c}^T\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} = \mathbf{b}, \ \mathbf{x} \geq \mathbf{0} \end{array}$$

The dual instance $\alpha'$ of the instance $\alpha$ is of the form

$$\begin{array}{ll} & \underline{\text{Dual Instance } \alpha'} \\ \text{maximize} & \mathbf{y}^T\mathbf{b} \\ \text{subject to} & \mathbf{y}^T\mathbf{A} \leq \mathbf{c}^T \end{array}$$

Suppose that $m$ linearly independent columns of the matrix $\mathbf{A}$ have been identified such that with these $m$ columns as "basic columns", no reduced cost coefficient is negative. Again for convenience of our discussion, suppose these $m$ columns are the first $m$ columns of $\mathbf{A}$ and let $\mathbf{B}$ be the $m \times m$ nonsingular submatrix consisting of these $m$ columns. Therefore, the corresponding tableau should have the form (following our convention, $\mathbf{a}_1, \ldots, \mathbf{a}_n$ denote the columns of the matrix $\mathbf{A}$).

$$\begin{array}{cccccccc|c} \mathbf{a}_1 & \mathbf{a}_2 & \cdots & \mathbf{a}_m & \mathbf{a}_{m+1} & \mathbf{a}_{m+2} & \cdots & \mathbf{a}_n & \\ \hline 1 & 0 & \cdots & 0 & y_{1,m+1} & y_{1,m+2} & \cdots & y_{1n} & x_1 \\ 0 & 1 & \cdots & 0 & y_{2,m+1} & y_{2,m+2} & \cdots & y_{2n} & x_2 \\ \cdot & \cdot & & \cdot & \cdot & \cdot & & \cdot & \cdot \\ 0 & 0 & \cdots & 1 & y_{m,m+1} & y_{m,m+2} & \cdots & y_{mn} & x_m \\ \hline 0 & 0 & \cdots & 0 & r_{m+1} & r_{m+2} & \cdots & r_n & z_0 \end{array}$$

where for each $j$, $m+1 \leq j \leq n$, we have

$$\mathbf{a}_j = \sum_{i=1}^{m} y_{ij}\mathbf{a}_i \quad \text{and} \quad r_j = c_j - \sum_{i=1}^{m} c_i y_{ij} \geq 0 \tag{4.37}$$

and

$$z_0 = -(c_1x_1 + c_2x_2 + \cdots + c_mx_m) \quad \text{and} \quad (x_1, \ldots, x_m)^T = \mathbf{B}^{-1}\mathbf{b} \tag{4.38}$$

We show how we find an optimal solution for the primal instance $\alpha$, starting from this tableau.

If $x_i \geq 0$ for all $1 \leq i \leq m$, then $\mathbf{x} = (x_1, \ldots, x_m, 0, \ldots, 0)$ is an optimal basic solution to the instance $\alpha$ so we are done.

Thus we suppose that there is an $x_p < 0$ for some $1 \leq p \leq m$. Fix this index $p$.

Let $\mathbf{y}^T = (c_1, \ldots, c_m)\mathbf{B}^{-1}$. Then $\mathbf{y}^T\mathbf{A}= (c_1, \ldots, c_m, c'_{m+1}, \ldots, c'_n)$, where $c'_j = \sum_{i=1}^m c_i y_{ij} \leq c_j$, for $j = m+1, \ldots, n$ (see the proof for Theorem 4.3.3). Thus, $\mathbf{y}^T\mathbf{A} \leq \mathbf{c}^T$ and $\mathbf{y}^T$ is a solution to the dual instance $\alpha'$, with the objective function value

$$\mathbf{y}^T\mathbf{b} = (c_1, \ldots, c_m)\mathbf{B}^{-1}\mathbf{b} = (c_1, \ldots, c_m)(x_1, \ldots, x_m)^T = -z_0$$

Thus, it is proper to call the vector $\mathbf{x}= (x_1, \ldots, x_m, 0, \ldots, 0)$ the *dual basic solution* to the instance $\alpha$ (distinguish it from the a basic solution to the dual instance $\alpha'$), and the columns $\mathbf{a}_1, \ldots, \mathbf{a}_m$ the *dual basic columns*.

Our intention is to replace a dual basic column by a new column so that the dual basic solution corresponds to an improved solution to the dual instance $\alpha'$.

Note that $\mathbf{B}^{-1}\mathbf{a}_i = (0, \ldots, 0, 1, 0, \ldots, 0)^T$ is the $i$th unit vector of dimension $m$ for $i = 1, \ldots, m$ (see equality (4.35)), and $\mathbf{B}^{-1}\mathbf{a}_j = (y_{1j}, y_{2j}, \ldots, y_{mj})^T$ for $j = m+1, \ldots, n$ (see equality (4.36)). Therefore, if we let $\mathbf{u}_p$ be the row vector given by the $p$th row of the matrix $\mathbf{B}^{-1}$, we will have

$$\mathbf{u}_p\mathbf{a}_i = \begin{cases} 0 & \text{if } i \neq p \\ 1 & \text{if } i = p \end{cases} \qquad \text{for } i = 1, \ldots, m \tag{4.39}$$

and

$$\mathbf{u}_p\mathbf{a}_j = y_{pj} \quad \text{for } j = m+1, \ldots, n \tag{4.40}$$

Let $\mathbf{y}_\epsilon^T = \mathbf{y}^T - \epsilon\mathbf{u}_p$. We show that with properly selected $\epsilon > 0$, $\mathbf{y}_\epsilon$ is an improved solution to the dual instance $\alpha'$.

First consider

$$\begin{aligned} \mathbf{y}_\epsilon^T\mathbf{A} &= (\mathbf{y}^T - \epsilon\mathbf{u}_p)\mathbf{A} = \mathbf{y}^T\mathbf{A} - \epsilon\mathbf{u}_p\mathbf{A} \\ &= \mathbf{y}^T\mathbf{A} - \epsilon\mathbf{u}_p[\mathbf{a}_1, \ldots, \mathbf{a}_n] \\ &= \mathbf{y}^T\mathbf{A} - \epsilon[\mathbf{u}_p\mathbf{a}_1, \ldots, \mathbf{u}_p\mathbf{a}_m, \mathbf{u}_p\mathbf{a}_{m+1}, \ldots, \mathbf{u}_p\mathbf{a}_n] \\ &= (c_1, \ldots, c_m, c'_{m+1}, \ldots, c'_n) - \epsilon(0, \ldots, 0, 1, 0, \ldots, 0, y_{p,m+1}, \ldots, y_{p,n}) \\ &= (c_1, \ldots, c_{p-1}, c_p - \epsilon, c_{p+1}, \ldots, c_m, c'_{m+1} - \epsilon y_{p,m+1}, \ldots, c'_n - \epsilon y_{pn}) \end{aligned} \tag{4.41}$$

The fifth equality is from the equalities (4.39) and (4.40).

If all $y_{pj} > 0$ for $j = m+1, \ldots, n$, then $\mathbf{y}_\epsilon$ is a solution to the dual instance $\alpha'$ for any $\epsilon \geq 0$ with the objective function value

$$\mathbf{y}_\epsilon^T\mathbf{b} = (\mathbf{y}^T - \epsilon\mathbf{u}_p)\mathbf{b} = \mathbf{y}^T\mathbf{b} - \epsilon\mathbf{u}_p\mathbf{b} = \mathbf{y}^T\mathbf{b} - \epsilon x_p$$

**Algorithm. Dual Simplex Method**

Input: an instance $\alpha$ with a tableau $\mathcal{T}$ for a dual basic solution $\mathbf{x}$ to $\alpha$

1. **while** $\mathcal{T}[p, n+1] < 0$ for some $1 \le p \le m$ **do**
   **if** no $\mathcal{T}[p, j] < 0$ for any $1 \le j \le n$
   **then** stop: the instance $\alpha$ has no solution
   **else** let $\mathcal{T}[p, q]$ have the minimum ratio in row $p$;
   call **TableauMove**$(\mathcal{T}, p, q)$;
2. stop: the tableau $\mathcal{T}$ gives an optimal solution $\mathbf{x}$ to $\alpha$.

Figure 4.4: The Dual Simplex Method algorithm

(note $\mathbf{B}^{-1}\mathbf{b} = (x_1, \ldots, x_m)^T$ thus $\mathbf{u}_p\mathbf{b} = x_p$). Since $x_p < 0$, $\mathbf{y}_\epsilon^T\mathbf{b}$ can be arbitrarily large. That is, the solution to the dual instance $\alpha'$ is unbounded. By Theorem 4.3.2, the primal instance $\alpha$ has no solution. Thus, again, we are done.

So we assume that $y_{pq} < 0$ for some $q$, $m+1 \le q \le n$. Select the index $q$ such that $c'_q - \epsilon y_{pq}$ is the first that meets $c_q$ when $\epsilon$ increases from 0. Therefore, the index $q$ should be chosen as follows.

$$\epsilon_0 = -\frac{r_q}{y_{pq}} = \min_{m+1 \le j \le n} \{-\frac{r_j}{y_{pj}} | y_{pj} < 0\}$$

(note $y_{pq} < 0$ and $r_q \ge 0$ so $\epsilon_0 \ge 0$). We verify that $\mathbf{y}_0^T = \mathbf{y}^T - \epsilon_0 \mathbf{u}_p$ is a solution to the dual instance $\alpha'$, i.e., $\mathbf{y}_0^T\mathbf{A} \le \mathbf{c}^T$.

Consider the equality (4.41). Since $\epsilon_0 \ge 0$, we have $c_p - \epsilon_0 \le c_p$. Moreover, for each $c'_j - \epsilon_0 y_{pj}$ with $j = m+1, \ldots, n$, if $y_{pj} \ge 0$, then of course $c'_j - \epsilon_0 y_{pj} \le c'_j \le c_j$; while for $y_{pj} < 0$, by our choice of $q$ we have

$$-\frac{r_q}{y_{pq}} \le -\frac{r_j}{y_{pj}} \quad \text{or} \quad \frac{r_q}{y_{pq}} \ge \frac{r_j}{y_{pj}}$$

and we have (note $y_{pj} < 0$)

$$c'_j - \epsilon_0 y_{pj} = c'_j + \frac{r_q}{y_{pq}} y_{pj} \le c'_j + \frac{r_j}{y_{pj}} y_{pj} = c'_j + r_j = c_j$$

This proves that $\mathbf{y}_0^T\mathbf{A} \le \mathbf{c}^T$ and $\mathbf{y}_0$ is a solution to the dual instance $\alpha'$.

We evaluate the objective function value for $\mathbf{y}_0$:

$$\mathbf{y}_0^T\mathbf{b} = (\mathbf{y}^T - \epsilon_0 \mathbf{u}_p)\mathbf{b} = \mathbf{y}^T\mathbf{b} - \epsilon_0 \mathbf{u}_p \mathbf{b} = \mathbf{y}^T\mathbf{b} - \epsilon_0 x_p$$

Since $x_p < 0$, we have $\mathbf{y}_0^T\mathbf{b} \geq \mathbf{y}^T\mathbf{b}$. In particular, if $\epsilon_0 > 0$, then $\mathbf{y}_0$ is an improvement over the solution $\mathbf{y}$ for the dual instance $\alpha'$. The case $\epsilon_0 = 0$, i.e., $r_q = 0$, is the *degenerate situation* for the dual simplex method. As we have discussed for the regular simplex method, the dual simplex method in general works fine with degenerate situations, and special techniques can be adopted to handle degenerate situations.

Therefore, the above procedure illustrates how we obtain an improved solution for the dual instance $\alpha'$ by replacing a dual basic column $\mathbf{a}_p$ by a new column $\mathbf{a}_q$. Note that this replacement is not done based on the tableau for the solution $\mathbf{y}$ to the dual instance $\alpha'$. Instead, it is accomplished based on the tableau for the dual basic solution $\mathbf{x}$ for the primal instance $\alpha$. This replacement can be simply done by calling the algorithm **TableauMove**$(\mathcal{T}, p, q)$ in Figure 4.2 when the indices $p$ and $q$ are decided. We summarize this method in Figure 4.4, where we say that an element $\mathcal{T}[p, q]$ in the $p$th row in $\mathcal{T}$ has the *minimum ratio* if the ratio $-r_q/y_{pq}$ is the minimum over all $-r_j/y_{pj}$ with $y_{pj} < 0$.

**Example 4.3.1.** Consider the following instance $\alpha$ for the LINEAR PROGRAMMING problem.

$$
\begin{array}{rc}
\text{minimize} & 3x_1 + 4x_2 + 5x_3 \\
\text{subject to} & -x_1 - 2x_2 - 3x_3 + x_4 = -5 \\
& -2x_1 - 2x_2 - x_3 + x_5 = -6 \\
& x_1, x_2, x_3, x_4, x_5 \geq 0
\end{array}
$$

The dual basic solution $\mathbf{x} = (0, 0, 0, -5, -6)^T$ to $\alpha$ has the tableau

| $\mathbf{a}_1$ | $\mathbf{a}_2$ | $\mathbf{a}_3$ | $\mathbf{a}_4$ | $\mathbf{a}_5$ | |
|---|---|---|---|---|---|
| $-1$ | $-2$ | $-3$ | 1 | 0 | $-5$ |
| $\boxed{-2}$ | $-2$ | $-1$ | 0 | 1 | $-6$ |
| 3 | 4 | 5 | 0 | 0 | 0 |

Pick $x_5 = -6$. To find a proper element in the second row, we compute the ratios $-r_q/y_{2q}$ and select the one with the minimum ratio. This makes us to pick $y_{21} = -2$ (as indicated by the box). Applying the algorithm **TableauMove**$(\mathcal{T}, 2, 1)$ gives us

| $\mathbf{a}_1$ | $\mathbf{a}_2$ | $\mathbf{a}_3$ | $\mathbf{a}_4$ | $\mathbf{a}_5$ | |
|---|---|---|---|---|---|
| 0 | $\boxed{-1}$ | $-5/2$ | 1 | $-1/2$ | $-2$ |
| 1 | 1 | $1/2$ | 0 | $-1/2$ | 3 |
| 0 | 1 | $7/2$ | 0 | $3/2$ | $-9$ |

Now pick $x_4 = -2$ and choose the element $y_{12} = -1$ (as indicated in the box). The algorithm **Tableau**$(\mathcal{T}, 1, 2)$ results in

| $\mathbf{a}_1$ | $\mathbf{a}_2$ | $\mathbf{a}_3$ | $\mathbf{a}_4$ | $\mathbf{a}_5$ | |
|---|---|---|---|---|---|
| 0 | 1 | 5/2 | −1 | 1/2 | 2 |
| 1 | 0 | −2 | 1 | −1 | 1 |
| 0 | 0 | 1 | 1 | 1 | −11 |

The last tableau yields a dual basic solution $\mathbf{x}_0 = (1, 2, 0, 0, 0)^T$ with $\mathbf{x}_0 \geq 0$. Thus it must be an optimal solution to the instance $\alpha$. The objective fucntion value on $\mathbf{x}_0$ is 11.

## 4.4 Polynomial time algorithms

It was an outstanding open problem whether the LINEAR PROGRAMMING problem could be solved in polynomial time, until the spring of 1979, the Russian mathematician L. G. Khachian published a proof that an algorithm, called *the Ellipsoid Algorithm*, solves the LINEAR PROGRAMMING problem in polynomial time [81]. Despite the great theoretical value of the Ellipsoid Algorithm, it is not clear at all that this algorithm can be practically useful. The most obvious among many obstacles is the large precision apparently required.

Another polynomial time algorithm for the LINEAR PROGRAMMING problem, called the *Projective Algorithm*, or more generally, the *Interior Point Algorithm*, was published by N. Karmarkar in 1984 [76]. The Projective Algorithm, and its derivatives, have great impact in the study of the LINEAR PROGRAMMING problem.

# Chapter 5

# Which Problems Are Not Tractable?

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

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

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

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

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

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

## 5.1 NP-hard optimization problems

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

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

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

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

Recall that the PARTITION problem is defined as follows.

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

and given a positive integer $c$, the $c$-MAKESPAN problem is defined by

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

that is, the $c$-MAKESPAN problem is the MAKESPAN problem in which the number of processors is a fixed constant $c$.

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

PROOF. The polynomial time computable functions $h$ and $g$ are described as follows.

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

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

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

It is easy to see that if $\alpha$ is a yes-instance for the PARTITION problem, then every optimal scheduling $P$ on $h(\alpha)$ splits the numbers $x_1, \ldots, x_n$ into two sets $S_1$ and $S_2$ of equal size $(\sum_{i=1}^n x_i)/2$, assigns each of the sets to a processor, and assigns each of the jobs of time $t_j$, $j = n+1, \ldots, n+c-2$, to a distinct processor. The scheduling $P$ has parallel completion time $(\sum_{i=1}^n x_i)/2 = \lceil (\sum_{i=1}^n x_i)/2 \rceil$. On the other hand, if $\alpha$ is a no-instance for the PARTITION problem or $P$ is not an optimal scheduling for $h(\alpha)$, then the scheduling $P$ on $h(\alpha)$ always has parallel completion time larger than $(\sum_{i=1}^n x_i)/2$. In particular, if $\sum_{i=1}^n x_i$ is an odd number, then any scheduling on $h(\alpha)$ has parallel completion time at least $\lceil (\sum_{i=1}^n x_i)/2 \rceil > (\sum_{i=1}^n x_i)/2$.

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

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

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

PROOF. Let $h$ and $g$ be the two polynomial time computable functions for the reduction from $D$ to $Q$. Let $\mathcal{A}$ be a polynomial time algorithm that solves the optimization problem $Q$. Now a polynomial time algorithm for the decision problem $D$ can be easily derived as follows: given an instance $x$ for $D$, we first construct the instance $h(x)$ for $Q$; then apply the algorithm $\mathcal{A}$ to find an optimal solution $y$ for $h(x)$; now $x$ is a yes-instance for $D$ if and only if $g(x, h(x), y) = 1$. By our assumption, all $h(x)$, $y$, and $g(x, h(x), y)$ are polynomial time computable (in particular note that since $h(x)$ is computable in polynomial time , the length $|h(x)|$ of $h(x)$ is bounded by a polynomial of $|x|$, and that since $\mathcal{A}$ runs in polynomial time, the length $|y|$ of $y$ is bounded by a polynomial of $|h(x)|$ thus by a polynomial of $|x|$). Thus, this algorithm runs in polynomial time and correctly decides if $x$ is a yes-instance for the decision problem $D$. $\square$

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

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

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

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

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

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

We give another example for NP-hard optimization problems, whose NP-hardness is from a not so obvious polynomial time reduction. Suppose that in the LINEAR PROGRAMMING problem, we require that we work only on the domain of integer numbers, then we get the INTEGER LINEAR PROGRAMMING problem, or for short the INTEGER LP problem. More formally, each instance of the INTEGER LP problem is a triple $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$, where for some integers $n$ and $m$, $\mathbf{b}$ is an $m$-dimensional vector of integer numbers, $\mathbf{c}$ is an $n$-dimensional vector of integer numbers, and $\mathbf{A}$ is an $m \times n$ matrix of integer numbers. A solution $\mathbf{x}$ to the instance $\alpha$ is an $n$-dimensional vector of integer numbers such that $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$, and a solution $\mathbf{x}$ is optimal if it minimizes the inner product $\mathbf{c}^T\mathbf{x}$. This gives the *standard form* for the INTEGER LP problem. We can similarly define the *general form* for the INTEGER LP problem, which involves more general inequalities such as the form given in (4.1). Moreover, it is not hard to verify that the translation rules described in Section 4.1 can be used to convert an instance in the general form for the INTEGER LP problem into an instance in the standard form for the INTEGER LP problem.

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

The hardness of the INTEGER LP problem is formally given by the following theorem.

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

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

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

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

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

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

Suppose that the clause $C_i$ in $\alpha$ is

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

We then construct a linear constraint

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

As we have seen in Section 4.4 , the general LINEAR PROGRAMMING problem can be solved in polynomial time. Theorem 5.1.4 shows that the INTEGER LP problem is much harder than the general LINEAR PROGRAMMING problem. Our later study will show that the INTEGER LP problem is actually one of the hardest NP-optimization problems.

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

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

(1) for any instance $x_1$ of $Q_1$, $\chi(x_1)$ is an instance of $Q_2$; and

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

The following theorem follows directly from the definition.

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

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

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

PROOF. Suppose that $Q_1$ is p-reducible to $Q_2$ via the instance function $\chi$ and the solution function $\psi$, both computable in polynomial time. Since $Q_1$ is NP-hard, there is an NP-hard decision problem $D$ that is polynomial time reducible to $Q_1$. Suppose that the decision problem $D$ is polynomial time reducible to $Q_1$ via two polynomial time computable functions $h$ and $g$ (see Definition 5.1.1). Define two new functions $h_1$ and $g_1$ as follows: for any instance $x$ of $D$, $h_1(x) = \chi(h(x))$; and for any solution $y$ to $h_1(x)$, $g_1(x, h_1(x), y) = g(x, h(x), \psi(h(x), h_1(x), y))$. It is not hard to verify by the definitions that for any instance $x$ of $D$, $h_1(x)$ is an instance of $Q_2$, and $g_1(x, h_1(x), y) = 1$ if and only if $y$ is an optimal solution to $Q_2$ and $x$ is a yes-instance of $D$. Moreover, the functions $h_1$ and $g_1$ are clearly polynomial time computable.

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

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

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

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

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

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

Given an instance $\alpha = \langle t_1, \ldots, t_n \rangle$ for the 2-MAKESPAN problem, $\chi(\alpha)$ is the instance $\chi(\alpha) = \langle t_1, \ldots, t_n; t_1, \ldots, t_n; B \rangle$ for the KNAPSACK problem, where $B = \lceil \sum_{i=1}^{n} t_i/2 \rceil$. Given any solution $S$ to $\chi(\alpha)$, which is a subset of $\{t_1, \ldots, t_n\}$ satisfying $\sum_{t_j \in S} t_j \leq B$, the value of $\psi(\alpha, \chi(\alpha), S)$ is the partition $(S, \{t_1, \ldots, t_n\} - S)$ of the set $\{t_1, \ldots, t_n\}$, which assigns all the jobs in $S$ to Processor-1, and all other jobs to Processor-2. Since an optimal solution $S$ to $\chi(\alpha)$ is a subset of $\{t_1, \ldots, t_n\}$ that maximizes the value $\sum_{t_j \in S} t_j$ subject to the constraint $\sum_{t_j \in S} t_j \leq \lceil \sum_{i=1}^{n} t_i/2 \rceil$, the solution $S$ must give the "most even" splitting $(S, \{t_1, \ldots, t_n\} - S)$ for the set $\{t_1, \ldots, t_n\}$. Therefore, $S$ is an optimal solution to the instance $\chi(\alpha)$ of the KNAPSACK problem if and only if $(S, \{t_1, \ldots, t_n\} - S)$ is an optimal solution to the instance $\alpha$ of the 2-MAKESPAN problem. Moreover, the instance function $\chi$ and the solution function $\psi$ are clearly computable in polynomial time. This completes the proof. $\square$

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

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

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

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

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

For example, consider the PLANAR GRAPH INDEP-SET problem (given a planar graph $G$, find the largest subset $S$ of vertices in $G$ such that no two vertices in $S$ are adjacent) and the INDEPENDENT SET problem (given a graph $G$, find the largest subset $S$ of vertices in $G$ such that no two vertices in $S$ are adjacent). Clearly the PLANAR GRAPH INDEP-SET problem is a subproblem of the INDEPENDENT SET problem. Since we have known that the PLANAR GRAPH INDEP-SET problem is NP-hard (see the remark following Theorem 5.1.3), we conclude that the INDEPENDENT SET problem is also NP-hard. Similarly, from the NP-hardness of the PLANAR GRAPH VERTEX-COVER problem (given a planar graph $G$, find a minimum set $S$ of vertices such that every edge in $G$ has at least one end in $S$), we derive the NP-hardness for the VERTEX COVER problem (given a graph $G$, find a minimum set $S$ of vertices such that every edge in $G$ has at least one end in $S$).

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

## 5.2 Integer linear programming is NPO

Recall that an optimization problem $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ is an *NP optimization* problem (or shortly NPO) 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 $p(|x|)$.

Traditionally, the word "NP" is used to indicate "the ability of guessing polynomially many bits in polynomial time". This fact is also correctly reflected in the definition of an NPO problem $Q$: if you can always guess polynomially many bits correctly (that constitutes a shortest optimal solution), then you can solve the problem $Q$ in polynomial time.

Note that the NP-hardness, as we studied in the last section, does not necessarily imply the NPO membership. Roughly speaking, the NP-hardness of an optimization problem $Q$ gives a lower bound on the computational difficulty for the problem (i.e., how hard the problem $Q$ is): solving $Q$ is at least as hard as solving the NP-complete problem SATISFIABILITY; while the NPO membership for an optimization problem $Q$ gives an upper bound on the computational difficulty for the problem (i.e., how easy the problem $Q$ is): you can solve the problem $Q$ in polynomial time *if* you can guess correctly.

The NPO membership for most NP optimization problems is obvious and straightforward. In particular, if an optimization problem $Q$ is a "subset problem" given in the form "given a set $S$ of elements with certain relations, find the 'best' subset of $S$ that satisfies certain properties," then the problem $Q$ is an NPO problem. A large number of optimization problems, such as MINIMUM SPANNING TREE, GRAPH MATCHING, TRAVELING SALESMAN, KNAPSACK, INDEPENDENT SET problems, belong to this category.

However, there are also NP-hard optimization problems, for which the NPO-membership is not so straightforward. The problem INTEGER LP is a well-known optimization problem belonging to this category. By Theorem 5.1.4, the INTEGER LP problem is NP-hard.

Given an instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ for the INTEGER LP problem, a solution $\mathbf{x}$ to $\alpha$ is an integer vector satisfying $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$. To show that the INTEGER LP problem is NPO, we must show that there is a polynomial $p(n)$ such that for any instance $\alpha$, there is an optimal solution $\mathbf{x}$ for $\alpha$ such that the length of $\mathbf{x}$ is bounded by $p(|\alpha|)$.

It is reasonable to suppose that the length $|\alpha|$ of the instance $\alpha$ is of order $\Theta(nm\lambda_{\max})$ but at least $nm + \lambda_{\max}$, where we assume that $\mathbf{A}$ is an $m \times n$ matrix and $\lambda_{\max}$ is the largest number of digits needed to represent an element appearing in the matrix $\mathbf{A}$ and in the vectors $\mathbf{b}$ and $\mathbf{c}$. Therefore, to prove that the INTEGER LP problem is NPO, we must show that for any instance $\alpha$ to the INTEGER LP problem, there is a "small" optimal solution $\mathbf{x}$ for $\alpha$ such that the number of digits of each element in $\mathbf{x}$ is bounded by a polynomial of $n$, $m$, and $\lambda_{\max}$. This, indeed, is the case, as we will discuss in the rest of this section.

Throughout this section, we will assume that $\mathbf{A}= [a_{ij}]$ is an $m \times n$ integer matrix with the column vectors $\mathbf{a}_1, \ldots, \mathbf{a}_n$, $\mathbf{b}= (b_i)$ is an $m$-dimensional integer vector, and $\mathbf{c}= (c_j)$ is an $n$-dimensional integer vector. For a real number $r$, we denote by $|r|$ the abslute value of $r$. For a matrix $\mathbf{A}$, we denote by $\|\mathbf{A}\|$ the largest $|a_{ij}|$ over all elements $a_{ij}$ in $\mathbf{A}$. Similarly we define $\|\mathbf{b}\|$ and $\|\mathbf{c}\|$. To make our discussion interesting, we can assume that $\|\mathbf{A}\| \geq 1$

and $\|\mathbf{c}\| \geq 1$.

We first regard $\mathbf{Ax} = \mathbf{b}$ as a linear system in the domain of real numbers, and discuss the properties of solutions to $\mathbf{Ax} = \mathbf{b}$ in the domain of real numbers. The following lemma indicates that if the linear system $\mathbf{Ax} = \mathbf{b}$ is solvable, then it must have a relatively "simple" rational solution.

**Lemma 5.2.1** *If the linear system* $\mathbf{Ax} = \mathbf{b}$ *has a solution, then it has a solution* $\mathbf{x} = (x_1, \ldots, x_n)^T$ *in which each element* $x_i$ *can be expressed as a quotient* $r_i/r$ *of two integers* $r_i$ *and* $r$ *(*$r$ *is common to all* $x_i$*) such that* $0 \leq |r_i| \leq m^m \|\mathbf{A}\|^{m-1}\|\mathbf{b}\|$*, and* $0 < |r| \leq (m\|\mathbf{A}\|)^m$.

PROOF. Without loss of generality, assume that the first $k$ columns $\mathbf{a}_1$, …, $\mathbf{a}_k$ of the matrix $\mathbf{A}$ are linearly independent and any $k+1$ columns of $\mathbf{A}$ are linearly dependent. Then the equation

$$x_1\mathbf{a}_1 + x_2\mathbf{a}_2 + \ldots + x_k\mathbf{a}_k = \mathbf{b}$$

has a unique solution $(x_1^0, \ldots, x_k^0)$. By Cramer's Rule (see Appendix C), this unique solution can be obtained from a $k \times k$ nonsingular submatrix $\mathbf{B}$ of $\mathbf{A}$ and a $k$-dimensional subvector $\mathbf{b}'$ of $\mathbf{b}$ such that for $i = 1, \ldots, k$,

$$x_i^0 = det(\mathbf{B}_i)/det(\mathbf{B})$$

where $\mathbf{B}_i$ is the matrix $\mathbf{B}$ with the $i$th column replaced by the vector $\mathbf{b}'$, and $det(\mathbf{B}_i)$ and $det(\mathbf{B})$ denote the determinants of the matrices $\mathbf{B}_i$ and $\mathbf{B}$, respectively. By the definition of a determinant (see Appendix C), we have

$$|det(\mathbf{B}_i)| \leq k!\|\mathbf{B}_i\|^{k-1}\|\mathbf{b}'\| \leq m^m\|\mathbf{A}\|^{m-1}\|\mathbf{b}\|$$

and

$$|det(\mathbf{B})| \leq k!\|\mathbf{B}\|^k \leq m!\|\mathbf{A}\|^m \leq (m\|\mathbf{A}\|)^m$$

where we have used facts $k \leq m$ and $m! \leq m^m$. Now if we let $r = det(\mathbf{B})$, $r_i = det(\mathbf{B}_i)$ for $i = 1, \ldots, k$, and $r_j = 0$ for $j = k+1, \ldots, n$, then the $n$-dimensional vector $\mathbf{x}_0 = (r_1/r, r_2/r, \ldots, r_n/r)$ is a solution to the linear system $\mathbf{Ax} = \mathbf{b}$. □

**Corollary 5.2.2** *Let* $\alpha = $ **(b, c, A)** *be an instance of the* LINEAR PROGRAMMING *problem. Then for any basic solution* $\mathbf{x}_0 = (x_1^0, \ldots, x_n^0)^T$ *to* $\alpha$*, we can express each* $x_i^0$ *as a quotient* $r_i/r$ *of two integers* $r_i$ *and* $r$ *such that* $0 \leq r_i \leq m^m\|\mathbf{A}\|^{m-1}\|\mathbf{b}\|$*, and* $0 < r \leq (m\|\mathbf{A}\|)^m$.

PROOF. Suppose without loss of generality that $\mathbf{a}_1$, …, $\mathbf{a}_k$ are the basic columns for the basic solution $\mathbf{x}_0$. Then the linear system $\mathbf{A}'\mathbf{x}' = \mathbf{b}$, where $\mathbf{A}' = [\mathbf{a}_1, \ldots, \mathbf{a}_k]$ is an $m \times k$ matrix and $\mathbf{x}' = (x_1, \ldots, x_k)^T$ is a $k$-dimensional vector of unknown variables, has a unique solution $(x_1^0, \ldots, x_k^0)^T$. By Lemma 5.2.1, each $x_i^0$, $i = 1, \ldots, k$, can be expressed as a quotient $r_i/r$ of two integers $r_i$ and $r$ such that $|r_i| \le m^m \|\mathbf{A}'\|^{m-1}\|\mathbf{b}\| \le m^m\|\mathbf{A}\|^{m-1}\|\mathbf{b}\|$, and $0 < |r| \le (m\|\mathbf{A}'\|)^m \le (m\|\mathbf{A}\|)^m$. Since $x_i \ge 0$, we can assume both $r_i$ and $r$ are non-negative. Now the corollary follows since for $j = k+1, \ldots, n$, we can simply let $r_j = 0$. $\square$

Now we move to study integer solutions for linear systems.

**Lemma 5.2.3** *If the linear system* $\mathbf{Ax} = \mathbf{b}$, $\mathbf{x} \ge \mathbf{0}$ *has a (real or integer) solution* $\mathbf{x}' = (x_1', x_2', \ldots, x_n')^T$ *with* $x_1' \ge 1$, *then there is an integer* $r$, $1 \le r \le ((m+1)\|\mathbf{A}\|)^{m+1}$, *such that the linear system* $\mathbf{Ax} = r\mathbf{b}$, $\mathbf{x} \ge \mathbf{0}$ *has an integer solution* $\mathbf{x}_0 = (x_1^0, x_2^0, \ldots, x_n^0)^T$ *with* $x_1^0 \ge 1$ *and* $\|\mathbf{x}_0\| \le (m+1)^{m+1}\|\mathbf{A}\|^m\|\mathbf{b}\|_*$, *where* $\|\mathbf{b}\|_* = \max\{\|\mathbf{b}\|, 1\}$ .

PROOF. Consider the linear system

$$\mathbf{A}'\mathbf{z} = \mathbf{b}', \qquad \mathbf{z} \ge \mathbf{0} \tag{5.4}$$

where

$$\mathbf{A}' = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} & 0 \\ a_{21} & a_{22} & \cdots & a_{2n} & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{m1} & a_{m2} & \ldots & a_{mn} & 0 \\ 1 & 0 & \ldots & 0 & -1 \end{pmatrix}, \quad \mathbf{z} = \begin{pmatrix} z_1 \\ z_2 \\ \cdot \\ z_n \\ z_{n+1} \end{pmatrix}, \quad \mathbf{b}' = \begin{pmatrix} b_1 \\ b_2 \\ \cdot \\ b_n \\ 1 \end{pmatrix}$$

The linear system in (5.4) has a solution $\mathbf{z} = (x_1', \ldots, x_n', x_1' - 1)$. Thus, by Theorem 4.1.1, it has a basic solution $\mathbf{z}_0 = (z_1^0, \ldots, z_{n+1}^0)$. By Corollary 5.2.2, each $z_i^0$ can be expressed as a quotient $r_i/r$ of two integers $r_i$ and $r$ such that (note that $\mathbf{A}'$ is an $(m+1) \times (n+1)$ matrix and $\|\mathbf{b}'\| = \|\mathbf{b}\|_*$) $0 \le r_i \le (m+1)^{m+1}\|\mathbf{A}'\|^m\|\mathbf{b}'\| = (m+1)^{m+1}\|\mathbf{A}\|^m\|\mathbf{b}\|_*$, and $0 < r \le ((m+1)\|\mathbf{A}'\|)^{m+1} = ((m+1)\|\mathbf{A}\|)^{m+1}$.

Now multiplying both sides of the equality $\mathbf{A}'\mathbf{z}_0 = \mathbf{b}'$ by $r$, we get $\mathbf{A}'(r\mathbf{z}_0) = r\mathbf{b}'$. That is, the integer vector $r\mathbf{z}_0 = (rz_1^0, \ldots, rz_{n+1}^0)^T$ is a solution to the linear system $\mathbf{A}'\mathbf{z} = r\mathbf{b}'$, $\mathbf{z} \ge \mathbf{0}$, which implies immediately that the $n$-dimensional integer vector $\mathbf{x}_0 = (rz_1^0, \ldots, rz_n^0)^T$ is a solution to the linear system $\mathbf{Ax} = r\mathbf{b}$, $\mathbf{x} \ge \mathbf{0}$. Moreover, from $\mathbf{A}'\mathbf{z}_0 = r\mathbf{b}'$, $\mathbf{z}_0 \ge \mathbf{0}$, and $r > 0$, we have $rz_1^0 - rz_{n+1}^0 = 1$, thus $rz_1^0 \ge 1$, and for each $i = 1, \ldots, n$

$$0 \le rz_i^0 = r(r_i/r) = r_i \le (m+1)^{m+1}\|\mathbf{A}\|^m\|\mathbf{b}\|_*$$

Thus, $\|\mathbf{x}_0\| \leq (m+1)^{m+1}\|\mathbf{A}\|^m\|\mathbf{b}\|_*$. $\square$

**Corollary 5.2.4** *If $\mathbf{y}^T\mathbf{A} \geq (1, \ldots, 1)$ has a (real or integer) solution, then it has an integer solution $\mathbf{y}_0$ with $\|\mathbf{y}_0\| \leq 2(n+1)^{n+1}\|\mathbf{A}\|^n$.*

PROOF. Write $\mathbf{y}^T\mathbf{A} \geq (1, \ldots, 1)$ in column form $\mathbf{A}^T\mathbf{y} \geq (1, \ldots, 1)^T$ then convert it into the standard form

$$\mathbf{A}^T\mathbf{u} - \mathbf{A}^T\mathbf{v} - \mathbf{z} = (1, \ldots, 1)^T, \quad \mathbf{u}, \mathbf{v}, \mathbf{z} \geq \mathbf{0} \tag{5.5}$$

where $\mathbf{z} = (z_1, \ldots, z_m)^T$ is the vector of surplus variables, $\mathbf{u} = (u_1, \ldots, u_m)^T$, $\mathbf{v} = (v_1, \ldots, v_m)^T$, and $\mathbf{y} = \mathbf{u} - \mathbf{v}$. Since the linear system $\mathbf{y}^T\mathbf{A} \geq [1, \ldots, 1]$ has a solution $\mathbf{y}' = (y_1', \ldots, y_m')$, the linear system in (5.5) has a solution $\mathbf{w}= (u_1', \ldots, u_m', v_1', \ldots, v_m', z_1', \ldots, z_m')^T$. Moreover, we can assume $u_1' \geq 1$ in $\mathbf{w}$ since the only constraints for $u_1'$ are $y_1' = u_1' - v_1'$ and $u_1' \geq 0$.

By Lemma 5.2.3, there is an integer $r > 0$ such that the linear system $\mathbf{A}^T\mathbf{u} - \mathbf{A}^T\mathbf{v} - \mathbf{z} = (r, \ldots, r)^T$, $\mathbf{u}, \mathbf{v}, \mathbf{z} \geq \mathbf{0}$ has an integer solution

$$\mathbf{w}_0 = (u_1^0, \ldots, u_m^0, v_1^0, \ldots, v_m^0, z_1^0, \ldots, z_m^0)^T$$

with $\|\mathbf{w}_0\| \leq (n+1)^{n+1}\|\mathbf{A}\|^n$ (note that $\mathbf{A}^T$ is an $n \times m$ matrix). Let $\mathbf{y}_0 = (u_1^0 - v_1^0, \ldots, u_m^0 - v_m^0)^T$. Then $\mathbf{y}_0$ is an integer vector and $\mathbf{A}^T\mathbf{y}_0 - (z_1^0, \ldots z_m^0)^T = (r, \ldots, r)^T \geq (1, \ldots, 1)^T$. Thus, $\mathbf{A}^T\mathbf{y}_0 \geq (1, \ldots, 1)^T$ or equivalently $\mathbf{y}_0^T\mathbf{A} \geq (1, \ldots, 1)$. Moreover, $\|\mathbf{y}_0\| \leq \max_i\{|u_i^0 - v_i^0|\} \leq 2\|\mathbf{w}_0\| \leq 2(n+1)^{n+1}\|\mathbf{A}\|^n$. $\square$

Now we discuss the properties of integer solutions for the integer linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$. We say that a solution $\mathbf{x}$ to a linear system is *nontrivial* if $\mathbf{x}$ has at least one nonzero element.

**Lemma 5.2.5** *If the integer linear system $\mathbf{A}\mathbf{x} = \mathbf{0}$, $\mathbf{x} \geq \mathbf{0}$ has no nontrivial integer solution $\mathbf{x}_0$ with $\|\mathbf{x}_0\| \leq (m+1)^{m+1}\|\mathbf{A}\|^m$, then the system $\mathbf{y}^T\mathbf{A}$ $\geq (1, \ldots, 1)$ has an integer solution $\mathbf{y}_0$ with $\|\mathbf{y}_0\| \leq 2(n+1)^{n+1}\|\mathbf{A}\|^n$.*

PROOF. Suppose the opposite that the system $\mathbf{y}^T\mathbf{A} \geq (1, \ldots, 1)$ has no integer solution $\mathbf{y}_0$ with $\|\mathbf{y}_0\| \leq 2(n+1)^{n+1}\|\mathbf{A}\|^n$, then by Corollary 5.2.4, the linear system $\mathbf{y}^T\mathbf{A} \geq (1, \ldots, 1)$ has no (real or integer) solution at all. Consider the following instance $\alpha'$ for the LINEAR PROGRAMMING problem

The Instance $\alpha'$

$$\begin{array}{ll} \text{maximize} & \mathbf{y}^T\mathbf{0} \\ \text{subject to} & \mathbf{y}^T\mathbf{A} \geq (1, \ldots, 1) \end{array}$$

The instance $\alpha'$ is the dual instance of the following instance $\alpha$ for the LINEAR PROGRAMMING problem:

$$\begin{array}{rc} & \underline{\text{The Instance } \alpha} \\ \text{minimize} & -\sum_{i=1}^{n} x_i \\ \text{subject to} & \mathbf{Ax} = \mathbf{0},\ \mathbf{x} \geq \mathbf{0} \end{array}$$

The instance $\alpha$ has an obvious solution $\mathbf{x} = \mathbf{0}$. Since the dual instance $\alpha'$ has no solution, by Theorem 4.3.2, the instance $\alpha$ has unbounded solutions. Thus, we can assume without loss of generality that the instance $\alpha$ has a solution $\mathbf{x}' = (x'_1, \ldots, x'_n)^T$ with $x'_1 \geq 1$. By Lemma 5.2.3 (note $\mathbf{b} = \mathbf{0}$ for the instance $\alpha$), the instance $\alpha$ has a nontrivial integer solution $\mathbf{x}_0 = (x_1^0, \ldots, x_n^0)$ with $x_1^0 \geq 1$ and $\|\mathbf{x}_0\| \leq (m+1)^{m+1}\|\mathbf{A}\|^m$. But this contradicts our assumption on the linear system $\mathbf{Ax} = \mathbf{0}$, $\mathbf{x} \geq \mathbf{0}$.

This contradiction proves the lemma. □

With the above preparations, now we are ready to derive an upper bound on integer solutions to an integer linear system.

**Theorem 5.2.6** *If the integer linear system* $\mathbf{Ax} = \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$ *has an integer solution, then it has an integer solution* $\mathbf{x}_0$ *with* $\|\mathbf{x}_0\| \leq 3(q+1)^{2q+4}\|\mathbf{A}\|^{2q+1}\|\mathbf{b}\|_*$, *where* $q = \max\{m, n\}$, *and* $\|\mathbf{b}\|_* = \max\{\|\mathbf{b}\|, 1\}$.

PROOF. Let $\mathbf{x}_0 = (x_1^0, \ldots, x_n^0)$ be an integer solution to the system $\mathbf{Ax} = \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$ with the value $\sum_{i=1}^{n} x_i^0$ minimized.

If all $x_i^0 \leq (m+1)^{m+1}\|\mathbf{A}\|^m$, then $\|\mathbf{x}_0\| \leq 3(q+1)^{2q+4}\|\mathbf{A}\|^{2q+1}\|\mathbf{b}\|_*$, and we are done.

Otherwise, suppose without loss of generality $x_1^0, \ldots, x_k^0 > (m+1)^{m+1}\|\mathbf{A}\|^m$ and $x_{k+1}^0, \ldots, x_n^0 \leq (m+1)^{m+1}\|\mathbf{A}\|^m$. Let $\mathbf{B} = [\mathbf{a}_1, \ldots, \mathbf{a}_k]$ be the submatrix consisting of the first $k$ columns of $\mathbf{A}$.

If $\mathbf{Bu} = \mathbf{0}$, $\mathbf{u} \geq \mathbf{0}$ has a nontrivial integer solution $\mathbf{u}_0 = (u_1^0, \ldots, u_k^0)$ with $\|\mathbf{u}_0\| \leq (m+1)^{m+1}\|\mathbf{B}\|^m$, then the vector $\mathbf{x}' = (x'_1, \ldots, x'_n)$, where $x'_i = x_i^0 - u_i^0$ for $i = 1, \ldots, k$ and $x'_j = x_j^0$ for $j = k+1, \ldots, n$, satisfies

$$\mathbf{Ax}' = \mathbf{Ax}_0 - \mathbf{Bu}_0 = \mathbf{Ax}_0 = \mathbf{b}$$

and $\mathbf{x}' \geq \mathbf{0}$ since $x_i^0 > (m+1)^{m+1}\|\mathbf{A}\|^m$ and $u_i^0 \leq (m+1)^{m+1}\|\mathbf{B}\|^m \leq (m+1)^{m+1}\|\mathbf{A}\|^m$ for all $i = 1, \ldots, k$. Thus, the integer vector $\mathbf{x}'$ is also a solution to the integer linear system $\mathbf{Ax} = \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$. However, this contradicts our assumption that $\sum_{i=1}^{n} x_i^0$ is minimized over all integer solutions to the system $\mathbf{Ax} = \mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$ since (note the vector $\mathbf{u}_0$ is nontrivial)

$$\sum_{i=1}^{n} x_i' = \sum_{i=1}^{n} x_i^0 - \sum_{i=1}^{k} u_i^0 < \sum_{i=1}^{n} x_i^0$$

Therefore, the integer linear system $\mathbf{Bu} = \mathbf{0}$, $\mathbf{u} \geq \mathbf{0}$ must have no nontrivial integer solutions $\mathbf{u}_0$ with $\|\mathbf{u}_0\| \leq (m+1)^{m+1}\|\mathbf{B}\|^m$. By Lemma 5.2.5, the system $\mathbf{y}^T\mathbf{B} \geq (1, \ldots, 1)$ has an integer solution $\mathbf{y}_0 = (y_1^0, \ldots, y_m^0)$ with $\|\mathbf{y}_0\| \leq 2(k+1)^{k+1}\|\mathbf{B}\|^k \leq 2(n+1)^{n+1}\|\mathbf{A}\|^n$.

Multiplying both sides of the equality $\mathbf{Ax}_0 = \mathbf{b}$ from left by the vector $\mathbf{y}_0^T$, we obtain

$$\sum_{i=1}^{n} x_i^0 \mathbf{y}_0^T \mathbf{a}_i = \mathbf{y}_0^T \mathbf{b}$$

Since $\mathbf{y}_0^T\mathbf{B} \geq (1, \ldots, 1)$, $\mathbf{y}_0^T\mathbf{a}_i \geq 1$ for $i = 1, \ldots, k$. We get for each $i$, $i = 1, \ldots, k$,

$$0 \leq x_i^0 \leq \sum_{i=1}^{k} x_i^0 \leq \sum_{i=1}^{k} x_i^0 \mathbf{y}_0^T \mathbf{a}_i = \mathbf{y}_0^T \mathbf{b} - \sum_{j=k+1}^{n} x_j^0 \mathbf{y}_0^T \mathbf{a}_j$$

Since

$$|\mathbf{y}_0^T \mathbf{b}| \leq m\|\mathbf{y}_0\| \cdot \|\mathbf{b}\| \leq 2m(n+1)^{n+1}\|\mathbf{A}\|^n\|\mathbf{b}\| \leq 2(q+1)^{q+2}\|\mathbf{A}\|^q\|\mathbf{b}\|$$

and since for $j = k+1, \ldots, n$ we have $x_j^0 \leq (m+1)^{m+1}\|\mathbf{A}\|^m$

$$|x_j^0 \mathbf{y}_0^T \mathbf{a}_j| \leq x_j^0 m\|\mathbf{y}_0\| \cdot \|\mathbf{A}\| \leq 2(q+1)^{2q+3}\|\mathbf{A}\|^{2q+1}$$

Thus, for $i = 1, \ldots, k$,

$$x_i^0 \leq |\mathbf{y}_0^T \mathbf{b}| + \sum_{j=k+1}^{n} |x_j^0 \mathbf{y}_0^T \mathbf{a}_j| \leq 3(q+1)^{2q+4}\|\mathbf{A}\|^{2q+1}\|\mathbf{b}\|_*$$

This gives $\|\mathbf{x}_0\| \leq 3(q+1)^{2q+4}\|\mathbf{A}\|^{2q+1}\|\mathbf{b}\|_*$. $\square$

So far we have concentrated on integer solutions for integer linear systems and have ignored the objective function $\mathbf{c}^T\mathbf{x}$ in the instance $\alpha =$ **(b, c, A)** of the INTEGER LP. In fact, if we know the objective function value $\sigma$ of optimal solutions to the instance $\alpha$, then a bound on optimal integer solutions to $\alpha$ can be derived directly from Theorem 5.2.6: an optimal integer solution to the instance $\alpha$ must be an integer solution to the linear system

$\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{c}^T\mathbf{x} = \sigma$, and $\mathbf{x} \geq \mathbf{0}$. According to Theorem 5.2.6, this should give an upper bound on optimal integer solutions to $\alpha$ in terms of $\mathbf{A}$, $\mathbf{b}$, $\mathbf{c}$, and $\sigma$. Thus, to derive such a bound, we first need to derive an upper bound for the objective function value $\sigma$ of optimal solutions to $\alpha$.

**Lemma 5.2.7** *Suppose that the instance* $\alpha = (\mathbf{b}, \mathbf{c}, \mathbf{A})$ *for the* INTEGER LP *problem has a finite optimal solution* $\mathbf{x}_0$. *Then* $\alpha$ *as an instance for the* LINEAR PROGRAMMING *problem also has a finite optimal solution.*

PROOF. Assume the opposite that $\alpha$ as an instance for the LINEAR PROGRAMMING problem has unbounded solutions. Then by Theorem 4.3.1, the linear system $\mathbf{y}^T\mathbf{A} \leq \mathbf{c}^T$ has no solution. Thus, the instance $\alpha_1'$ for the LINEAR PROGRAMMING problem

| | The Instance $\alpha_1'$ |
|---|---|
| maximize | $\mathbf{y}^T\mathbf{0}$ |
| subject to | $\mathbf{y}^T\mathbf{A} \leq \mathbf{c}^T$ |

also has no solution. Since $\alpha_1'$ is the dual instance of the instance $\alpha_1 = (\mathbf{0}, \mathbf{c}, \mathbf{A})$ and $\alpha_1$ has an obvious solution $\mathbf{x} = \mathbf{0}$, by Theorem 4.3.2, the instance $\alpha_1$ has unbounded solutions. Therefore, there is a solution $\mathbf{x}_1$ to $\alpha_1$ such that $\mathbf{c}^T\mathbf{x}_1 < 0$.

Such a solution $\mathbf{x}_1$ can actually be found by the simplex method. Applying the simplex method using a tableau $\mathcal{T}$, starting from the instance $\alpha_1$. We must get into a stage in which a negative reduced cost coefficient $r_q < 0$ is found and all elements in the $q$th column of the tableau $\mathcal{T}$ are less than or equal to 0 (see Lemma 4.2.3). In this case, a solution $\mathbf{x}_1$ to the instance $\alpha_1$ satisfying $\mathbf{c}^T\mathbf{x} < 0$ can be constructed using the elements in the tableau $\mathcal{T}$, the elements in the vector $\mathbf{c}$, and a sufficiently large positive integer $\epsilon$ (see Section 4.3, the discussion before Lemma 4.2.3). Because the tableau $\mathcal{T}$ starts with the elements in $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{c}$, which are all integers, and because the tableau transformations perform only additions, subtractions, multiplications, and divisions on tableau elements, we conclude that all tableau elements are rational numbers. In consequence, all elements in the solution $\mathbf{x}_1$ are rational numbers. Now let $\mathbf{x}_2 = r\mathbf{x}_1$ for a proper large positive integer $r$, then the vector $\mathbf{x}_2$ is an integer solution to the instance $\alpha_1 = (\mathbf{0}, \mathbf{c}, \mathbf{A})$ satisfying (note $\mathbf{c}^T\mathbf{x}_1 < 0$) $\mathbf{c}^T\mathbf{x}_2 \leq \mathbf{c}^T\mathbf{x}_1 < 0$.

Finally, let $\mathbf{x}_3 = \mathbf{x}_0 + \mathbf{x}_2$, then $\mathbf{x}_3$ is an integer vector and

$$\mathbf{A}\mathbf{x}_3 = \mathbf{A}\mathbf{x}_0 + \mathbf{A}\mathbf{x}_2 = \mathbf{A}\mathbf{x}_0 = \mathbf{b} \qquad \mathbf{x}_3 \geq \mathbf{0}$$

Thus, $\mathbf{x}_3$ is an integer solution to the instance $\alpha$. Moreover,

$$\mathbf{c}^T\mathbf{x}_3 = \mathbf{c}^T\mathbf{x}_0 + \mathbf{c}^T\mathbf{x}_2 < \mathbf{c}^T\mathbf{x}_0$$

But this contradicts our assumption that $\mathbf{x}_0$ is the optimal integer solution to the instance $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$.

In conclusion, $\alpha$ as an instance for the LINEAR PROGRAMMING problem must have a finite optimal solution. □

Now we are ready to derive an upper bound on the objective function value of optimal solutions for an instance for the INTEGER LP problem.

**Theorem 5.2.8** *Suppose that the instance* $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ *for the* INTEGER LP *has a finite optimal solution* $\mathbf{x}_0$. *Then*

$$|\mathbf{c}^T\mathbf{x}_0| \leq 3(q+1)^{2q+5}\|\mathbf{A}\|^{2q+1}\|\mathbf{c}\| \cdot \|\mathbf{b}\|_*$$

*where* $q = \max\{m, n\}$ *and* $\|\mathbf{b}\|_* = \max\{\|\mathbf{b}\|, 1\}$.

PROOF. If $\mathbf{c}^T\mathbf{x}_0 \geq 0$, then by Theorem 5.2.6, there is an integer solution $\mathbf{x}_1$ to $\alpha$ such that $\|\mathbf{x}_1\| \leq 3(q+1)^{2q+4}\|\mathbf{A}\|^{2q+1}\|\mathbf{b}\|_*$. The theorem follows because

$$|\mathbf{c}^T\mathbf{x}_0| = \mathbf{c}^T\mathbf{x}_0 \leq \mathbf{c}^T\mathbf{x}_1 \leq n\|\mathbf{c}\| \cdot \|\mathbf{x}_1\|$$

On the other hand, suppose that $\mathbf{c}^T\mathbf{x}_0 < 0$. Since $\alpha$ as an instance for the INTEGER LP problem has a finite optimal solution, by Lemma 5.2.7, $\alpha$ as an instance for the LINEAR PROGRAMMING problem also has a finite optimal solution. By Theorem 4.1.1, $\alpha$, as an instance for the LINEAR PROGRAMMING problem, has an optimal basic solution $\mathbf{x}_2$. By Corollary 5.2.2, each element in $\mathbf{x}_2$ can be expressed as a quotient $r_i/r$ of two non-negative integers $r_i$ and $r$ such that $r_i \leq m^m\|\mathbf{A}\|^{m-1}\|\mathbf{b}\|$. In particular, no element in $\mathbf{x}_2$ is larger than the integer $m^m\|\mathbf{A}\|^{m-1}\|\mathbf{b}\|$. Let $x'_{\max}$ be the largest element in $\mathbf{x}_2$ (note $\mathbf{x}_2 \geq \mathbf{0}$), then $x_{\max} \leq m^m\|\mathbf{A}\|^{m-1}\|\mathbf{b}\| \leq q^q\|\mathbf{A}\|^{q-1}\|\mathbf{b}\|$. Now the theorem follows because

$$|\mathbf{c}^T\mathbf{x}_0| \leq |\mathbf{c}^T\mathbf{x}_2| \leq n\|\mathbf{c}\|x'_{\max}$$

□

Now we are ready for our main theorem.

**Theorem 5.2.9** *Suppose that the instance* $\alpha = (\mathbf{b},\ \mathbf{c},\ \mathbf{A})$ *for the* INTEGER LP *problem has a finite optimal solution, then* $\alpha$ *has an optimal solution*

$\mathbf{x}_0$ *such that* $\|\mathbf{x}_0\| \leq 9(q+2)^{4q+11} a_{\max}^{4q+6}$, *where* $q = \max\{m, n\}$, *and* $a_{\max} = \max\{\|\mathbf{b}\|, \|\mathbf{c}\|, \|\mathbf{A}\|\}$.

PROOF. Let $\sigma$ be the value of $\mathbf{c}^T\mathbf{x}$ for any optimal solution $\mathbf{x}$ to $\alpha$. Then an integer vector $\mathbf{x}$ is an optimal solution to the instance $\alpha$ if and only if $\mathbf{x}$ is a solution to the linear system $\mathbf{A}^+\mathbf{x} = \mathbf{b}^+$, $\mathbf{x} \geq \mathbf{0}$, where $\mathbf{A}^+$ is an $(m+1) \times n$ matrix and $\mathbf{b}^+$ is an $(m+1)$-dimensional vector such that

$$\mathbf{A}^+ = \begin{pmatrix} \mathbf{A}^+ \\ \mathbf{c}^T \end{pmatrix} \qquad \mathbf{b}^+ = \begin{pmatrix} \mathbf{b} \\ \sigma \end{pmatrix}$$

By Theorem 5.2.6, the integer linear system $\mathbf{A}^+\mathbf{x} = \mathbf{b}^+$, $\mathbf{x} \geq \mathbf{0}$, has an integer solution $\mathbf{x}_0$ such that

$$\|\mathbf{x}_0\| \leq 3(q+2)^{2q+6} a_{\max}^{2q+3} \|\mathbf{b}^+\|_*$$

Now the theorem follows since $\|\mathbf{b}^+\|_* = \max\{\|\mathbf{b}\|, |\sigma|, 1\}$ and by Theorem 5.2.8,

$$|\sigma| \leq 3(q+1)^{2q+5} \|\mathbf{A}\|^{2q+1} \|\mathbf{c}\| \cdot \|\mathbf{b}\|_* \leq 3(q+2)^{2q+5} a_{\max}^{2q+3}$$

□

**Corollary 5.2.10** *The* INTEGER LP *problem is NPO.*

PROOF. Let $\alpha = (\mathbf{b}, \mathbf{c}, \mathbf{A})$ be any instance of the INTEGER LP problem. Let $q = \max\{m, n\}$ and $a_{\max} = \max\{\|\mathbf{b}\|, \|\mathbf{c}\|, \|\mathbf{A}\|\}$. Then the length $|\alpha|$ of the instance $\alpha$ is larger than $q + \log(a_{\max})$, i.e., we need at least one bit for each element in $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{A}$, and need at least $\log(a_{\max})$ bits for some element in $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{A}$.

By Theorem 5.2.9, there is an optimal solution $\mathbf{x}_0$ to $\alpha$ such that $\|\mathbf{x}_0\| \leq 9(q+2)^{4q+11} a_{\max}^{4q+6}$. Therefore, each element in $\mathbf{x}_0$ can be represented by at most $\log(\|\mathbf{x}_0\|)$ bits, and the optimal solution $\mathbf{x}_0$ can be represented by at most $n \log(\|\mathbf{x}_0\|)$ bits. Now

$$\log(\|\mathbf{x}_0\|) \leq \log 9 + (4q+11)\log(q+2) + (4q+6)\log(a_{\max}) = O(|\alpha|^2)$$

That is, the optimal solution $\mathbf{x}_0$ of the instance $\alpha$ can be represented by $O(n|\alpha|^2) = O(|\alpha|^3)$ bits. □

There is an intuitive geometric interpretation for Corollary 5.2.2 and Theorem 5.2.9. Recall that the constraints $\mathbf{A}\mathbf{x} = \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$ define a polytope $P$ in the $n$-dimensional Euclidean space $\mathcal{E}^n$ and that basic solutions correspond to extreme points on the polytope $P$. Corollary 5.2.2 says that the extreme points on $P$ cannot be "too far" from the original unless very large integers are involved in defining the polytope $P$ that create very small angles. Theorem 5.2.9 basically says that it is very difficult for the solution set for the instance $\alpha$ to avoid all small integer vectors unless very small angles caused by very large coefficients are involved.

## 5.3 Polynomial time approximation

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

**Definition 5.3.1** An algorithm $\mathcal{A}$ is an *approximation algorithm* for an optimization problem $Q = (I_Q, S_Q, f_Q, opt_Q)$, if on any input instance $x \in I_Q$, the algorithm $\mathcal{A}$ produces a solution $y \in S_Q(x)$.

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

**Definition 5.3.2** An approximation algorithm $\mathcal{A}$ for an optimization problem $Q = (I_Q, S_Q, f_Q, opt_Q)$ has an *approximation ratio* $r(n)$, if on any input instance $x \in I_Q$, the solution $y$ produced by the algorithm $\mathcal{A}$ satisfies

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

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

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

**Algorithm. Graham-Schedule**

Input: $I = \langle t_1, \ldots, t_n; m \rangle$, all integers
Output: a scheduling of the $n$ jobs of processing time $t_1, t_2, \ldots, t_n$ on $m$ identical processors

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

Figure 5.1: Graham-Schedule

**Remark 5.3.3** By the definition, an approximation ratio is at least as large as 1. It is easy to see that the closer the approximation ratio to 1, the better the approximation quality of the approximation algorithm.

**Definition 5.3.4** An optimization problem *can be polynomial time approximated to a ratio* $r(n)$ if it has a polynomial time approximation algorithm whose approximation ratio is $r(n)$.

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

Makespan

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

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

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

$opt_Q$: min

A simple approximation algorithm is based on the greedy method: to minimize the parallel completion time, we always assign the next job to the processor that has the lightest load. This algorithm is due to R. Graham [56], and is given in Figure 5.1.

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

Now we study the approximation ratio of the algorithm **Graham-Schedule**.

**Theorem 5.3.1** *The algorithm* **Graham-Schedule** *for the* MAKESPAN *problem has approximation ratio bounded by* $2 - (1/m)$.

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

If the processor $P_1$ is assigned only one job, then the job has processing time $T$, and any scheduling on $\alpha$ has parallel completion time at least $T$. In this case, the scheduling $S$ is an optimal scheduling with approximation ratio 1.

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

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

Thus

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

Now let the parallel completion time of an optimal scheduling on the instance $\alpha$ be $Opt(\alpha)$. It is easily to see that $Opt(\alpha)$ is at least as large as $\left(\sum_{i=1}^{n} t_i\right)/m$, and at least as large as $t_0$. We conclude

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

This gives

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

and completes the proof. □

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

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

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

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

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

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

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

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

**Lemma 5.3.2** *The approximation ratio* $2 - (1/m)$ *for the approximation algorithm* **Graham-Schedule** *is tight.*

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

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

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

Now we consider the second question: can we have a polynomial time approximation algorithm for the MAKESPAN problem that has an approximation ratio better than $2-(1/m)$. By looking at the instance $\alpha$ constructed on the proof of Lemma 5.3.2, we should realize that the bad performance for the algorithm **Graham-Schedule** occurs in the situation that we first assign small jobs, which somehow gives a balanced assignment among the processors, while a latter large job may simply break the balance by increasing the load of one processor significantly while unchanging the load of the other processors. This then results in a very unbalanced assignment among the processors thus worsens the parallel completion time. To avoid this situation, we presort the jobs, in a non-increasing order of their processing time, before we apply the algorithm **Graham-Schedule**. We call this the **Modified Graham-Schedule**.

**Theorem 5.3.3** *The* **Modified Graham-Schedule** *algorithm for the* MAKESPAN *problem has an approximation ratio bounded by 4/3.*

PROOF. According to the algorithm, we assume $t_1 \geq \cdots \geq t_n$ in the input $\alpha = \langle t_1, \ldots, t_n; m \rangle$ to the algorithm **Graham-Schedule**, and analyze the approximation ratio of the algorithm.

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

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

$$k - h - 1 = 2(m - h) \tag{5.6}$$

and by the assumption on the index $k$, for each $i \leq h$, we have $t_i + t_k > T_0$. Now consider any optimal scheduling $\mathcal{S}_0$ on the instance $\alpha$. The parallel completion time of $\mathcal{S}_0$ is $T_0$. If a processor is assigned a job $t_i$ by $\mathcal{S}_0$ with $i \leq h$, then the processor cannot be assigned any other jobs in $t_1, \ldots, t_k$ by $\mathcal{S}_0$ since $t_k$ is the smallest among $t_1, \ldots, t_k$ and $t_i + t_k > T_0$. Moreover, no processor is assigned more than two jobs in $t_{h+1}, \ldots, t_k$ since each of these jobs has processing time larger than $T_0/3$. Therefore, we need at least $h + \lceil (k-h)/2 \rceil = h + (m - h + 1) = m + 1$ processors for the jobs $t_1, \ldots, t_k$ in order to keep the parallel completion time larger than $T_0$ (note here we have used the equality (5.6)). This contradicts the fact that $\mathcal{S}_0$ is an optimal scheduling of parallel completion time $T_0$ on the instance $\langle t_1, \ldots, t_n; m \rangle$.

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

Now let $\mathcal{S}$ be the scheduling constructed by the algorithm **Modified Graham-Schedule** with parallel completion time $T$ for the instance $\alpha$. If $T = T_0$, then the approximation ratio is 1 so less than 4/3. If $T > T_0$, let processor $P_j$ have load $T$ and let $t_k$ be the last job assigned to processor $P_j$.

Since when the algorithm assigns $t_k$ to $P_j$ the parallel completion time of $t_1, \ldots, t_k$ exceeds $T_0$, by the above discussion, we must have $t_k \leq T_0/3$ ($t_k$ may not be the first such a job but recall that the jobs are sorted in non-increasing order). Let $T = t + t_k$. Then at the time the job $t_k$ was assigned, all processors had load at least $t$. Therefore, $mt \leq \sum_{i=1}^{k-1} t_i \leq \sum_{i=1}^{n} t_i$, which implies $t \leq \sum_{i=1}^{n} t_i/m \leq T_0$. This gives immediately

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

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

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

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

It is natural to ask whether the bound $\frac{4}{3} - \frac{1}{3m}$ is tight on approximation ratios for approximation algorithms for the MAKESPAN problem. It is, in fact, not. For example, Hochbaum and Shmoys [63] have developed a polynomial time approximation algorithm of approximation ratio $1 + \epsilon$ for any fixed constant $\epsilon > 0$. Such an algorithm is called a *polynomial time approximation scheme* for the problem. Therefore, there are polynomial time approximation algorithms for the MAKESPAN problem whose approximation ratio can be arbitrarily close to 1. These kind of approximation algorithms will be discussed in more detail in the next few chapters.

Unfortunately, the algorithm presented in [63] runs in time $O((n/\epsilon)^{1/\epsilon^2})$, which is totally impractical even for moderate values of $n$ and $\epsilon$. Thus, we come to the third question: can we keep the approximation ratio $1+\epsilon$ while improving the running time for approximation algorithms for the MAKESPAN problem? Progress has been made towards this direction. For example, recently, Hochbaum and Shmoys [65] have developed an approximation algorithm for the MAKESPAN problem whose approximation ratio remains $1 + \epsilon$ with running time improved to $O(n) + f(1/\epsilon)$, where $f(1/\epsilon)$ is a function independent of $n$.

# Part II

# $(1+\epsilon)$-Approximable Problems

# Chapter 6

# Fully Polynomial Time Approximation Schemes

Recall that the approximation ratio for an approximation algorithm is a measure to evaluate the approximation performance of the algorithm. The closer the ratio to 1 the better the approximation performance of the algorithm. It is notable that there is a class of NP-hard optimization problems, most originating from scheduling problems, for which there are polynomial time approximation algorithms whose approximation ratio $1 + \epsilon$ can be as close to 1 as desired. Of course, the running time of such an algorithm increases with the reciprocal of the error bound $\epsilon$, but in a very reasonable way: it is bounded by a polynomial of $1/\epsilon$. Such an approximation algorithm is called a *fully polynomial time approximation scheme* (or shortly FPTAS) for the NP-hard optimization problem. A fully polynomial time approximation scheme seems the best possible we can expect for approximating an NP-hard optimization problem.

In this chapter, we introduce the main techniques for constructing fully polynomial time approximation schemes for NP-hard optimization problems. These techniques include pseudo-polynomial time algorithms and approximation by scaling. Two NP-hard optimization problems, the KNAPSACK problem and the $c$-MAKESPAN problem, are used as examples to illustrate the techniques. In the last section of this chapter, we also give a detailed discussion on what NP-hard optimization problems may not have a fully polynomial time approximation scheme. An important concept, the strong NP-hardness, is introduced, and we prove that in most cases, a strongly NP-hard optimization problem has no fully polynomial time approximation scheme unless our working conjecture P $\neq$ NP fails.

## 6.1 Pseudo-polynomial time algorithms

We first consider algorithms that solve certain NP-hard optimization problems *precisely*. Of course, we cannot expect that these algorithms run in polynomial time. However, these algorithms run in *pseudo-polynomial time* in the sense that the running time of these algorithms is bounded by a two-variable polynomial whose variables are the length of the input instance and the largest number appearing in the input instance. These pseudo-polynomial time algorithms will play a crucial role in our later development of approximation algorithms for the NP-hard optimization problems.

We start with the KNAPSACK problem, which is defined as follows.

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

That is, the KNAPSACK problem is to take the maximum value with a knapsack of size $B$, given $n$ items of size $s_i$ and value $v_i$, $i = 1, \ldots, n$. To simplify our description, for a subset $S$ of $\{1, \ldots, n\}$, we will call $\sum_{i\in S} s_i$ the *size* of $S$ and $\sum_{i\in S} v_i$ the *value* of $S$. Let $V_0$ be a value not smaller than the value of optimal solutions to the instance $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$. For each index $i$, $1 \leq i \leq n$ and for each value $v \leq V_0$, we consider the following question

> **Question** $K(i, v)$
>
> Is there a subset $S$ of $\{1, \ldots, i\}$ such that the size of $S$ is not larger than $B$ and the value of $S$ is equal to $v$?

The answer to Question $K(i, v)$ is "yes" *if and only if* at least one of the following two cases is true: (1) there is a subset $S'$ of $\{1, \ldots, i-1\}$ such that the size of $S'$ is not larger than $B$ and the value of $S'$ is equal to $v$ (in this case, simply let $S$ be $S'$), and (2) there is a subset $S''$ of $\{1, \ldots, i-1\}$ such that the size of $S''$ is not larger than $B - s_i$ and the value of $S''$ is equal to $v - v_i$ (in this case, let $S = S'' \cup \{i\}$). Therefore, the solution to Question $K(i, v)$ seems to be implied by solutions to the questions of the form $K(i-1, *)$.

For small values of $i$ and $v$, the solution to Question $K(i, v)$ can be decided directly. In particular, the answer to $K(0, v)$ is always "no" for $v > 0$; and the answer to $K(0, 0)$ is "yes".

The above discussion motivates the following dynamic programming algorithm for solving the KNAPSACK problem. We first compute $K(0, v)$ for all $v$, $0 \le v \le V_0$. Then, inductively we compute each $K(i, v)$ based on the solutions to $K(i-1, v')$ for all $0 \le v' \le V_0$. For each item $K(i, v)$, we associate it with a subset $S$ in $\{1, \ldots, i\}$ such that the size of $S$ is not larger than $B$ and the value of $S$ is equal to $v$, if such a subset exists at all.

Now a potential problem arises. How do we handle two different witnesses for a "yes" answer to the Question $K(i, v)$? More specifically, suppose that we find two subsets $S_1$ and $S_2$ of $\{1, \ldots, i\}$ such that both of $S_1$ and $S_2$ have size bounded by $B$ and value equal to $v$, should we keep both of them with $K(i, v)$, or ignore one of them? Keeping both can make $K(i, v)$ exponentially grow as $i$ increases, which will significantly slow down our algorithm. Thus, we intend to ignore one of $S_1$ and $S_2$. Which one do we want to ignore? Intuitively, the one with larger size should be ignored (recall that $S_1$ and $S_2$ have the same value). However, we must make sure that ignoring the set of larger size will not cause a loss of the track of optimal solutions to the original instance of the KNAPSACK problem. This is ensured by the following lemma.

**Lemma 6.1.1** *Let $S_1$ and $S_2$ be two subsets of $\{1, \ldots, i\}$ such that $S_1$ and $S_2$ have the same value, and the size of $S_1$ is at least as large as the size of $S_2$. If $S_1$ leads to an optimal solution $S = S_1 \cup S_3$ for the* KNAPSACK *problem, where $S_3 \subseteq \{i+1, \ldots, n\}$, then $S' = S_2 \cup S_3$ is also an optimal solution for the* KNAPSACK *problem.*

PROOF. Let $size(S)$ and $value(S)$ denote the size and value of a subset $S$ of $\{1, \ldots, n\}$, respectively. We have

$$size(S') = size(S_2) + size(S_3)$$

and

$$size(S) = size(S_1) + size(S_3)$$

By the assumption that $size(S_1) \ge size(S_2)$, we have $size(S) \ge size(S')$. Since $S$ is an optimal solution, we have $size(S) \le B$, which implies $size(S') \le B$. Thus $S'$ is also a solution to the KNAPSACK problem. Moreover, since $value(S_1) = value(S_2)$, we have

$$\begin{aligned} value(S') &= value(S_2) + value(S_3) \\ &= value(S_1) + value(S_3) \\ &= value(S) \end{aligned}$$

**Algorithm. Knapsack-Dyn**$(n, V_0)$
Input: $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$, all positive integers
Output: a subset $S \subseteq \{1, ..., n\}$ of size $\leq B$ and value maximized

**Subroutine. Put**$(S_0, K[i, v])$
  **if** $K[i, v] = *$ **then** $K[i, v] = S_0$
  **else if** $size(S_0) < size(K[i, v])$ **then** $K[i, v] = S_0$.

1. **for** all $0 \leq i \leq n$ and $0 \leq v \leq V_0$ **do** $K[i, v] = *$;
2. $K[0, 0] = \phi$; $\{\phi$ is the empty set$\}$
3. **for** all $0 \leq i \leq n - 1$ and $0 \leq v \leq V_0$ **do**
   **if** $K[i, v] \neq *$ **then**
     **Put**$(K[i, v], K[i + 1, v])$;
     **if** $size(K[i, v]) + s_{i+1} \leq B$ **then**
       **Put**$(K[i, v] \cup \{i + 1\}, K[i + 1, v + v_{i+1}])$;
4. return the item $K[n, v] \neq *$ with $v$ maximized.

Figure 6.1: Dynamic programming for KNAPSACK

Thus, $S'$ is also an optimal solution. □

By Lemma 6.1.1, for two subsets $S_1$ and $S_2$ of $\{1, \ldots, i\}$ that both witness the "yes" answer to Question $K(i, v)$, if the one of larger size leads to an optimal solution, then the one with smaller size also leads to an optimal solution. Therefore, ignoring the set of larger size will not lead to a loss of the track of optimal solutions. That is, if we can derive an optimal solution based on the set of larger size, then we can also derive an optimal solution based on the set of smaller size using exactly the same procedure.

Now a dynamic programming algorithm based on the above discussion can be described as in Figure 6.1. Here the order of computation is slightly different from the one described above: instead of computing $K(i, v)$ based on $K(i - 1, v)$ and $K(i - 1, v - v_i)$, we start from each $K(i - 1, v')$ and try to "extend" it to $K(i, v')$ and $K(i, v' + v_i)$.

The subroutine $\mathbf{Put}(S_0, K[i, v])$ is used to solve the multiple witness problem, where $S_0$ is a subset of $\{1, \ldots, i\}$ such that $S_0$ has value $v$.

Step 4 of the algorithm **Knapsack-Dyn**$(n, V_0)$ finds the largest value $v \leq V_0$ such that $K[n, v] \neq *$. Obviously, if $V_0$ is not smaller than the value of optimal solutions to the input instance, then step 4 of the algorithm will find the subset $S$ of $\{1, 2, \ldots, n\}$ with the largest value under the constraint that $S$ has size bounded by $B$.

According to our discussion, it should be clear that the algorithm **Knapsack-Dyn**$(n, V_0)$ solves the KNAPSACK problem for any value $V_0$ not smaller than the value of optimal solutions to the input instance.

**Lemma 6.1.2** *The algorithm* **Knapsack-Dyn**$(n, V_0)$ *runs in time* $O(nV_0)$.

PROOF. We show data structures on which the **if** statement in Step 3 can be executed in constant time. The theorem follows directly from this discussion.

For each item $K[i, v]$, which is for a subset $S_{iv}$ of $\{1, \ldots, i\}$, we attach three parameters: (1) the size of $S_{iv}$, (2) a marker $m_{iv}$ indicating whether $i$ is contained in $S_{iv}$, and (3) a pointer $p_{iv}$ to an item $K[i-1, v']$ in the previous row such that the set $S_{iv}$ is derived from the set $K[i-1, v']$. Note that the actual set $S_{iv}$ is *not* stored in $K[i, v]$.

With these parameters, the size of the set $S_{iv}$ can be directly read from $K[i, v]$ in constant time. Moreover, it is also easy to verify that the subroutine calls $\texttt{Put}(K[i, v], K[i+1, v])$ and $\texttt{Put}(K[i, v] \cup \{i+1\}, K[i+1, v+v_{i+1}])$ can be performed in constant time by updating the parameters in $K[i+1, v]$ and $K[i+1, v+v_{i+1}]$.

This shows that steps 1-3 of the algorithm **Knapsack-Dyn**$(n, V_0)$ take time $O(nV_0)$.

We must show how the actual optimal solution $K[n, v]$ is returned in step 4. After we have decided the item $K[n, v]$ in step 4, which corresponds to an optimal solution $S_{nv}$ that is a subset of $\{1, \ldots, n\}$, we first check the marker $m_{nv}$ to see if $S_{nv}$ contains $n$, then follow the point $p_{nv}$ to an item $K[n-1, v']$, where we can check whether the set $S_{nv}$ contains $n-1$ and a pointer to an item in the $(n-2)$nd row, and so on. In time $O(n)$, we will be able to "backtrack and collect" all elements in $S_{nv}$ and return the actual set $S_{nv}$. $\square$

A straightforward implementation of the algorithm **Knapsack-Dyn**$(n, V_0)$ uses a 2-dimensional array $K[0..n, 0..V_0]$, which takes $O(nV_0)$ amount of computer memory. A more careful implementation can reduce the amount of computer memory from $O(nV_0)$ to $O(V_0)$, as follows. Observe that at any moment, only two rows $K[i, \cdot]$ and $K[i+1, \cdot]$ of the array $K[0..n, 0..V_0]$ need to be kept: when the values of the $i$th row become available, all values for rows before the $i$th row will not be used further so they do not need to be kept. Therefore, in the algorithm **Knapsack-Dyn**$(n, V_0)$, we can use two arrays of size $V_0$ to keep the current two rows, which take only $O(V_0)$ amount of computer memory.

In general, we can conveniently let the bound $V_0 = \sum_{i=1}^{n} v_i$, which is an obvious upper bound for the optimal solution value. With this bound $V_0$, the algorithm **Knapsack-Dyn**$(n, V_0)$ runs in time polynomial in both $n$ and $V_0$, and solves the KNAPSACK problem precisely. Unfortunately, since the value $V_0$ can be larger than any polynomial of $n$, the algorithm **Knapsack-Dyn**$(n, V_0)$ is not a polynomial time algorithm in terms of the input length $n$. On the other hand, the algorithm **Knapsack-Dyn**$(n, V_0)$ does provide very important information about the KNAPSACK problem, in particular from the following two viewpoints:

1. If values of all items in the input instance are bounded by a polynomial of $n$, then the value $V_0$ is also bounded by a polynomial of $n$. In this case, the algorithm **Knapsack-Dyn**$(n, V_0)$ runs in time polynomial in $n$ and constructs an optimal solution for each given input instance; and

2. The algorithm has laid an important foundation for approximation algorithms for the KNAPSACK problem. This will be discussed in detail in the next section.

The algorithm **Knapsack-Dyn**$(n, V_0)$ is a typical method for solving a class of optimization problems, in particular many scheduling problems. To study this method in a more general sense, we first introduce a few terminologies.

**Definition 6.1.1** Suppose $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ is an optimization problem. For each input instance $x \in I_Q$ we define:

- $\text{length}(x)$ = the length of a binary representation of $x$; and
- $\max(x)$ = the largest number that appears in the input $x$.

In particular, if no number appears in the input instance $x$, we define $\max(x) = 1$.

The definitions of $\text{length}(x)$ and $\max(x)$ can vary by some degree without loss of the generality of our discussion. For example, $\text{length}(x)$ can also denote the length of the representation of the input $x$ based on any fixed alphabet, and $\max(x)$ can be defined to be the sum of all numbers appearing in the input $x$. Our discussion below will be valid for any of these variations. The point is that for two different definition systems ($\text{length}(x)$, $\max(x)$) and ($\text{length}'(x)$, $\max'(x)$), we require that $\text{length}(x)$ and $\text{length}'(x)$ are polynomially related and that $\max(x)$ and $\max'(x)$ are polynomially related for all input instances $x$.

**Definition 6.1.2** Let $Q$ be an optimization problem. A algorithm $\mathcal{A}$ solving $Q$ runs in *pseudo-polynomial time* if there is a two-variable polynomial $p$ such that on any input instance $x$ of $Q$, the running time of the algorithm $\mathcal{A}$ is bounded by $p(\text{length}(x), \max(x))$. In this case, we also say that the problem $Q$ is solvable in pseudo-polynomial time.

Consider the KNAPSACK problem. It is clear that for any input instance $\alpha = \langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$, if we let $V_0 = \sum_{i=1}^{n} v_i \leq n \cdot \max(\alpha)$, then the algorithm **Knapsack-Dyn**$(n, V_0)$ constructs an optimal solution for a given instance $\alpha$ in time $O(nV_0)$, which is bounded by a polynomial of length$(\alpha)$ and $\max(\alpha)$. Thus,

**Theorem 6.1.3** *The* KNAPSACK *problem is solvable in pseudo-polynomial time.*

The KNAPSACK problem is a maximization problem. As another example, we present a pseudo-polynomial time algorithm for the minimization problem $c$-MAKESPAN, where $c$ is a fixed positive integer.

Recall that the $c$-MAKESPAN problem is defined as follows.

$c$-MAKESPAN $= (I_Q, S_Q, f_Q, opt_Q)$

$I_Q$: the set of tuples $T = \langle t_1, \ldots, t_n\rangle$, where each $t_i$ is an integer denoting the processing time for the $i$th job

$S_Q$: $S_Q(T)$ is the set of partitions $P = (S_1, \ldots, S_c)$ of the numbers $\langle t_1, \ldots, t_n\rangle$ into $c$ parts ($P$ is called a *scheduling* of $\langle t_1, \ldots, t_n\rangle$)

$f_Q$: $f_Q(T, P) = \max_{1 \leq d \leq c}\{\sum_{t_j \in S_d} t_j\}$

$opt_Q$: min

By Theorem 5.1.3, the $c$-MAKESPAN problem is NP-hard. Thus, there is no polynomial time algorithm for the $c$-MAKESPAN problem unless P = NP.

Let $T_0$ be a value not smaller than the value of optimal solutions to the instance $\langle t_1, \ldots, t_n\rangle$. Note that every scheduling $(S_1, \ldots, S_c)$ of the $n$ jobs $\langle t_1, \ldots, t_n\rangle$, where $S_d$ is the subset of $\{1, \ldots, n\}$ that corresponds to the jobs assigned to the $d$th processor, can be written as a $c$-tuple $(T_1, \ldots, T_c)$ with $0 \leq T_d \leq T_0$ for all $1 \leq d \leq c$, where $T_d = \sum_{h \in S_d} t_h$ is the total execution time assigned to the $d$th processor. The $c$-tuple $(T_1, \ldots, T_c)$ will be called the *time configuration* for the scheduling $(S_1, \ldots, S_c)$.

Now as for the KNAPSACK problem, for each $i$, $0 \leq i \leq n$, and for each time configuration $(T_1, \ldots, T_c)$, $1 \leq T_d \leq T_0$, $1 \leq d \leq c$, we ask the question

**Algorithm.** $c$-**Makespan-Dyn**$(n, T_0)$

Input: $n$ jobs with processing time $t_1, \ldots, t_n$
Output: an optimal scheduling of the jobs on $c$ processors

1. **for** $i = 0$ **to** $n$ **do**
   **for** each time configuration $(T_1, \ldots, T_c)$, $0 \le T_d \le T_0$ **do**
   $H[i, T_1, \ldots, T_c] = *$;
2. $H[0, 0, \ldots, 0] = 0$;
3. **for** $i = 0$ **to** $n - 1$ **do**
   **for** each time configuration $(T_1, \ldots, T_c)$, $0 \le T_d \le T_0$ **do**
   **if** $H[i, T_1, \ldots, T_c] \ne *$ **then**
   **for** $d = 1$ **to** $c$ **do**
   $H[i+1, T_1, \ldots, T_{d-1}, T_d + t_{i+1}, T_{d+1}, \ldots, T_c] = d$;
   { record that job $t_{i+1}$ is assigned to processor $d$. }
4. return the $H[n, T_1, \ldots, T_c] \ne *$ with $\max_d\{T_d\}$ minimized;

Figure 6.2: Dynamic programming for $c$-Makespan

> Is there a scheduling of the first $i$ jobs $\{t_1, \ldots, t_i\}$ that gives the time configuration $(T_1, \ldots, T_c)$?

This question is equivalent to the following question

> Is there an index $d$ such that the first $i - 1$ jobs $\{t_1, \ldots, t_{i-1}\}$ can be scheduled with the time configuration $(T_1, \ldots, T_{d-1}, T_d - t_i, T_{d+1}, \ldots, T_c)$?

This observation suggests the dynamic programming algorithm given in Figure 6.2. A $c+1$ dimensional array $H[0..n, 0..T_0, \ldots, 0..T_0]$ is used such that the item $H[i, T_1, \ldots, T_c]$ records the existence of a scheduling on the first $i$ jobs with the time configuration $(T_1, \ldots, T_c)$. Again, instead of recording the whole scheduling corresponding to $H[i, T_1, \ldots, T_c]$, we can simply record the processor index to which the $i$th job is assigned. A pointer is used in $H[i, T_1, \ldots, T_c]$ that points to an item of form $H[i-1, \cdot, \ldots, \cdot]$ so that the machine assignment of each of the first $i - 1$ jobs can be found following the pointers.

An obvious upper bound $T_0$ on the value of optimal solutions is $\sum_{i=1}^{n} t_i$. The following theorem follows directly from the algorithm $c$-**Makespan-Dyn**$(n, T_0)$, with $T_0 = \sum_{i=1}^{n} t_i$.

**Theorem 6.1.4** *The algorithm* $c$**-Makespan-Dyn**$(n, T_0)$ *solves the problem* $c$-MAKESPAN *in time* $O(nT_0^c)$*. In consequence, the* $c$-MAKESPAN *problem is solvable in pseudo-polynomial time.*

In many practical applications, developing a pseudo-polynomial time algorithm for an NP-hard optimization problem may have significant impact. First, in most practical applications, numbers appearing in an input instance are in general not extremely large. For example, numbers appearing in scheduling problems in general represent processing resource (e.g., computational time and storage) requirements for tasks, which are unlikely to be very large because we will actually process the tasks after the scheduling and we could not afford to do so if any task requires an inordinately large amount of resource. For this kind of applications, a pseudo-polynomial time algorithm will become a polynomial time algorithm and solve the problem, even if the original problem is NP-hard in its general form.

Furthermore, a pseudo-polynomial time algorithm can be useful even when there is no natural bound on the numbers appearing in input instances. In general, input instances that are of practical interests and contain very large numbers might be very rare. If this is the case, then a pseudo-polynomial time algorithm will work efficiently for most input instances, and only "slow down" in very rare situations.

## 6.2 Approximation by scaling

In the last section, we presented an algorithm **Knapsack-Dyn**$(n, V_0)$ that, on an input instance $\alpha = \langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$ of the KNAPSACK problem, constructs an optimal solution for $\alpha$ in time $O(nV_0)$, where $V_0 = \sum_{i=1}^n v_i$. If $V_0$ is not bounded by any polynomial function of $n$, then the running time of the algorithm is not polynomial. Is there a way to lower the value of $V_0$? Well, an obvious way is to divide each value $v_i$ by a sufficiently large number $K$ so that $V_0$ is replaced by a smaller value $V_0' = V_0/K$. In order to let the algorithm **Knapsack-Dyn**$(n, V_0')$ to run in polynomial time, we must have $V_0' \le cn^d$ for some constants $c$ and $d$, or equivalently, $K \ge V_0/(cn^d)$. Another problem is that the value $v_i/K$ may no longer be an integer while by our definition, all input values in an instance of the KNAPSACK problem are integers. Thus, we will take $v_i' = \lfloor v_i/K \rfloor$. This gives a new instance $\alpha'$ for the KNAPSACK problem

$$\alpha' = \langle s_1, \ldots, s_n; v_1', \ldots, v_n'; B\rangle$$

where $v'_i = \lfloor v_i/K \rfloor$, for $i = 1, \ldots, n$, and $V'_0 = \lceil V_0/K \rceil$ is obviously an upper bound on the value of optimal solutions to the instance $\alpha'$. For $K \geq V_0/(cn^d)$ for some constants $c$ and $d$, the algorithm **Knapsack-Dyn**$(n, V'_0)$ finds an optimal solution for $\alpha'$ in polynomial time. Note that a solution to $\alpha'$ is also a solution to $\alpha$ and we intend to "approximate" the optimal solution to $\alpha$ by an optimal solution to $\alpha'$. Since the application of the floor function $\lfloor \cdot \rfloor$, we lose precision thus an optimal solution for $\alpha'$ may not be an optimal solution for $\alpha$. How much precision have we lost? Intuitively, the larger the value $K$, the more precision we would lose. Thus, we want $K$ to be as small as possible. On the other hand, we want $K$ to be as large as possible so that the running time of the algorithm **Knapsack-Dyn**$(n, V'_0)$ can be bounded by a polynomial. Now a natural question is whether there is a value $K$ that makes the algorithm **Knapsack-Dyn**$(n, V'_0)$ run in polynomial time and cause not much precision loss so that the optimal solution to the instance $\alpha'$ is "close" to the optimal solution to the instance $\alpha$. For this, we need the following formal analysis.

Let $S \subseteq \{1, \ldots, n\}$ be an optimal solution to the instance $\alpha$, and let $S' \subseteq \{1, \ldots, n\}$ be the optimal solution to the instance $\alpha'$ produced by the algorithm **Knapsack-Dyn**$(n, V'_0)$. Note that $S$ is also a solution to the instance $\alpha'$ and that $S'$ is also a solution to the instance $\alpha$. Let $Opt(\alpha) = \sum_{i \in S} v_i$ and $Apx(\alpha) = \sum_{j \in S'} v_i$ be the objective function values of the solutions $S$ and $S'$, respectively. Therefore, $Opt(\alpha)/Apx(\alpha)$ is the approximation ratio for the algorithm we proposed. In order to bound the approximation ratio by a given constant $\epsilon$, we consider

$$\begin{aligned}
Opt(\alpha) &= \sum_{i \in S} v_i \\
&= K \sum_{i \in S} \frac{v_i}{K} \\
&\leq K \sum_{i \in S} (\lfloor \frac{v_i}{K} \rfloor + 1) \\
&\leq Kn + K \sum_{i \in S} \lfloor \frac{v_i}{K} \rfloor \\
&= Kn + K \sum_{i \in S} v'_i
\end{aligned}$$

The last inequality is because the cardinality of the set $S$ is bounded by $n$.

Now since $S'$ is an optimal solution to $\alpha' = \langle s_1, \ldots, s_n; v'_1, \ldots, v'_n; B \rangle$, while $S$ is also a solution to $\alpha'$, we must have

$$\sum_{i \in S} v'_i \le \sum_{i \in S'} v'_i$$

Thus,

$$\begin{aligned} Opt(\alpha) &\le Kn + K \sum_{i \in S'} v'_i \\ &= Kn + K \sum_{i \in S'} \lfloor \frac{v_i}{K} \rfloor \\ &\le Kn + K \sum_{i \in S'} \frac{v_i}{K} \\ &= Kn + Apx(\alpha) \end{aligned} \tag{6.1}$$

This gives us the approximation ratio.

$$\frac{Opt(\alpha)}{Apx(\alpha)} \le 1 + \frac{Kn}{Apx(\alpha)}$$

Without loss of generality, we can assume that $s_i \le B$ for all $i = 1, \ldots, n$ (otherwise, the index $i$ can be simply deleted from the input instance since it can never make contribution to a feasible solution to $\alpha$). Thus, $Opt(\alpha)$ is at least as large as $\max_{1 \le i \le n}\{v_i\} \ge V_0/n$, where $V_0 = \sum_{i=1}^{n} v_i$. From inequality (6.1), we have

$$Apx(\alpha) \ge Opt(\alpha) - Kn \ge \frac{V_0}{n} - Kn$$

It follows that

$$\begin{aligned} \frac{Opt(\alpha)}{Apx(\alpha)} &\le 1 + \frac{Kn}{\frac{V_0}{n} - Kn} \\ &= 1 + \frac{Kn^2}{V_0 - Kn^2} \end{aligned}$$

Thus, in order to bound the approximation ratio by $1 + \epsilon$, one should have

$$\frac{Kn^2}{V_0 - Kn^2} \le \epsilon$$

This leads to $K \le (\epsilon V_0)/(n^2(1+\epsilon))$.

Recall that to make the algorithm **Knapsack-Dyn**$(n, V'_0)$ run in polynomial time on the input instance $\alpha'$, we must have $K \ge V_0/(cn^d)$ for some

**Algorithm. Knapsack-Apx**

Input: $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B \rangle$, and $\epsilon > 0$
Output: a subset $S' \subseteq \{1, ..., n\}$, such that $\sum_{i \in S'} s_i \leq B$

1. $V_0 = \sum_{i=1}^n v_i$; $\quad K = \frac{V_0}{(1+1/\epsilon)n^2}$;
2. **for** $i = 1$ **to** $n$ **do** $\quad v_i' = \lfloor v_i / K \rfloor$;
3. apply algorithm **Knapsack-Dyn**$(n, \lceil V_0/K \rceil)$ on
   $\langle s_1, \ldots, s_n; v_1', \ldots, v_n'; B \rangle$
4. return the subset $S' \subseteq \{1, \ldots, n\}$ obtained in step 3.

Figure 6.3: FPTAS for the KNAPSACK problem

constants $c$ and $d$. Combining these two relations, we get $c = 1 + 1/\epsilon$, and $d = 2$, and the value

$$K = V_0/(cn^d) = \frac{V_0}{(1 + 1/\epsilon)n^2}$$

makes the algorithm **Knapsack-Dyn**$(n, V_0')$ run in time $O(n^3(1 + 1/\epsilon)) = O(n^3/\epsilon)$ and produces a solution $S'$ to the instance $\alpha$ with approximation ratio bounded by $\epsilon$.

We summarize the above discussion in the algorithm given in Figure 6.3.

**Theorem 6.2.1** *For any input instance $\alpha$ of the* KNAPSACK *problem and for any real number $\epsilon > 0$, the algorithm* **Knapsack-Apx** *runs in time $O(n^3/\epsilon)$ and produces a solution to $\alpha$ with approximation ratio bounded by $1 + \epsilon$.*

According to Theorem 6.2.1, the running time of the approximation algorithm **Knapsack-Apx** increases when the input size $n$ increases and the error bound $\epsilon$ decreases. This seems reasonable and necessary. Moreover, the running time of the algorithm increases "slowly" with $n$ and $1/\epsilon$ — it is bounded by a polynomial of $n$ and $1/\epsilon$. This seems the best we can expect for an approximation algorithm for an NP-hard optimization problem. This motivates the following definition.

**Definition 6.2.1** An optimization problem $Q$ has a *fully polynomial time approximation scheme* (FPTAS) if it has an approximation algorithm $A$ such that given $\langle x, \epsilon \rangle$, where $x$ is an input instance of $Q$ and $\epsilon$ is a positive

**Algorithm.** $c$-**Makespan-Apx**

Input: $\langle t_1, \ldots, t_n; \epsilon\rangle$, all $t_i$'s are integers
Output: a scheduling of the $n$ jobs on $c$ processors

1. $T_0 = \sum_{i=1}^{n} t_i$; $\quad K = \epsilon T_0/(cn)$;
2. **for** $i = 1$ **to** $n$ **do** $\quad t'_i = \lceil t_i/K \rceil$;
3. Apply algorithm $c$-**Makespan-Dyn**$(n, T'_0)$ on input $\langle t'_1, \ldots, t'_n\rangle$, where $T'_0 = \lceil T_0/K \rceil + n$;
4. return the scheduling obtained in step 3.

Figure 6.4: FPTAS for the $c$-Makespan problem

constant, $A$ finds a solution for $x$ with approximation ratio bounded by $1+\epsilon$ in time polynomial in both $n$ and $1/\epsilon$.

By the definition, the Knapsack problem has a fully polynomial time approximation scheme. In the following, we present a fully polynomial time approximation scheme for the $c$-Makespan problem.

The approach for developing a fully polynomial time approximation scheme for the $c$-Makespan problem is similar to that for the Knapsack problem. For an input instance $\alpha = \langle t_1, \ldots, t_n\rangle$ of the $c$-Makespan problem, we have the dynamic programming algorithm $c$-**Makespan-Dyn**$(n, T_0)$, which constructs an optimal solution to the instance $\alpha$ in time $O(nT_0^c)$, where $T_0 = \sum_{i=1}^{n} t_i$. We reduce the running time of the algorithm by scaling the value $T_0$ by dividing all $t_i$ in the input instance $\alpha$ by a large number $K$. By properly choosing the scaling factor $K$, we can make the algorithm $c$-**Makespan-Dyn** to run on the new instance in polynomial time and keep the approximation ratio bounded by $1+\epsilon$. Because of the similarity, some details in the algorithms and in the analysis are omitted. The reader is advised to refer to corresponding discussion on the Knapsack problem and complete the omitted parts for a better understanding.

The approximation algorithm for the $c$-Makespan problem is presented in Figure 6.4.

**Theorem 6.2.2** *The algorithm* $c$-**Makespan-Apx** *on input* $\langle t_1, \ldots, t_n; \epsilon\rangle$ *produces a scheduling* $(S'_1, \ldots, S'_c)$ *with approximation ratio bounded by* $1+\epsilon$ *and runs in time* $O(n^{c+1}/\epsilon^c)$.

Proof. It is easy to see that the time complexity of the algorithm $c$-**Makespan-Apx** is dominated by step 3.

Since $T'_0 = \lceil T_0/K \rceil + n = cn/\epsilon + n = O(n/\epsilon)$, by Theorem 6.1.4, the algorithm $c$-**Makespan-Dyn**$(n, T'_0)$ in step 3, thus the algorithm $c$-**Makespan-Apx**, runs in time $O(n(T'_0)^c) = O(n^{c+1}/\epsilon^c)$.

Now let $(S_1, \ldots, S_c)$ be an optimal solution to the input instance $\alpha = \langle t_1, \ldots, t_n \rangle$ of the $c$-MAKESPAN problem, and let $(S'_1, \ldots, S'_c)$ be the optimal solution to the input instance $\alpha' = \langle t'_1, \ldots, t'_n \rangle$ obtained by the algorithm $c$-**Makespan-Dyn**. Note that $(S_1, \ldots, S_c)$ is also a solution to the instance $\alpha' = \langle t'_1, \ldots, t'_n \rangle$ and $(S'_1, \ldots, S'_c)$ is also a solution to the instance $\alpha = \langle t_1, \ldots, t_n \rangle$.

For all $d$, $1 \le d \le c$, let

$$T_d = \sum_{h \in S_d} t_h \qquad V_d = \sum_{h \in S_d} t'_h$$
$$T'_d = \sum_{h \in S'_d} t_h \qquad V'_d = \sum_{h \in S'_d} t'_h$$

Without loss of generality, suppose

$$T_1 = \max_{1 \le d \le c} \{T_d\} \qquad V_2 = \max_{1 \le d \le c} \{V_d\}$$
$$T'_3 = \max_{1 \le d \le c} \{T'_d\} \qquad V'_4 = \max_{1 \le d \le c} \{V'_d\}$$

Therefore, on instance $\langle t_1, \ldots, t_n \rangle$, the scheduling $(S_1, \ldots, S_c)$ has parallel completion time $T_1$ and the scheduling $(S'_1, \ldots, S'_c)$ has parallel completion time $T'_3$; and on instance $\langle t'_1, \ldots, t'_n \rangle$, the scheduling $(S_1, \ldots, S_c)$ has parallel completion time $V_2$ and the scheduling $(S'_1, \ldots, S'_c)$ has parallel completion time $V'_4$. The approximation ratio given by the algorithm $c$-**Processor-Apx** is $T'_3/T_1$.

We have

$$T'_3 = \sum_{h \in S'_3} t_h = K \sum_{h \in S'_3} (t_h/K) \le K \sum_{h \in S'_3} t'_h = KV'_3 \le KV'_4$$

The last inequality is by the assumption $V'_4 = \max_{1 \le d \le c}\{V'_d\}$.

Now since $(S'_1, \ldots, S'_c)$ is an optimal scheduling on instance $\langle t'_1, \ldots, t'_n \rangle$, we have $V'_4 \le V_2$. Thus,

$$T'_3 \le KV_2 = K \sum_{h \in S_2} t'_h = K \sum_{h \in S_2} \lceil t_h/K \rceil$$
$$\le K \sum_{h \in S_2} (\frac{t_h}{K} + 1) \le T_2 + Kn \le T_1 + Kn$$

The last inequality is by the assumption $T_1 = \max_{1 \leq d \leq c}\{T_d\}$.

This gives us immediately

$$T_3'/T_1 \leq 1 + Kn/T_1$$

It is easy to see that $T_1 \geq \sum_{i=1}^{n} t_i/c = T_0/c$, and recall that $K = \epsilon T_0/(cn)$, we obtain $Kn/T_1 \leq \epsilon$. That is, the scheduling $(S_1', \ldots, S_c')$ produced by the algorithm **$c$-Makespan-Apx** has approximation ratio bounded by $1+\epsilon$. $\square$

**Corollary 6.2.3** *For a fixed constant c, the c-*MAKESPAN *problem has a fully polynomial time approximation scheme.*

Theorem 6.2.1 and Theorem 6.2.2 present fully polynomial time approximation schemes for the KNAPSACK problem and the $c$-MAKESPAN problem, respectively, using the pseudo-polynomial time algorithms for the problems by properly scaling and rounding input instances. Most known fully polynomial time approximation schemes for optimization problems are derived using this method. In fact, ??? showed the evidence that this is essentially the only way to derive fully polynomial time approximation schemes for optimization problems. Therefore, pseudo-polynomial time algorithms are closely related to fully polynomial time approximation schemes for optimization problems. Actually, we can show that under a very general condition, having a pseudo-polynomial time algorithm is a necessary condition for the existence of a fully polynomial time approximation scheme for an optimization problem.

**Theorem 6.2.4** *Let $Q = \langle I, S, f, opt\rangle$ be an optimization problem such that for all input instance $x \in I$ we have $Opt(x) \leq p(length(x), \max(x))$, where $p$ is a two variable polynomial. If $Q$ has a fully polynomial time approximation scheme, then $Q$ can be solved in pseudo-polynomial time.*

PROOF. Suppose that $Q$ is a minimization problem, i.e., $opt = \min$. Since $Q$ has a fully polynomial time approximation scheme, there is an approximation algorithm $A$ for $Q$ such that for any input instance $x \in I$, the algorithm $A$ produces a solution $y \in S(x)$ in time $p_1(|x|, 1/\epsilon)$ satisfying

$$\frac{f(x,y)}{Opt(x)} \leq 1 + \epsilon$$

where $p_1$ is a two variable polynomial.

In particular, let $\epsilon = 1/(p(\text{length}(x), \max(x)) + 1)$, then the solution $y$ satisfies

$$f(x, y) \leq Opt(x) + \frac{Opt(x)}{p(\text{length}(x), \max(x)) + 1} < Opt(x) + 1$$

Now since both $f(x, y)$ and $Opt(x)$ are integers and $f(x, y) \geq Opt(x)$, we get immediately $f(x, y) = Opt(x)$. That is, the solution produced by the algorithm $A$ is actually an optimal solution. Moreover, the running time of the algorithm $A$ for producing the solution $y$ is bounded by

$$p_1(|x|, p(\text{length}(x), \max(x)) + 1)$$

which is a polynomial of length$(x)$ and $\max(x)$. We conclude that the optimization problem $Q$ can be solved in pseudo-polynomial time. $\square$

## 6.3 Improving time complexity

We have shown that the KNAPSACK problem and the $c$-MAKESPAN problem can be approximated within a ratio $1 + \epsilon$ in polynomial time for any given $\epsilon > 0$. On the other hand, one should observe that the running time of the approximation algorithms is very significant. For the KNAPSACK problem, the running time of the approximation algorithm **Knapsack-Apx** is $O(n^3/\epsilon)$; and for the $c$-MAKESPAN problem, the running time of the approximation algorithm $c$-**Makespan-Apx** is $O(n^{c+1}/\epsilon^c)$. When the input size $n$ is reasonably large and the required error bound $\epsilon$ is very small, these algorithms may become impractical.

In this section, we discuss several techniques that have been used extensively in developing efficient approximation algorithms for scheduling problems. We should point out that these techniques are not only useful for improving the algorithm running time, but also often important for achieving better approximation ratios.

### Reducing the number of parameters

Consider the approximation algorithm $c$-**Makespan-Apx** for the problem $c$-MAKESPAN (Figure 6.4). The running time of the algorithm is dominated by step 3, which is a call to the dynamic programming algorithm $c$-**Makespan-Dyn**$(n, T_0')$. Therefore, if we can improve the time complexity of the dynamic programming algorithm $c$-**Makespan-Dyn**$(n, T_0')$, we

```
.........
4.  for i = 0 to n − 1 do
      for each (T_1, ..., T_{c−1}), 0 ≤ T_d ≤ T_0 do
        if H[i, T_1, ..., T_{c−1}] ≠ * then
          H[i + 1, T_1, ..., T_{c−1}] = H[i, T_1, ..., T_{c−1}] + t_{i+1};
          { assign job t_{i+1} to processor c }
          for d = 1 to c − 1 do
            H[i + 1, T_1, ..., T_{d−1}, T_d + t_{i+1}, T_{d+1}, ..., T_{c−1}]
                          = H[i, T_1, ..., T_{c−1}];
            { assign job t_i to processor d. }
.........
```

Figure 6.5: Modified algorithm $c$-**Makespan-Dyn**

improve the running time of the approximation algorithm $c$-**Makespan-Apx**.

The algorithm $c$-**Makespan-Dyn**$(n, T_0')$ (see Figure 6.2) works on a $(c+1)$-dimensional array $H[0..n, 0..T_0', \ldots, 0..T_0']$, where $T_0' = O(n/\epsilon)$ (see the proof of Theorem 6.2.2). The item $H[i, T_1, \ldots, T_c] = d$ records the fact that there is a scheduling for the first $i$ jobs, which assigns the job $i$ to the processor $d$ with a time configuration $(T_1, \ldots, T_c)$. The running time of the algorithm $c$-**Makespan-Dyn**$(n, T_0')$ is necessarily at least $O(n(T_0')^c) = O(n^{c+1}/\epsilon^c)$.

To reduce the running time, we reduce the dimension of the array $H[\cdot, \cdots, \cdot]$ from $c+1$ to $c$, as follows. We let the value of the item $H[i, T_1, \ldots, T_{c-1}]$ to record the processing time of the $c$th processor. More precisely, $H[i, T_1, \ldots, T_{c-1}] = T_c$ if there is a scheduling for the first $i$ jobs whose time configuration is $(T_1, \ldots, T_{c-1}, T_c)$. The modification of the algorithm $c$-**Makespan-Dyn**$(n, T_0')$ based on this change is straightforward, for which we present the part for step 3 in Figure 6.5. Of course, we still need to keep another two pieces of information related to each item $H[i, T_1, \ldots, T_{c-1}]$: a processor index $d$ indicating that the job $i$ is assigned to processor $d$, and a pointer to an item $H[i-1, T_1', \ldots, T_{c-1}']$ for constructing the actual scheduling corresponding to the time configuration $(T_1, \ldots T_{c-1}, H[i, T_1, \ldots, T_{c-1}])$.

The running time of the algorithm $c$-**Makespan-Dyn**$(n, T_0)$ now is obviously bounded by $O(nT_0^{c-1})$. Therefore, if step 3 in the algorithm $c$-**Makespan-Apx** (Figure 6.4) calls the modified algorithm $c$-**Makespan-**

**Dyn**$(n, T_0')$, where $T_0' = O(n/\epsilon)$, the running time of the algorithm $c$-**Makespan-Apx** is reduced from $O(n(T_0')^c) = O(n^{c+1}/\epsilon^c)$ to $O(n(T_0')^{c-1}) = O(n^c/\epsilon^{c-1})$. We summarize the discussion in the following theorem.

**Theorem 6.3.1** *The algorithm* $c$-**Makespan-Apx** *on input* $\langle t_1, \ldots, t_n; \epsilon \rangle$ *produces a scheduling* $(S_1', \ldots, S_c')$ *with approximation ratio bounded by* $1 + \epsilon$ *and runs in time* $O(n^c/\epsilon^{c-1})$.

### Reducing the search space

Consider the dynamic programming algorithm **Knapsack-Dyn**$(n, V_0)$ (Figure 6.1). For an input instance $\alpha = \langle s_1, \ldots, s_n; v_1, \ldots, v_n; B \rangle$, in order to let the algorithm **Knapsack-Dyn**$(n, V_0)$ construct an optimal solution, $V_0$ must be not smaller than the value of optimal solutions to the input instance. In particular, we can let $V_0 = \sum_{i=1}^{n} v_i$. We used a 2-dimensional array $K[0..n, 0..V_0]$. The item $K[i, v]$ records the fact that there is a subset of $\{1, \ldots, i\}$ of value $v$ and size bounded by $B$. Note that the value of an optimal solution to $\alpha$ can be as small as $V_0/n$. Therefore, if we can derive a closer upper bound $V^*$ on the value of optimal solutions to $\alpha$, we may speed up our dynamic programming algorithm by calling **Knapsack-Dyn**$(n, V^*)$ instead of **Knapsack-Dyn**$(n, V_0)$.

To derive a better bound on the optimal solution value, we can perform a "pre-approximation algorithm" that provides a bound $V^*$ not much larger than the optimal solution value. Then this value $V^*$ can be used as an upper bound for the optimal solution value in our dynamic programming algorithm.

Let $S$ be a set of items whose size and value are $s_i$ and $v_i$, respectively, for $i = 1, \ldots, n$. Let $B$ be an integer. A *$B$-partition of $S$* is a triple $(S', S'', r)$, where $r \in S''$, such that

(1) $S' \cup S'' = S$ and $S' \cap S'' = \emptyset$;

(2) $v_j/s_j \geq v_r/s_r \geq v_k/s_k$ for all $j \in S'$ and all $k \in S''$; and

(3) $\sum_{j \in S'} s_j \leq B$ but $\sum_{j \in S'} s_j + s_r > B$.

Now consider the algorithm **Pre-Apx** given in Figure 6.6. We first analyze the complexity of the algorithm.

**Lemma 6.3.2** *The algorithm* **Pre-Apx** *runs in linear time.*

PROOF. It is sufficient to show that the $B$-partition $(S', S'', r)$ of the set $\{1, 2, \ldots, n\}$ can be constructed in linear time.

**Algorithm. Pre-Apx**

Input: $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$, all positive integers
Output: a subset $S$ of $\{1, \ldots, n\}$ of size bounded by $B$

1. construct a $B$-partition $(S', S'', r)$ for the set $\{1, 2, \ldots, n\}$;
2. let $v_k = \max_i\{v_i\}$;
3. **if** $v_k > \sum_{j \in S'} v_j$ **then** return $\{k\}$ **else** return $S'$.

Figure 6.6: Finding an upper bound on optimal solution value

If we sort the items by the ratios $v_i/s_i$, then the $B$-partition can be trivially constructed. However, sorting the items takes time $\Omega(n \log n)$. Therefore, we should avoid sorting.

We use the linear time algorithm that, given a set $S$ of $n$ numbers, returns the median of $S$ (i.e., the $(\lfloor n/2 \rfloor)$th largest number in $S$) (the readers are referred to [14] for more detailed discussion of this algorithm).

We perform a binary search procedure as follows. First we find, in linear time, an item $h$ in $S$ such that the ratio $v_h/s_h$ is the median over all ratios $v_1/s_1$, ..., $v_n/s_n$. The item $h$ partitions the set $S$ into two subsets $S_1$ and $S_2$ of equal size, where for each item $j$ in $S_1$, $v_j/s_j \geq v_h/s_h$, and for each item $k$ in $S_2$, $v_k/s_k \leq v_h/s_h$. Assume $h \in S_2$. The subsets $S_1$ and $S_2$ can be constructed in linear time. Let $size(S_1) = \sum_{j \in S_1} s_j$. There are two possible cases: (1) $size(S_1) \leq B$. In this case we recursively construct a $B'$-partition $(S_2', S_2'', r)$ of the set $S_2$, where $B' = B - size(S_1)$. Now $(S_1 \cup S_2', S_2'', r)$ is a $B$-partition of the set $S$; and (2) $size(S_1) > B$. In this case we construct a $B$-partition $(S_1', S_1'', r)$ of the set $S_1$. Then $(S_1', S_1'' \cup S_2, r)$ is a $B$-partition of the set $S$. Note that each of the subsets $S_1$ and $S_2$ has at most $n/2$ items. Thus, if we let $t(n)$ be the running time of this recursive procedure, we have

$$t(n) = O(n) + t(n/2)$$

It is easy to verify that $t(n) = O(n)$. That is, the $B$-partition $(S', S'', r)$ can be constructed in linear time. This completes the proof of the lemma. □

Note that the algorithm **Pre-Apx** is an approximation algorithm for the KNAPSACK problem, whose approximation ratio is given by the following lemma.

**Lemma 6.3.3** *The approximation algorithm* **Pre-Apx** *for the* KNAPSACK *problem has an approximation ratio bounded by* 2.

PROOF. First note the following fact, where $v_j$, $v_k$, $s_j$, and $s_k$ are all positive integers,

$$\frac{v_j}{s_j} \geq \frac{v_k}{s_k} \quad \text{implies} \quad \frac{v_j}{s_j} \geq \frac{v_j + v_k}{s_j + s_k} \geq \frac{v_k}{s_k} \tag{6.2}$$

Let $(S', S'', r)$ be the $B$-partition of $\{1, 2, \ldots, n\}$ constructed by the algorithm **Pre-Apx**. The algorithm **Pre-Apx** constructs a solution $S_{apx} \subseteq \{1, 2, \ldots, n\}$ whose value is

$$\max\{\sum_{j \in S'} v_j, v_1, v_2, \ldots, v_n\}$$

Let $\overline{S} = S' \cup \{r\}$ and let $S_{opt}$ be an optimal solution.

Let $S_0 = \overline{S} \cap S_{opt}$. Thus, $\overline{S} = S_0 \cup T_1$ and $S_{opt} = S_0 \cup T_2$, where $T_1 \cap T_2 = \emptyset$. Note that for any $j \in T_1$ and any $k \in T_2$, we have

$$\frac{v_j}{s_j} \geq \frac{v_r}{s_r} \geq \frac{v_k}{s_k}$$

Thus, by repeatedly using the relation (6.2), we have

$$\frac{\sum_{j \in T_1} v_j}{\sum_{j \in T_1} s_j} \geq \frac{v_r}{s_r} \geq \frac{\sum_{k \in T_2} v_k}{\sum_{k \in T_2} s_k} \tag{6.3}$$

Now sine the size of $\overline{S}$ is larger than $B$ while the size of $S_{opt}$ is bounded by $B$, we must have $\sum_{j \in T_1} s_j > \sum_{k \in T_2} s_k$. Combining this with the inequality in (6.3), we get

$$\sum_{j \in T_1} v_j \geq \sum_{j \in T_2} v_k$$

This shows that the value of the set $\overline{S}$ is not smaller than the value of the optimal solution $S_{opt}$. Since $\overline{S} = S' \cup \{r\}$, according to the algorithm **Pre-Apx**, the value of $\overline{S}$ is bounded by twice of the value of the solution $S_{apx}$ constructed by the algorithm **Pre-Apx**. This proves that the approximation ratio of the algorithm **Pre-Apx** is bounded by 2. $\square$

Therefore, given an input instance $\alpha$ of the KNAPSACK problem, we can first apply the algorithm **Pre-Apx** to construct a solution. Suppose that this solution has value $V^*$, then we have $V^* \leq Opt(\alpha) \leq 2V^*$, where $Opt(\alpha)$ is the value of an optimal solution to $\alpha$. Thus, the value $2V^*$ can be used as an upper bound for the optimal solution value for $\alpha$.

**Algorithm. Knapsack-Apx (Revision I)**

Input: $\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B\rangle$, and $\epsilon > 0$
Output: a subset $S' \subseteq \{1, ..., n\}$, such that $\sum_{i \in S'} s_i \leq B$

1. call algorithm **Pre-Apx** to obtain a solution of value $V^*$;
2. $K = \frac{V^*}{n(1+1/\epsilon)}$;
3. **for** $i = 1$ **to** $n$ **do** $v_i' = \lfloor v_i/K \rfloor$;
4. apply algorithm **Knapsack-Dyn**$(n, V_0')$ on $\langle s_1, \ldots, s_n; v_1', \ldots, v_n'; B\rangle$, where $V_0' = \lfloor 2V^*/K \rfloor$;
5. return the subset $S' \subseteq \{1, \ldots, n\}$ obtained in step 4.

Figure 6.7: Revision I for the FPTAS for the KNAPSACK problem

We show how this refinement improves the running time of our fully polynomial time approximation scheme for the KNAPSACK problem. For this, we modify our scaling factor $K$ in the algorithm **Knapsack-Apx** (Figure 6.3). The modified algorithm is given in Figure 6.7.

The following theorem shows that the modified algorithm **Knapsack-Apx (Revision I)** for the KNAPSACK problem has the same approximation ratio but the running time improved by a fact $n$.

**Theorem 6.3.4** *The algorithm* **Knapsack-Apx (Revision I)** *for the* KNAPSACK *problem has approximation ratio* $1+\epsilon$ *and running time bounded by* $O(n^2/\epsilon)$.

PROOF. Again the time complexity of the algorithm is dominated by step 4, which calls the dynamic programming algorithm **Knapsack-Dyn**$(n, V_0')$. By Lemma 6.1.2, step 4 of the algorithm takes time $O(nV_0')$. Since $V_0' = \lfloor 2V^*/K \rfloor = 2n(1+1/\epsilon) = O(n/\epsilon)$, we conclude that the running time of the algorithm **Knapsack-Apx (Revision I)** is bounded by $O(n^2/\epsilon)$.

We must ensure that the value $V_0'$ is a large enough upper bound for the value $Opt(\alpha')$ of optimal solutions to the input instance $\alpha' = \langle s_1, \ldots, s_n; v_1', \ldots, v_n'; B\rangle$. For this, let $S$ be an optimal solution to the instance $\alpha'$. Then

$$Opt(\alpha') = \sum_{i \in S} v_i' = \sum_{i \in S} \lfloor v_i/K \rfloor \leq (\sum_{i \in S} v_i)/K$$

Observing that $S$ is also a solution to the original instance $\alpha =$

$\langle s_1, \ldots, s_n; v_1, \ldots, v_n; B \rangle$ and by Lemma 6.3.3 $Opt(\alpha) \leq 2V^*$, we have

$$Opt(\alpha') \leq (\sum_{i \in S} v_i)/K \leq Opt(\alpha)/K \leq 2V^*/K$$

Since $Opt(\alpha')$ is an integer, we get $Opt(\alpha') \leq \lfloor 2V^*/K \rfloor = V_0'$. Therefore, the value $V_0'$ is a valid upper bound for the value $Opt(\alpha')$.

Now we analyze the approximation ratio for the algorithm **Knapsack-Apx (Revision I)**. Using exactly the same derivation as we did in Section 6.2, we get (see the relation in (6.1) in Section 6.2)

$$Opt(\alpha) \leq Kn + Apx(\alpha)$$

where $Apx(\alpha)$ is the value of the solution constructed by the algorithm **Knapsack-Apx (Revision I)**. Dividing both sides by $Apx(\alpha)$, we get

$$\frac{Opt(\alpha)}{Apx(\alpha)} \leq 1 + \frac{Kn}{Apx(\alpha)}$$

Moreover, since $Apx(\alpha) \geq Opt(\alpha) - Kn \geq V^* - Kn$ (note here we have used a better estimation $Opt(\alpha) \geq V^*$ than the one in Section 6.2, in which the estimation $Opt(\alpha) \geq \sum_{i=1}^n v_i/n$ was used), we get

$$\frac{Opt(\alpha)}{Apx(\alpha)} \leq 1 + \frac{Kn}{V^* - Kn} = 1 + \epsilon$$

This completes the proof of the theorem. □

### Separating large items and small items

Another popular technique for improving the running time (and sometimes also the approximation ratio) is to treat large items and small items in an input instance differently. The basic idea of this technique can be described as follows: we first set a threshold value $T$. The items whose value is larger than or equal to $T$ are *large items* and the items whose value is smaller than $T$ are *small items*. We use common methods, such as the dynamic programming method and the scaling method, to construct a solution for the large items. Then we add the small items by greedy method. This approach is based on the following observations: (1) the number of large items is relatively small so that the running time of the dynamic programming can be reduced; (2) applying the floor or ceiling function on the scaled values (such as $\lfloor v_i/K \rfloor$ for the KNAPSACK problem and $\lceil t_i/K \rceil$ for the $c$-MAKESPAN problem) only for large items in general loses less precision; and

(3) greedy method for adding small items in general introduces only small approximation errors.

We illustrate this technique by re-considering the $c$-MAKESPAN problem.

Let $\alpha = \langle t_1, t_2, \ldots, t_n \rangle$ be an input instance of the $c$-MAKESPAN problem. Let $T_0 = \sum_{i=1}^{n} t_i$. A job $t_i$ is a large job if $t_i \geq \epsilon T_0/c$, and a job $t_j$ is a small job if $t_i < \epsilon T_0/c$. Let $\alpha_l$ be the set of all large jobs and let $\alpha_s$ be the set of all small jobs. Note that the number $n_l$ of large jobs is bounded by

$$n_l \leq \frac{T_0}{\epsilon T_0/c} = \frac{c}{\epsilon} \tag{6.4}$$

Without loss of generality, we suppose that the first $n_l$ jobs $t_1$, $t_2$, ..., $t_{n_l}$ are large jobs and the rest of the jobs are small jobs.

Apply the algorithm $c$-**Makespan-Apx** (see Figure 6.4) to the $n_l$ large jobs $\langle t_1, t_2, \ldots, t_{n_l} \rangle$ with the following modifications:

1. let $T_0' = \sum_{j=1}^{n_l} t_j$, and set $K = \epsilon^2 T_0'/c^2$; and
2. use $T_0'' = \lceil T_0'/K \rceil + n_l$ in the call to the dynamic programming algorithm $c$-**Makespan-Dyn**$(n_l, T_0'')$ on the scaled instance $\alpha' = \langle t_1', t_2', \ldots, t_{n_l}' \rangle$, where $t_j' = \lceil t_j/K \rceil$, $j = 1, \ldots, n_l$.

The value $T_0''$ is a valid upper bound on the parallel completion time for the scaled instance $\alpha' = \langle t_1', t_2', \ldots, t_{n_l}' \rangle$ because we have (we use the inequality (6.4) here)

$$\sum_{j=1}^{n_l} t_j' = \sum_{j=1}^{n_l} \lceil t_j/K \rceil \leq \frac{\sum_{j=1}^{n_l} t_j}{K} + n_l = \frac{T_0'}{K} + n_l \leq T_0''$$

By the analysis in Theorem 6.3.1, the running time of the algorithm $c$-**Makespan-Apx** on the large jobs $\alpha' = \langle t_1, \ldots, t_{n_l} \rangle$ is bounded by $O(n_l (T_0'')^{c-1})$. Replacing $T_0''$ by $\lceil T_0'/K \rceil + n_l$, $n_l$ by $c/\epsilon$, and $K$ by $\epsilon^2 T_0'/c^2$, we conclude that the running time of the algorithm $c$-**Makespan-Apx** on the large jobs $\alpha_l = \langle t_1, \ldots, t_{n_l} \rangle$ is bounded by $O(1/\epsilon^{2c-1})$.

To analyze the approximation ratio for the algorithm $c$-**Makespan-Apx** on the large jobs $\alpha_l = \langle t_1, \ldots, t_{n_l} \rangle$, we follow exactly the same analysis given in the proof of Theorem 6.2.2 except that we replace $n$, the total number of jobs in the input, by $n_l$, the total number of large jobs. This analysis gives

$$\frac{Apx(\alpha_l)}{Opt(\alpha_l)} \leq 1 + \frac{K n_l}{Opt(\alpha_l)}$$

where $Apx(\alpha_l)$ is the parallel completion time of the scheduling constructed by the algorithm $c$-**Makespan-Apx** for the large jobs $\alpha_l$, while $Opt(\alpha_l)$ is the parallel completion time of an optimal scheduling for the large jobs $\alpha_l$. By the inequalities $n_l \le c/\epsilon$ and $Opt(\alpha_l) \ge T_0'/c$, we obtain

$$\frac{Apx(\alpha_l)}{Opt(\alpha_l)} \le 1 + \epsilon$$

Note that the optimal parallel completion time for the large job set $\alpha_l$ cannot be larger than the optimal parallel completion time for the original set $\alpha = \langle t_1, \ldots, t_n \rangle$ of jobs. Therefore, if we let $Opt(\alpha)$ be the optimal parallel completion time for the original job set $\alpha$, then we have

$$Apx(\alpha_l) \le Opt(\alpha_l)(1+\epsilon) \le Opt(\alpha)(1+\epsilon) = Opt(\alpha) + \epsilon \cdot Opt(\alpha)$$

Now we are ready to describe an approximation algorithm for the $c$-Makespan problem: given an input instance $\alpha = \langle t_1, \ldots, t_n \rangle$ for the $c$-Makespan problem, (1) construct, in time $O(1/\epsilon^{2c-1})$, a scheduling $\mathcal{S}_l$ of parallel completion time bounded by $Opt(\alpha) + \epsilon \cdot Opt(\alpha)$ for the set $\alpha_l$ of large jobs; (2) assign the small jobs by the greedy method, i.e., we assign each small job to the most lightly loaded processor as described in algorithm **Graham-Schedule** (see Figure 5.1). The assignment of the small jobs can be easily done in time $O(n)$ (note that the number $c$ of processors is a fixed constant and that the number of small jobs is bounded by $n$). Thus, the running time of this approximation algorithm is bounded by $O(n + 1/\epsilon^{2c-1})$. We claim that this algorithm has an approximation ratio bounded by $1+\epsilon$. Let $Apx(\alpha)$ be the parallel completion time for the scheduling constructed by this algorithm.

Suppose that processor $d$ has the longest running time $T_d = Apx(\alpha)$. Consider the last job $t_i$ assigned to processor $d$. If $t_i$ is a large job, then the processor $d$ is assigned no small jobs. Thus, $T_d$ is the parallel completion time of the scheduling $\mathcal{S}_l$ for the large jobs $\alpha_l$. By the above analysis,

$$Apx(\alpha) = T_d = Apx(\alpha_l) \le Opt(\alpha)(1+\epsilon)$$

On the other hand, if $t_i$ is a small job, then $t_i < \epsilon T_0/c$ and by the greedy method, all $c$ processors have running time at least $T_d - t_i$. Therefore, $\sum_{i=1}^n t_i \ge c(T_d - t_i)$ and $Opt(\alpha) \ge T_d - t_i$. This gives (note $T_0/c \le Opt(\alpha)$)

$$Apx(\alpha) = T_d \le Opt(\alpha) + t_i \le Opt(\alpha) + \epsilon T_0/c \le Opt(\alpha)(1+\epsilon)$$

Therefore, in any case, the ratio $Apx(\alpha)/Opt(\alpha)$ is bounded by $1+\epsilon$.

We summarize the above discussion into the following theorem.

**Theorem 6.3.5** *There is a fully polynomial time approximation scheme for the c-*MAKESPAN *problem that, given an input instance $\alpha$ and an $\epsilon > 0$, constructs a scheduling for $\alpha$ of approximation ratio bounded by $1+\epsilon$ in time bounded by $O(n + 1/\epsilon^{2c-1})$.*

Note that the fully polynomial time approximation scheme for the $c$-MAKESPAN problem in Theorem 6.3.5 runs in linear time when the error bound $\epsilon$ is fixed.

The technique can also be applied to the KNAPSACK problem. With a more complex analysis, it can be shown that there is a fully polynomial time approximation scheme for the KNAPSACK problem that, given an input instance $\alpha$ for the KNAPSACK problem and an $\epsilon > 0$, constructs a subset of $\alpha$ of size bounded by $B$ and value at least $Opt(\alpha)/(1+\epsilon)$ in time $O(n/\epsilon^2)$.

Further improvements on the KNAPSACK problem are possible. For example, with a more careful treatment of the large and small items, one can develop a fully polynomial time approximation scheme for the KNAPSACK problem of running time $O(n/\log(1/\epsilon) + 1/\epsilon^4)$. Interested readers are referred to [88] for detailed discussions.

## 6.4 Which problems have no FPTAS?

Fully polynomial time approximation schemes seem the best we can expect for NP-hard optimization problems. An NP-hard optimization problem with a fully polynomial time approximation scheme can be approximated to a ratio $1 + \epsilon$ for any $\epsilon > 0$ within a reasonable computational time, which is bounded by a polynomial of the input length and $1/\epsilon$. We have seen that several NP-hard optimization problems, such as the KNAPSACK problem and the $c$-MAKESPAN problem, have fully polynomial time approximation schemes.

A natural question is whether every NP-hard optimization problem has a fully polynomial time approximation scheme. If not, how do we determine whether a given NP-hard optimization problem has a fully polynomial time approximation scheme. We discuss this problem in this section.

**Definition 6.4.1** Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem. For each input instance $x \in I_Q$, define $Opt_Q(x) = opt_Q\{f_Q(x,y) | y \in S_Q(x)\}$. That is, $Opt_Q(x)$ is the value of the objective function $f_Q$ on input instance $x$ and an optimal solution to $x$.

The following theorem provides a very convenient sufficient condition for an NP-hard optimization problem to have no fully polynomial time approximation schemes.

**Theorem 6.4.1** *Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem. If there is a fixed polynomial p such that for all input instances $x \in I_Q$, $Opt_Q(x)$ is bounded by $p(|x|)$, then Q does not have a fully polynomial time approximation scheme unless Q can be precisely solved in polynomial-time.*

PROOF. Let $A$ be an approximation algorithm that is a fully polynomial time approximation scheme for the optimization problem $Q$. We show that $Q$ can be precisely solved in polynomial time.

By the definition, we can suppose that the running time of $A$ is bounded by $O(n^c/\epsilon^d)$, where $c$ and $d$ are fixed constants. Moreover, by the condition given in the theorem, we can assume that $Opt_Q(x) \leq n^h$, where $h$ is also a fixed constant.

First assume that $opt_Q = \min$. For an input instance $x \in I_Q$, let $A(x)$ be the objective function value on the input $x$ and the solution to $x$ constructed by the algorithm $A$. Thus, we know that for any $\epsilon > 0$, the algorithm $A$ constructs in time $O(n^c/\epsilon^d)$ a solution with approximation ratio $A(x)/Opt(x) \leq 1 + \epsilon$. Also note that $A(x)/Opt(x) \geq 1$.

Now, let $\epsilon = 1/n^{h+1}$, then the algorithm $A$ constructs a solution with approximation ratio bounded by

$$1 \leq \frac{A(x)}{Opt(x)} \leq 1 + \frac{1}{n^{h+1}}$$

which gives

$$Opt(x) \leq A(x) \leq Opt(x) + Opt(x)/n^{h+1}$$

Since both $Opt(x)$ and $A(x)$ are integers, and $Opt(x) \leq n^h$ implies that $Opt(x)/n^{h+1}$ is a number strictly less than 1, we conclude that

$$Opt(x) = A(x)$$

That is, the algorithm $A$ actually constructs an optimal solution to the input instance $x$. Moreover, the running time of $A$ is bounded by $O(n^c/(1/n^{h+1})^d) = O(n^{c+hd+d})$, which is a polynomial of $n$.

The case that $opt_Q = \max$ can be proved similarly. Note that in this case, we should also have $A(x) \leq n^h$. Thus, in time $O(n^c/(1/n^{h+1})^d) =$

$O(n^{c+hd+d})$, the algorithm $A$ constructs a solution to $x$ with the value $A(x)$ such that

$$1 \le \frac{Opt(x)}{A(x)} \le 1 + \frac{1}{n^{h+1}}$$

which gives

$$A(x) \le Opt(x) \le A(x) + A(x)/n^{h+1}$$

Now since $A(x)/n^{h+1} < 1$, we conclude $Opt(x) = A(x)$. $\square$

In particular, Theorem 6.4.1 says that if $Opt_Q(x)$ is bounded by a polynomial of the input length $|x|$ and $Q$ is known to be NP-hard, then $Q$ does not have a fully polynomial time approximation scheme unless P = NP.

Theorem 6.4.1 is actually very powerful. Most NP-hard optimization problems satisfy the condition stated in the theorem, thus we can derive directly that these problems have no fully polynomial time approximation scheme. We will give a few examples below to illustrate the power of Theorem 6.4.1.

Consider the following problem:

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 $S$

$opt_Q$: max

It is easy to apply Theorem 6.4.1 to show that the INDEPENDENT SET problem has no fully polynomial time approximation scheme. In fact, the value of the objective function is bounded by the number of vertices in the input graph $G$, which is certainly bounded by a polynomial of the input length $|G|$.

There are many other graph problems (actually, most graph problems) like the INDEPENDENT SET problem that ask to optimize the size of a subset of vertices or edges of the input graph satisfying certain given properties. For all these problems, we can conclude directly from Theorem 6.4.1 that they do not have a fully polynomial time approximation scheme unless they can be solved precisely in polynomial time.

Let us consider another example of a problem for which no fully polynomial time approximation scheme exists.

BIN PACKING

INPUT: $\langle t_1, t_2, \ldots, t_n; B\rangle$, all integers and $t_i \le B$ for all $i$

OUTPUT: a packing of the $n$ objects of size $t_1, \ldots, t_n$ into the minimum number of bins of size $B$

It is pretty easy to prove that the NP-complete problem PARTITION is polynomial time reducible to the BIN PACKING problem (see Chapter 7 for more detailed discussions). Thus, the BIN PACKING problem is NP-hard. The BIN PACKING problem can be interpreted as a scheduling problem in which $n$ jobs of processing time $t_1, \ldots, t_n$ are given, the parallel completion time $B$ is fixed, and we are looking for a scheduling of the jobs so that the number of processors used in the scheduling is minimized. Since $t_i \le B$ for all $i$, we know that at most $n$ bins are needed to pack the $n$ objects. Thus, $Opt(x) \le n$ for all input instances $x$ of $n$ objects. By Theorem 6.4.1, we conclude directly that the BIN PACKING problem has no fully polynomial time approximation scheme unless P = NP.

What if the condition of Theorem 6.4.1 does not hold? Can we still derive a conclusion of nonexistence of a fully polynomial time approximation scheme for an optimization problem? We study this problem starting with the famous TRAVELING SALESMAN problem, and will derive general rules for this kind of optimization problems.

TRAVELING SALESMAN

INPUT: a weighted complete graph $G$

OUTPUT: a simple cycle containing all vertices of $G$ (such a simple cycle is called a *traveling salesman tour*) and the weight of the cycle is minimized

The TRAVELING SALESMAN problem obviously does not satisfy the condition stated in Theorem 6.4.1. For example, if all edges of the input graph $G$ of $n$ vertices have weight of order $\Theta(2^n)$, then the weight of the minimum traveling salesman tour is $\Omega(n2^n)$ while a binary representation of the input graph $G$ has length bounded by $O(n^3)$ (note that the length of the binary representation of a number of order $\Theta(2^n)$ is $O(n)$ and $G$ has $O(n^2)$ edges). Therefore, Theorem 6.4.1 does not apply to the TRAVELING SALESMAN problem.

To show the non-approximability of the TRAVELING SALESMAN problem, we first consider a simpler version of the TRAVELING SALESMAN problem, which is defined as follows.

TRAVELING SALESMAN 1-2

INPUT: a weighted complete graph $G$ such that the weight of each
edge of $G$ is either 1 or 2

OUTPUT: a traveling salesman tour of minimum weight

**Lemma 6.4.2** *The* TRAVELING SALESMAN 1-2 *problem is NP-hard.*

PROOF. We show that the well-known NP-complete problem HAMILTONIAN CIRCUIT can be polynomial time reducible to the TRAVELING SALESMAN 1-2 problem.

By the definition, for each undirected unweighted graph $G$ of $n$ vertices, the HAMILTONIAN CIRCUIT problem asks if $G$ contains a Hamiltonian circuit, i.e., a simple cycle of length $n$ (for more discussion of the problem, the reader is referred to [50]).

Given an input instance $G = (V, E)$ for the HAMILTONIAN CIRCUIT problem, we add edges to $G$ to make a weighted complete graph $G' = (V, E \cup E')$ such that for each edge $e \in E$ of $G'$ that is in the original graph $G$, we assign a weight 1 and for each edge $e' \in E'$ of $G'$ that is not in the original graph $G$, we assign a weight 2. The graph $G'$ is certainly an input instance of the TRAVELING SALESMAN 1-2 problem. Now, let $T$ be a minimum weighted traveling salesman tour in $G'$. It is easy to verify that the weight of $T$ is equal to $n$ if and only if the original graph $G$ contains a Hamiltonian circuit.

This completes the proof. $\square$

Theorem 6.4.1 can apply to the TRAVELING SALESMAN 1-2 problem directly.

**Lemma 6.4.3** *The* TRAVELING SALESMAN 1-2 *problem has no fully polynomial time approximation scheme unless P = NP.*

PROOF. Since the weight of a traveling salesman tour for an input instance $G$ of the TRAVELING SALESMAN 1-2 problem is at most $2n$, assuming that $G$ has $n$ vertices, the condition stated in Theorem 6.4.1 is satisfied by the TRAVELING SALESMAN 1-2 problem. Now the theorem follows from Theorem 6.4.1 and Lemma 6.4.2. $\square$

Now we are ready for a conclusion on the approximability of the TRAVELING SALESMAN problem in its general form.

**Theorem 6.4.4** *The* TRAVELING SALESMAN *problem has no fully polynomial time approximation scheme unless* $P = NP$.

PROOF. Since each input instance for the TRAVELING SALESMAN 1-2 problem is also an input instance for the TRAVELING SALESMAN problem, a fully polynomial time approximation scheme for the TRAVELING SALESMAN problem should also be a fully polynomial time approximation scheme for the TRAVELING SALESMAN 1-2 problem. Now the theorem follows from Lemma 6.4.3. □

Theorem 6.4.4 illustrates a general technique for proving the nonexistence of fully polynomial time approximation schemes for an NP-hard optimization problem when Theorem 6.4.1 is not applicable. We formulate it as follows.

Let $Q = (I_Q, S_Q, f_Q, opt_Q)$ be an optimization problem. Recall that for each input instance $x$ of $Q$, $\mathrm{length}(x)$ is the length of a binary representation of $x$ and $\max(x)$ is the largest number that appears in $x$.

**Definition 6.4.2** Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem and let $q$ be any function. A subproblem $Q'$ of $Q$ is a *$Q_q$-subproblem* if $Q' = \langle I'_Q, S_Q, f_Q, opt_Q \rangle$ such that $I'_Q \subseteq I_Q$ and for all $x \in I'_Q$, $\max(x) \leq q(\mathrm{length}(x))$.

The following definition was first introduced and studied by Garey and Johnson [50].

**Definition 6.4.3** An optimization problem $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ is *NP-hard in the strong sense* if a $Q_q$-subproblem of $Q$ is NP-hard for some polynomial $q$.

The concept of the strong NP-hardness can be naturally extended to decision problems.

The TRAVELING SALESMAN problem is an example of optimization problems that are NP-hard in the strong sense, as shown by the following theorem.

**Theorem 6.4.5** *The* TRAVELING SALESMAN *problem is NP-hard in the strong sense.*

PROOF. If we denote by $Q$ the TRAVELING SALESMAN problem, then the TRAVELING SALESMAN 1-2 problem is a $Q_2$-subproblem of $Q$. By

Lemma 6.4.2, the TRAVELING SALESMAN 1-2 problem is NP-hard. Now by the above definition, the TRAVELING SALESMAN problem is NP-hard in the strong sense. □

If the condition $\max(x) \leq p(\text{lengh}(x))$ for some fixed polynomial $p$ is satisfied for all input instances $x$ of an NP-hard optimization problem $Q$, then $Q$ is NP-hard in the strong sense. Note that for many NP-hard optimization problems, in particular for many NP-hard optimization problems for which the condition of Theorem 6.4.1 is satisfied, the condition $\max(x) \leq p(\text{lengh}(x))$ is satisfied trivially. Thus, these NP-hard optimization problems are also NP-hard in the strong sense. On the other hand, there are many other NP-hard optimization problems that are not NP-hard in the strong sense. In particular, we have the following theorem.

**Theorem 6.4.6** *If an NP-hard optimization problem $Q$ is solvable in pseudo-polynomial time, then $Q$ is not NP-hard in the strong sense unless $P = NP$.*

PROOF. Suppose that $Q$ is solvable in pseudo-polynomial time. Let $A$ be an algorithm such that for an input instance $x$ of $Q$, the algorithm $A$ constructs an optimal solution to $x$ in time $O((\text{length}(x))^c(\max(x))^d)$ for some constants $c$ and $d$.

If $Q$ is NP-hard in the strong sense, then there is a $Q_p$-subproblem $Q'$ of $Q$ for some fixed polynomial $p$ such that $Q'$ is also NP-hard. However, for all input instances $x$ of $Q'$, $\max(x) \leq p((length)(x))$. Thus, the algorithm $A$ constructs an optimal solution for each input instance $x$ of $Q'$ in time

$$O((\text{length}(x))^c(\max(x))^d) = O((\text{length}(x))^c(p(\text{length}(x)))^d)$$

which is bounded by a polynomial of $\text{length}(x)$. Thus, the NP-hard optimization problem $Q'$ can be solved by the polynomial time algorithm $A$, which implies P = NP. □

In particular, Theorem 6.4.6 combined with Theorem 6.1.3 and Theorem 6.1.4 gives

**Corollary 6.4.7** *The* KNAPSACK *problem and the $c$-*MAKESPAN *problem are not NP-hard in the strong sense unless $P = NP$.*

The following theorem serves as a fundamental theorem for proving the nonexistence of fully polynomial time approximation schemes for an NP-hard optimization problem, in particular when Theorem 6.4.1 is not applicable. We say that a two-parameter function $f(x, y)$ is a polynomial of $x$

and $y$ if $f(x, y)$ can be written as a finite sum of the terms of form $x^c y^d$, where $c$ and $d$ are non-negative integers.

**Theorem 6.4.8** *Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem that is NP-hard in the strong sense. Suppose that for all input instances $x$ of $Q$, $Opt_Q(x)$ is bounded by a polynomial of* length$(x)$ *and* max$(x)$*. Then $Q$ has no fully polynomial time approximation scheme unless P = NP.*

PROOF. The proof of this theorem is very similar to the discussion we have given for the TRAVELING SALESMAN problem.

Since $Q$ is NP-hard in the strong sense, a $Q_q$-subproblem $Q'$ is NP-hard for a polynomial $q$. Let $Q' = \langle I'_Q, S_Q, f_Q, opt_Q \rangle$. Then for each input instance $x \in I'_Q$, we have $\max(x) \leq q(\text{length}(x))$. Combining this condition with the condition stated in the theorem that $Opt_Q(x)$ is bounded by a polynomial of length$(x)$ and max$(x)$, we derive that $Opt_Q(x)$ is bounded by a polynomial of length$(x)$ for all input instances $x \in I'_Q$. Now by Theorem 6.4.1, the problem $Q'$ has no fully polynomial time approximation scheme unless P = NP. Since each input instance of $Q'$ is also an input instance of $Q$, a fully polynomial time approximation scheme for $Q$ is also a fully polynomial time approximation scheme for $Q'$. Now the theorem follows. □

**Remark.** How common is the situation that $Opt_Q(x)$ is bounded by a polynomial of length$(x)$, and max$(x)$? In fact, this situation is fairly common because for most optimization problems, the objective function value is defined through additions or constant number of multiplications on the numbers appearing in the input instance $x$, which is certainly bounded by a polynomial of length$(x)$ and max$(x)$. Of course, the condition is not universely true for general optimization problems. For example, an objective function can be simply defined to be the exponentiation of the sum of a subset of input values, which cannot be bounded by any polynomial of length$(x)$ and max$(x)$.

In general, it is easy to verify the condition that $Opt_Q(x)$ is bounded by a polynomial of length$(x)$ and max$(x)$. Therefore, in order to apply Theorem 6.4.8, we need to prove the strong NP-hardness for a given optimization problem $Q$. There are two general techniques serving for this purpose. The first one is to pick an NP-complete problem $L$ and show that $L$ is polynomial time reducible to a $Q_q$-subproblem of $Q$ for some polynomial $q$. Our polynomial time reduction from the HAMILTONIAN CIRCUIT problem to the TRAVELING SALESMAN 1-2 problem, which leads to the strong NP-hardness of the general TRAVELING SALESMAN problem (Theorem 6.4.5), well illustrates this technique.

The second technique is to develop a polynomial time reduction from a known strongly NP-hard optimization problem $R$ to the given optimization problem $Q$. For this, we also require that for each polynomial $p$, the reduction transforms a $R_p$-subproblem of $R$ into a $Q_q$-subproblem of $Q$ for some polynomial $q$. We explain this technique by showing that the following familiar optimization problem is NP-hard in the strong sense.

MAKESPAN

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

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

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

$opt_Q$: min

To show that the MAKESPAN problem is NP-hard in the strong sense, we reduce the following strongly NP-hard (decision) problem to the MAKESPAN problem.

THREE-PARTITION

INPUT: $\{t_1, t_2, \ldots, t_{3m}; B\}$, all integers, where $B/4 \le t_i \le B/2$ for all $i$, and $\sum_{i=1}^{3m} t_i = mB$

QUESTION: Can $\{t_1, \ldots, t_{3m}\}$ be partitioned into $m$ sets, each of size $B$?

The THREE-PARTITION problem has played a fundamental role in proving strong NP-hardness for many scheduling problems. For a proof that the THREE-PARTITION problem is NP-hard in the strong sense, the reader is referred to Garey and Johnson's authoritative book [50], Section 4.2.2.

The reduction $\mathcal{R}$ from the THREE-PARTITION problem to the MAKESPAN problem is straightforward: given an input instance $\alpha = \{t_1, t_2, \ldots, t_{3m}; B\}$ of the THREE-PARTITION problem, we construct an input instance $\beta = \{t_1, t_2, \ldots, t_{3m}; m\}$ for the MAKESPAN problem. It is clear that the optimal parallel completion time for the instance $\beta$ for the MAKESPAN problem is

$B$ if and only if $\alpha$ is a yes-instance for the THREE-PARTITION problem. Note that the input length length($\beta$) is at least 1/2 times the input length length($\alpha$) (in fact, length($\alpha$) and length($\beta$) are roughly equal), and that $\max(\beta)$ is bounded by $\max(\alpha) + \text{length}(\alpha)$.

Since the THREE-PARTITION problem is NP-hard in the strong sense, there is a polynomial $q$ such that a (THREE-PARTITION)$_q$-subproblem $R'$ of the THREE-PARTITION problem is NP-hard. Let $Q'$ be a subproblem of the MAKESPAN problem such that $Q'$ consists of input instances of the form $\{t_1, t_2, \ldots, t_{3m}; m\}$, where $\sum_{i=1}^{3m} t_i = mB$ and $\{t_1, t_2, \ldots, t_{3m}; B\}$ is an instance of $R'$. Therefore, the polynomial time reduction $\mathcal{R}$ reduces the problem $R'$ to the problem $Q'$. Therefore, the problem $Q'$ is NP-hard. Moreover, for each instance $\alpha$ of $R'$, we have $\max(\alpha) \leq q(\text{length}(\alpha))$. Now for each instance $\beta$ of $Q'$ that is the image of an instance $\alpha$ of $R'$ under the reduction $\mathcal{R}$, we have

$$\begin{aligned} \max(\beta) &\leq \max(\alpha) + \text{length}(\alpha) \leq q(\text{length}(\alpha)) + \text{length}(\alpha) \\ &\leq q(2\text{length}(\beta)) + 2\text{length}(\beta) \end{aligned}$$

Therefore, the NP-hard problem $Q'$ is a (MAKESPAN)$_p$-subproblem of the MAKESPAN problem, where $p$ is a polynomial. This proves that the MAKESPAN problem is NP-hard in the strong sense.

It is trivial to verify that the other conditions of Theorem 6.4.8 are satisfied by the MAKESPAN problem. Thus,

**Theorem 6.4.9** *The* MAKESPAN *problem is NP-hard in the strong sense. Moreover, the* MAKESPAN *problem has no fully polynomial time approximation scheme unless P = NP.*

We should point out that the $c$-MAKESPAN problem, i.e., the MAKESPAN problem with the number of processors is fixed by a constant $c$, has a fully polynomial time approximation scheme (Corollary 6.2.3). However, if the number $m$ of processors is given as a variable in the input, then the problem becomes NP-hard in the strong sense and has no fully polynomial time approximation scheme.

# Chapter 7

# Asymptotic Approximation Schemes

Using Theorem 6.4.1, it is fairly easy to show that certain NP-hard optimization problems have no fully polynomial time approximation scheme. In fact, for some NP-hard optimization problems, such as the Bin Packing problem and the Graph Edge Coloring problem we are studying in this chapter, it is also possible to prove that there is a constant $c > 1$ such that these problems have no polynomial time approximation algorithm of approximation ratio smaller than $c$. The proofs for the nonapproximability of these problems are based on the observation that even when the value of optimal solutions is very small, the problems still remain NP-hard. In consequence, no polynomial time approximation algorithm is expected that can return an optimal solution when a given input instance has small optimal solution value, while in this case a small approximation error may significantly effect the approximation ratio. Therefore in terms of the regular approximation ratio, which is defined as the worst case ratio of optimal solution value and the approximation solution value over all kinds of input instances, these optimization problems cannot have polynomial time approximation algorithm of ratio arbitrarily close to 1.

On the other hand, we find out that for some of these optimization problems, it is possible to develop polynomial time approximation algorithms whose approximation ratio becomes arbitrarily close to 1 *when the optimal solution value is large enough.* Therefore, here we are interested in the *asymptotic behavior* of the approximation ratio in terms of the optimal solution values.

In this chapter, we discuss the asymptotic approximability for optimization problems. Two optimization problems are considered: the BIN PACKING problem and the GRAPH EDGE COLORING problem. Since both these two problems are minimization problems, our definitions and discussions are based on minimization problems only. However, it should be straightforward to extend these definitions and discussions to maximization problems.

As before, for any input instance $x$ of an optimization problem, we denote by $Opt(x)$ the value of optimal solutions to $x$.

**Definition 7.0.4** Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be a minimization problem and let $A$ be an approximation algorithm for $Q$. The *asymptotic approximation ratio* of $A$ is bounded by $r_0$ if for any $r > r_0$, there is an integer $N$ such that for any input instance $x$ of $Q$ with $Opt(x) \geq N$, the algorithm $A$ constructs a solution $y$ to $x$ satisfying $f_Q(x, y)/Opt(x) \leq r$.

Analogous to fully polynomial time approximation scheme defined in terms of approximation ratio of an approximation algorithm, asymptotic fully polynomial time approximation scheme of an optimization problem can be defined in terms of the asymptotic approximation ratio of approximation algorithms.

**Definition 7.0.5** An optimization problem $Q$ has an *asymptotic fully polynomial time approximation scheme* (AFPTAS) if it has a family of approximation algorithms $\{A_\epsilon \mid \epsilon > 0\}$ such that for any $\epsilon > 0$, $A_\epsilon$ is an approximation algorithm for $Q$ of asymptotic approximation ratio bounded by $1 + \epsilon$ and of running time bounded by a polynomial of the input length and $1/\epsilon$.

## 7.1 The Bin Packing problem

The BIN PACKING problem is to pack items of various sizes into bins of a fixed size so that the number of bins used is minimized. The BIN PACKING problem has many potential practical applications, from loading trucks subject to weight limitations to packing television commercials into station breaks. Moreover, the BIN PACKING problem has been of fundamental theoretical significance, serving as an early proving ground for many of the classical approaches to analyzing the performance of approximation algorithms for optimization problems.

In this section, we study a number of well-known approximation algorithms for the BIN PACKING problem. We start with the famous "First Fit" algorithm and show that if the items are sorted in a nonincreasing order of their sizes, then the algorithm achieves the optimal approximation ratio 1.5. Then we study the asymptotic approximation ratio for the BIN PACKING problem. We show in detail that for any $\epsilon > 0$, there is a polynomial time approximation algorithm $A_\epsilon$ for the BIN PACKING problem such that the asymptotic approximation ratio of $A_\epsilon$ is bounded by $1 + \epsilon$. We then explain how this algorithm can be converted into an asymptotic fully polynomial time approximation scheme for the BIN PACKING problem.

### 7.1.1 Preliminaries and simple algorithms

We start with the formal definition of the BIN PACKING problem.

> BIN PACKING $= \langle I_Q, S_Q, f_Q, opt_Q \rangle$
>
> $I_Q$: the set of tuples $\alpha = \langle s_1, \ldots, s_n; T \rangle$, where $s_i$ and $T$ are positive integers;
>
> $S_Q(\alpha)$: the set of partitions $Y = (B_1, \ldots, B_r)$ of $\{s_1, \ldots, s_n\}$ such that $\sum_{s_i \in B_j} s_i \leq T$ for all $j$;
>
> $f_Q(\alpha, Y)$: the number of subsets in the partition $Y$ of $\alpha$;
>
> $opt_Q = \min$

In other words, the BIN PACKING problem is to pack $n$ given items of size $s_1$, …, $s_n$ into the minimum number of bins of size $T$.

If any item has size $s_i$ larger than $T$, then obviously there is no way to pack the items into bins of size $T$. Therefore, we will always assume that all items have size less than or equal to the bin size.

**Lemma 7.1.1** *The* BIN PACKING *problem is NP-hard.*

PROOF. We show that the NP-complete decision problem PARTITION is polynomial time reducible to the BIN PACKING problem.

Recall that the PARTITION problem is defined as follows: given a set $S = \{a_1, \ldots, a_n\}$ of $n$ integers, can $S$ be partitioned into two disjoint subsets $S_1$ and $S_2$ such that $\sum_{a_i \in S_1} a_i = \sum_{a_j \in S_2} a_j$?

Given an instance $S = \{a_1, \ldots, a_n\}$ of the PARTITION problem, let $T = \sum_{i=1}^{n} a_i$. If $T$ is an odd number, we construct an instance $\alpha = \langle T, T, T; T \rangle$ for the BIN PACKING problem (in this case, $S$ is a no-instance for the PARTITION problem and an optimal solution to the instance $\alpha$ of the BIN PACKING problem uses three bins). If $T$ is an even number, we construct an instance $\alpha = \langle a_1, \ldots, a_n; T/2 \rangle$ for the BIN PACKING problem. Now it is very easy to see that $S$ is a yes-instance for the PARTITION problem if and only if an optimal solution to the instance $\alpha$ of the Bin Packing problem uses two bins. □

It is easy to verify that the Bin Packing problem satisfies the conditions in Theorem 6.4.1. Therefore, the Bin Packing problem has no fully polynomial time approximation scheme unless P = NP.

In fact, a stronger lower bound on the approximation ratios of approximation algorithms for the Bin Packing problem can be derived. Observe that in the proof of Lemma 7.1.1, we actually proved that deciding if the value of optimal solutions to an instance $\alpha$ of the Bin Packing problem is 2 is NP-hard. Therefore, any polynomial time approximation algorithm for the Bin Packing problem can only guarantee a packing of at least three bins for the instances whose optimal solution value is 2. This leads to the following theorem.

**Theorem 7.1.2** *There is no polynomial time approximation algorithm for the* Bin Packing *problem with approximation ratio less than 1.5 unless P = NP.*

Proof. Suppose that we have a polynomial time approximation algorithm $A$ with approximation ratio less than 1.5 for the Bin Packing problem. We show how we can use this algorithm to solve the NP-complete problem Partition in polynomial time.

Given an input instance $S = \{a_1, \cdots, a_n\}$ for the Partition problem, if $\sum_{i=1}^n a_i$ is an odd number, then we know $S$ is a no-instance. Otherwise, let $T = (\sum_{i=1}^n a_i)/2$, and let $\alpha = \langle a_1, a_2, \cdots, a_n; T\rangle$ be an instance for the Bin Packing problem. Now apply the approximation algorithm $A$ for the Bin Packing problem on the instance $\alpha$. Suppose that the approximation algorithm $A$ uses $m$ bins for this input instance $\alpha$. There are two different cases.

If $m \geq 3$, then since we have

$$m/Opt(\alpha) < 1.5$$

we get $Opt(\alpha) > 2$. That is, the items $a_1, \ldots, a_n$ cannot be packed into two bins of size $T = (\sum_{i=1}^n a_i)/2$. Consequently, the instance $S = \{a_1, \cdots, a_n\}$ is a no-instance for the Partition problem.

On the other hand, if $m \leq 2$, then we must have $m = 2$. Thus, the items $a_1, \ldots, a_h$ can be evenly split into two sets of equal size $T = (\sum_{i=1}^n a_i)/2$. That is, the instance $S = \{a_1, \cdots, a_n\}$ is a yes-instance for the Partition problem.

> **Algorithm. First-Fit**
>
> Input: $\alpha = \langle s_1, s_2, \ldots, s_n; T \rangle$
> Output: a packing of the items $s_1, \ldots, s_n$ into bins of size $T$
>
> 1. suppose that all bins $B_1, B_2, \ldots$, are empty;
> 2. **for** $i = 1$ **to** $n$ **do** put the item $s_i$ in the first bin it fits.

Figure 7.1: The **First-Fit** algorithm

Therefore, the instance $S$ is a yes-instance for the PARTITION problem if and only if the approximation algorithm $A$ uses two bins to pack the instance $\alpha$. Since by our assumption, the approximation algorithm $A$ runs in polynomial time, we conclude that the PARTITION problem can be solved in polynomial time.

Since the PARTITION problem is NP-complete, this implies P = NP. The theorem is proved. □

The technique used in Theorem 7.1.2 can be used to derive lower bounds on approximation ratios for other optimization problems. In general, suppose that $Q$ is an NP-hard minimization problem for which deciding whether an instance $x$ of $Q$ has optimal value $c$ is NP-hard. Then we can derive directly that no polynomial time approximation algorithm for $Q$ can have approximation ratio smaller than $(c+1)/c$.

Now we consider approximation algorithms for the BIN PACKING problem. The simple algorithm **First-Fit** given in Figure 7.1, which is based on the greedy method, has been well-known.

The **for** loop in step 2 of the algorithm **First-Fit** is executed $n$ times, and in each execution of the loop, we go through the bins $B_1, B_2, \ldots$, to find a bin that fits the current item. Since the number of bins used cannot be larger than the number of items in the input, we conclude that the algorithm **First-Fit** runs in time $O(n^2)$.

Now we analyze the approximation ratio for the algorithm **First-Fit**.

**Theorem 7.1.3** *The algorithm* **First-Fit** *has approximation ratio 2.*

PROOF. We observe that there is at most one used bin whose content is not larger than $T/2$. In fact, suppose that there are two used bins $B_i$ and $B_j$ whose contents are bounded by $T/2$. Without loss of generality, let $i < j$. Then the algorithm **First-Fit** would have put the items in the bin $B_j$ into the bin $B_i$ since the bin $B_i$ has enough room for these items and the bin $B_i$ is considered before the bin $B_j$ by the algorithm **First-Fit**.

Now the theorem can be proved in two cases.

Suppose that the contents of all used bins are not less than $T/2$. Let $m$ be the number of bins used by the algorithm **First-Fit**. We have

$$\sum_{i=1}^{n} s_i \geq m(T/2)$$

Since the bin size is $T$, we need at least

$$\lceil (\sum_{i=1}^{n} s_i)/T \rceil \geq \lceil (mT)/(2T) \rceil \geq m/2$$

bins to pack the $n$ items, i.e., $Opt(\alpha) \geq m/2$. Therefore, the approximation ratio is bounded in this case by

$$\frac{m}{Opt(\alpha)} \leq \frac{m}{m/2} = 2$$

Now suppose that there is a used bin whose content $x$ is less than $T/2$. Again let $m$ be the number of bins used by the algorithm **First-Fit**. Therefore, there are $m-1$ bins with contents at least $T/2$. This gives

$$\sum_{i=1}^{n} s_i \geq \frac{(m-1)T}{2} + x > \frac{(m-1)T}{2}$$

Thus, $\lceil (\sum_{i=1}^{n} s_i)/T \rceil > (m-1)/2$

If $m-1$ is an even number, then since both $\lceil (\sum_{i=1}^{n} s_i)/T \rceil$ and $(m-1)/2$ are integers, we get

$$\lceil (\sum_{i=1}^{n} s_i)/T \rceil \geq (m-1)/2 + 1 > m/2$$

If $m-1$ is an odd number, then

$$\lceil (\sum_{i=1}^{n} s_i)/T \rceil \geq \lceil (m-1)/2 \rceil = (m-1)/2 + 1/2 = m/2$$

Note that any packing should use at least $\lceil (\sum_{i=1}^{n} s_i)/T \rceil$ bins. In particular,

$$Opt(\alpha) \geq \lceil (\sum_{i=1}^{n} s_i)/T \rceil$$

**Algorithm. First-Fit-Decreasing**

Input: $\alpha = \langle s_1, s_2, \ldots, s_n; T \rangle$
Output: a packing of the items $s_1, \ldots, s_n$ into bins of size $T$

1. suppose that all bins $B_1, B_2, \ldots,$ are empty;
2. sort the items $s_1, \ldots, s_n$ in non-increasing order,
   let the sorted list be $L = \{s'_1, \ldots, s'_n\}$;
3. **for** $i = 1$ **to** $n$ **do** put the item $s'_i$ in the first bin it fits.

Figure 7.2: The **First-Fit-Decreasing** algorithm

The above analysis shows that the approximation ratio is bounded by

$$\frac{m}{Opt(\alpha)} \leq \frac{m}{\lceil (\sum_{i=1}^{n} s_i)/T \rceil} \leq \frac{m}{m/2} = 2$$

This proves the theorem. $\square$

The bound 2 in Theorem 7.1.3 is not tight. With a more involved proof [48], one can show that for any instance $\alpha$ of the Bin Packing problem, the algorithm **First-Fit** packs $\alpha$ using no more than $\lceil (17/10) Opt(\alpha) \rceil$ bins. Moreover, instances $\beta$ of the Bin Packing problem have been constructed for which the algorithm **First-Fit** uses more than $(17/10)Opt(\beta) - 2$ bins [73]. Thus, roughly speaking, the approximation ratio of the algorithm **First-Fit** is 1.7.

A better approximation algorithm is obtained by observing that the worst performance for the algorithm **First-Fit** seems to occur when small items are put before large items. For example, suppose that we have four items of size $T/3$, $T/3$, $2T/3$, and $2T/3$, respectively. If we put the two smaller items of size $T/3$ first into a bin, then no larger items of size $2T/3$ can fit into that bin. Thus each larger item will need a new bin, resulting in a packing of three bins. On the other hand, if we put the two larger items of size $2T/3$ into two bins first, then the two smaller items of size $T/3$ can also fit into these two bins, resulting in a packing of two bins. Based on this observation, we first sort the items by their sizes in non-increasing order before we apply the algorithm **First-Fit** on them. The modified algorithm, called **First-Fit-Decreasing** algorithm, is given in Figure 7.2, which is also a well-known approximation algorithm for the Bin Packing problem.

Since the sorting can be done in time $O(n \log n)$, the running time of the algorithm **First-Fit-Decreasing** is also bounded by $O(n^2)$.

To analyze the approximation ratio for the algorithm **First-Fit-Decreasing**, we start with the following lemma.

**Lemma 7.1.4** *Let $\alpha = \langle s_1, \ldots, s_n; T\rangle$ be an instance of the* Bin Packing *problem. Suppose that the algorithm* **First-Fit-Decreasing** *uses $r$ bins $B_1$, $B_2$, …, $B_r$ to pack $s_1$, …, $s_n$. Then for any bin $B_j$ with $j > Opt(\alpha)$, every item in the bin $B_j$ has size at most $T/3$.*

Proof. To simplify expressions, we let $m = Opt(\alpha)$ be the number of bins used by an optimal solution to the instance $\alpha$. If $m = r$, then there is nothing to prove. Thus, we assume that $m < r$.

The algorithm **First-Fit-Decreasing** first sorts $s_1$, $s_2$, …, $s_n$ into a non-increasing list $L = \{s'_1, s'_2, \ldots, s'_n\}$. To prove the lemma, it suffices to show that the first item $s'_k$ put in the bin $B_{m+1}$ has size at most $T/3$ since according to the algorithm, only items $s'_i$ with $i \geq k$ can be put in the bin $B_j$ with $j > m$, and for these items we have $s'_i \leq s'_k$.

Assume for contradiction that $s'_k > T/3$. Thus, $s'_i > T/3$ for all $i \leq k$. Consider the moment the algorithm **First-Fit-Decreasing** puts the item $s'_k$ in the bin $B_{m+1}$. We must have the following situation at this moment:

1. all bins $B_1$, …, $B_m$, $B_{m+1}$ are nonempty and all other bins are empty;
2. only the items $s'_1$, …, $s'_k$ have been put in bins;
3. each of the bins $B_1$, …, $B_{m+1}$ contains at most two items — this is because each of the items $s'_1$, …, $s'_k$ has size larger than $T/3$;
4. for any two bins $B_i$ and $B_j$, where $1 \leq i, j \leq m$, if bin $B_i$ contains one item while bin $B_j$ contains two items, then $i < j$ (proof: suppose $i > j$. Let the first item in $B_j$ be $s'_{j_1}$ and the second item in $B_j$ be $s'_{j_2}$, and let the item in $B_i$ be $s'_{i_1}$. According to the algorithm, we must have $s'_{j_1} \geq s'_{i_1}$ and $s'_{j_2} \geq s'_k$. Thus, $s'_{i_1} + s'_k \leq s'_{j_1} + s'_{j_2} \leq T$. Therefore, the algorithm would have put the item $s'_k$ in the bin $B_i$.)

Therefore, there must be an index $h$, such that each bin $B_j$ with $j \leq h$ contains a single item $s'_j$, while each bin $B_j$ with $h < j \leq m$ contains exactly two of the items $s'_{h+1}$, $s'_{h+2}$, …, $s'_{k-1}$. Thus, $k - h - 1$ is an even number and

$$k - h - 1 = 2(m - h)$$

Moreover, for any $j$ and $q$ such that $j \leq h$ and $j < q \leq k$, we must have $s'_j + s'_q > T$ — otherwise, the item $s'_q$ would have been put in the bin $B_j$.

Let $\mathcal{P} = (B'_1, B'_2, \ldots, B'_m)$ be an optimal packing of the items $s'_1$, ..., $s'_n$. If a bin $B'_j$ contains the item $s'_j$, for $j \leq h$, then the bin $B'_j$ cannot contain any other items $s'_q$, for $1 \leq q \leq k$ and $q \neq j$, since by the above analysis, $s'_j + s'_q > T$. Thus, the $h$ items $s'_1$, ..., $s'_h$ must be contained in $h$ different bins in $\mathcal{P}$ and these $h$ bins do not contain any of the items $s'_{h+1}$, ..., $s'_k$. Moreover, each of the bins in $\mathcal{P}$ can contain at most two items from $s'_{h+1}, \ldots, s'_k$ since each of these items has size larger than $T/3$. Therefore, the $k-h = 2(m-h)+1$ items $s'_{h+1}, \ldots, s'_k$ require at least another $\lceil (k-h)/2 \rceil = m - h + 1$ bins in $\mathcal{P}$. But this implies that the packing $\mathcal{P}$ uses at least $h + (m - h + 1) = m + 1 = Opt(\alpha) + 1$ bins, contradicting our assumption that $\mathcal{P}$ is an optimal packing of $\alpha$. This contradiction proves the lemma. $\square$

Now we are ready to derive the approximation ratio for the algorithm **First-Fit-Decreasing**.

**Theorem 7.1.5** *The approximation ratio of the algorithm* **First-Fit-Decreasing** *is* 1.5.

PROOF. Let $\alpha = \langle s_1, \ldots, s_n; T \rangle$ be any input instance of the BIN PACKING problem. Suppose that $Opt(\alpha) = m$ and that the algorithm **First-Fit-Decreasing** packs the items $s_1$, ..., $s_n$ using $r$ bins $B_1$, $B_2$, ..., $B_r$.

If $r = m$, then the approximation ratio $r/m = 1 < 1.5$ so the theorem holds. Thus, we assume $r > m$.

Let $s_k$ be an item in the bin $B_r$. By Lemma 7.1.4, we have $s_k \leq T/3$. Therefore, the content of each of the bins $B_1$, ..., $B_{r-1}$ is larger than $2T/3$ — otherwise the item $s_k$ would have been put in one of the bins $B_1$, ..., $B_{r-1}$. The same reason also shows that the content of the bin $B_{r-1}$ plus $s_k$ is larger than $T$. If we let $|B_j|$ be the content of the bin $B_j$, then we have

$$\sum_{i=1}^{n} s_i \geq \sum_{j=1}^{r-1} |B_j| + s_k = \sum_{j=1}^{r-2} |B_j| + (|B_{r-1}| + s_k) > (2T/3)(r-2) + T$$

Since the content of a bin in any packing cannot be larger than $T$, any packing of the $n$ items $s_1$, ..., $s_n$ uses at least $(\sum_{i=1}^{n} s_i)/T$ bins. Therefore,

$$Opt(\alpha) \geq (\sum_{i=1}^{n} s_i)/T > 2(r-2)/3 + 1 = (2r-1)/3$$

Observing that both $Opt(\alpha)$ and $r$ are integers, we easily verify that $Opt(\alpha) > (2r-1)/3$ implies $Opt(\alpha) \geq 2r/3$, which gives the approximation ratio

$$r/Opt(\alpha) \leq 3/2 = 1.5$$

This completes the proof of the theorem. □

Theorem 7.1.5 together with Theorem 7.1.2 shows that in terms of the approximation ratio, the algorithm **First-Fit-Decreasing** is the best possible polynomial time approximation algorithm for the BIN PACKING problem.

The algorithm **First-Fit-Decreasing** is also a good example to illustrate the difference between the regular approximation ratio and the asymptotic approximation ratio of an approximation algorithm. Theorem 7.1.5 claims that the regular approximation ratio of the algorithm **First-Fit-Decreasing** is 1.5 (and by Theorem 7.1.2, this bound is tight). On the other hand, it can be proved that for any instance $\alpha$ of the BIN PACKING problem, the number $r$ of bins used by the algorithm **First-Fit-Decreasing** for $\alpha$ satisfies the following condition (see [50] and its reference)

$$\frac{r}{Opt(\alpha)} \leq \frac{11}{9} + \frac{4}{Opt(\alpha)}$$

Therefore, the asymptotic approximation ratio of the algorithm **First-Fit-Decreasing** is bounded by $11/9 = 1.22\cdots$.

### 7.1.2 The $(\delta, \pi)$-Bin Packing problem

In order to develop an asymptotic polynomial time approximation scheme for the BIN PACKING problem, we first study a restricted version of the BIN PACKING problem, which will be called the $(\delta, \pi)$-BIN PACKING problem. The $(\delta, \pi)$-BIN PACKING problem is the general BIN PACKING problem with the following two restrictions. First, we assume that the input items have at most a constant number $\pi$ of different sizes. Second, we assume that the size of each input item is at least as large as a $\delta$ factor of the bin size. The following is a formal definition.

> $(\delta, \pi)$-BIN PACKING
>
> INPUT: $\langle t_1 : n_1, t_2 : n_2, \ldots, t_\pi : n_\pi; B\rangle$, where $\delta B \leq t_i \leq B$ for all $i$, interpreted as: for the $n = \sum_{i=1}^{\pi} n_i$ input items, $n_i$ of them are of size $t_i$, for $i = 1, \ldots, \pi$
>
> OUTPUT: a packing of the $n$ items into the minimum number of bins of size $B$

We first study the properties of the $(\delta, \pi)$-BIN PACKING problem. Let $\alpha = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$ be an input instance for the $(\delta, \pi)$-BIN PACKING problem. Suppose that an optimal packing packs the items in $\alpha$ into $m$ bins

$B_1, B_2, \ldots, B_m$. Consider the first bin $B_1$. Suppose that the bin $B_1$ contains $b_1$ items of size $t_1$, $b_2$ items of size $t_2$, ..., and $b_\pi$ items of size $t_\pi$. We then call

$$(b_1, b_2, \ldots, b_\pi)$$

the *configuration* of the bin $B_1$. Since each item has size at least $\delta B$ and the bin size is $B$, the bin $B_1$ contains at most $1/\delta$ items. In particular, we have $0 \leq b_i \leq 1/\delta$ for all $i$. Therefore, the total number of different bin configurations is bounded by $(1/\delta)^\pi$.

Now consider the set $\alpha'$ of items that is obtained from the set $\alpha$ with all items packed in the bin $B_1$ removed. The set $\alpha'$ can be written as

$$\alpha' = \langle t_1 : (n_1 - b_1), t_2 : (n_2 - b_2), \ldots, t_\pi : (n_\pi - b_\pi); B\rangle$$

Note that $\alpha'$ is also an input instance for the $(\delta, \pi)$-Bin Packing problem and the packing $(B_2, B_3, \ldots, B_m)$ is an optimal packing for $\alpha'$ ($\alpha'$ cannot be packed into fewer than $m - 1$ bins otherwise the set $\alpha$ can be packed into fewer than $m$ bins). Therefore, if we can pack the set $\alpha'$ into a minimum number of bins then an optimal packing for the set $\alpha$ can be obtained by packing the rest of the items into a single bin $B_1$.

Now the problem is that we do not know the configuration for the bin $B_1$. Therefore, we will try all possible configurations for a single bin, and recursively find an optimal packing for the rest of the items. As pointed out above, the number of bin configurations is bounded by $(1/\delta)^\pi$, which is a constant when both $\delta$ and $\pi$ are fixed. In the following, we present a dynamic programming algorithm that constructs an optimal packing for an input instance for the $(\delta, \pi)$-Bin Packing problem.

Fix an input instance $\alpha = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$ of the $(\delta, \pi)$-Bin Packing problem. Each subset of items in $\alpha$ can be written as a $\pi$-tuple $[h_1, \ldots, h_\pi]$ with $0 \leq h_i \leq n_i$ to specify that the subset contains $h_i$ items of size $t_i$ for all $i$. In particular, the input instance $\alpha$ itself can be written as $[n_1, \ldots, n_\pi]$.

Let $\#H[h_1, \ldots, h_\pi]$ denote the minimum number of bins needed to pack the subset $[h_1, \ldots, h_\pi]$ of the input instance $\alpha$ for the $(\delta, \pi)$-Bin Packing problem. Suppose that $\#H[h_1, \ldots, h_\pi] \geq 1$. According to the discussion above, we know that $\#H[h_1, \ldots, h_\pi]$ is equal to 1 plus $\#H[h_1 - b_1, \ldots, h_\pi - b_\pi]$ for some bin configuration $(b_1, b_2, \ldots, b_\pi)$. On the other hand, since $\#H[h_1, \ldots, h_\pi]$ corresponds to an optimal packing of the subset $[h_1, \ldots, h_\pi]$, $\#H[h_1, \ldots, h_\pi]$ is actually equal to 1 plus the minimum of $\#H[h_1 - b_1, \ldots, h_\pi - b_\pi]$ over all consistent bin configurations $(b_1, \ldots, b_\pi)$. This suggests an algorithm that uses the dynamic programming technique to compute the value of $\#H[h_1, \ldots, h_\pi]$. In particular, $\#H[n_1, \ldots, n_\pi]$ gives the optimal value for the input instance $\alpha$ for the $(\delta, \pi)$-BIN PACKING problem.

**Definition 7.1.1** Fix an input instance $\alpha = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$ for the $(\delta, \pi)$-BIN PACKING problem. Let $\alpha' = [h_1, \ldots, h_\pi]$ be a subset of the input items in $\alpha$, where $h_i \leq n_i$ for all $i$. A $\pi$-tuple $(b_1, \ldots, b_\pi)$ is an *addable bin configuration to* $\alpha'$ if

1. $h_i + b_i \leq n_i$ for all $i = 1, \ldots, \pi$; and
2. $\sum_{i=1}^{\pi} t_i b_i \leq B$.

Intuitively, an addable bin configuration specifies a bin configuration that can be obtained using the items in $\alpha$ that are not in the subset $\alpha'$.

Now we are ready for presenting the following dynamic programming algorithm. We use a $\pi$-dimensional array $H[1..n_1, \ldots, 1..n_\pi]$ (note that $\pi$ is a fixed constant) such that $H[i_1, \ldots, i_\pi]$ records an optimal packing for the subset $[i_1, \ldots, i_\pi]$ of $\alpha$. We use the notation $\#H[i_1, \ldots, i_\pi]$ to denote the number of bins used in the packing $H[i_1, \ldots, i_\pi]$. For a packing $H[i_1, \ldots, i_\pi]$ and a bin configuration $(b_1, \ldots, b_\pi)$, we will use

$$H[i_1, \ldots, i_\pi] \oplus (b_1, \ldots, b_\pi)$$

to represent the packing for the subset $[i_1 + b_1, \ldots, i_\pi + b_\pi]$ that is obtained from $H[i_1, \ldots, i_\pi]$ by adding a new bin with configuration $(b_1, \ldots, b_\pi)$. The dynamic programming algorithm $(\delta, \pi)$-**Precise**, which solves the $(\delta, \pi)$-BIN PACKING problem precisely, is presented in Figure 7.3.

The **if** statement in the inner loop body can obviously be done in time $O(n)$ (note that $\pi$ is a constant). Since $b_i \leq 1/\delta$ for all $i = 1, \ldots \pi$, there are at most $(1/\delta)^\pi$ addable bin configurations for each subset $[i_1, \ldots, i_\pi]$. Moreover, $n_i \leq n$ for all $i = 1, \ldots, \pi$. Therefore, the **if** statement in the inner loop body can be executed at most $n^\pi(1/\delta)^\pi$ times. We conclude that the running time of the algorithm $(\delta, \pi)$-**Precise** is bounded by $O(n^{\pi+1}(1/\delta)^\pi)$, which is a polynomial of $n$ when $\delta$ and $\pi$ are fixed constants.

The algorithm $(\delta, \pi)$-**Precise** is not very satisfying. In particular, even for a moderate constant $\pi$ of different sizes, the factor $n^{\pi+1}$ in the time complexity makes the algorithm not practically useful. On the other hand, we will see that our approximation algorithm for the general BIN PACKING problem is based on solving the $(\delta, \pi)$-BIN PACKING problem with a very large constant $\pi$ and a very small constant $\delta$. Therefore, we need, if possible at all, to improve the above time complexity. In particular, we would like to see if there is an algorithm that solves the $(\delta, \pi)$-BIN PACKING problem

**Algorithm.** $(\delta, \pi)$-**Precise**

Input: $\alpha = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B \rangle$, where $t_i \geq \delta B$ for all $i$
Output: a bin packing of $\alpha$ using minimum number of bins.

1. $\#H[i_1, \ldots, i_\pi] = +\infty$ for all $0 \leq i_j \leq n_j$, $1 \leq j \leq \pi$;
2. $H[0, \ldots, 0] = \phi$; $\#H[0, \ldots, 0] = 0$;
3. **for** $i_1 = 0$ to $n_1$ **do**

   $\ddots$

   **for** $i_\pi = 0$ to $n_\pi$ **do**
     **for** each bin configuration $(b_1, \ldots, b_\pi)$
         addable to the subset $[i_1, \ldots, i_\pi]$ **do**
       **if** $\#H[i_1 + b_1, \ldots, i_\pi + b_\pi] > 1 + \#H[i_1, \ldots, i_\pi]$
       **then**
         $H[i_1 + b_1, \ldots, i_\pi + b_\pi] = H[i_1, \ldots, i_\pi] \oplus (b_1, \ldots, b_\pi)$;
         $\#H[i_1 + b_1, \ldots, i_\pi + b_\pi] = \#H[i_1, \ldots, i_\pi] + 1$

Figure 7.3: The $(\delta, \pi)$-**Precise** algorithm

such that in the time complexity of the algorithm, the exponent of $n$ is independent of the values of $\pi$ and $\delta$.

Fix an input instance $\alpha = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B \rangle$ for the $(\delta, \pi)$-Bin Packing problem. We say that a $\pi$-tuple $(b_1, \ldots, b_\pi)$ is a *feasible bin configuration* if $b_i \leq n_i$ for all $i$ and $t_1 b_1 + \cdots t_\pi b_\pi \leq B$. Since $t_i \geq \delta B$ for all $i$, we have $b_i \leq 1/\delta$ for all $i$. Therefore, there are totally at most $(1/\delta)^\pi$ feasible bin configurations. Let all feasible bin configurations be

$$\begin{array}{rcl} T_1 & = & (b_{11}, b_{12}, \cdots, b_{1\pi}) \\ T_2 & = & (b_{21}, b_{22}, \cdots, b_{2\pi}) \\ & \vdots & \\ T_q & = & (b_{q1}, b_{q2}, \cdots, b_{q\pi}) \end{array} \tag{7.1}$$

where $q \leq (1/\delta)^\pi$. Note that the above list of feasible bin configurations can be constructed in time independent of the number $n = \sum_{i=1}^{\pi} n_i$ of items in the input instance $\alpha$. Now each bin packing $Y$ of the input instance $\alpha$ can be written as a $q$-tuple $(x_1, x_2, \ldots, x_q)$, where $x_j$ is the number of bins of bin configuration $T_j$ used in the packing $Y$. Moreover, there is essentially only one bin packing that corresponds to the $q$-tuple $(x_1, x_2, \ldots, x_q)$, if we ignore the ordering of the bins used. An optimal packing corresponds to a

$q$-tuple $(x_1, x_2, \ldots, x_q)$ with $x_1 + \cdots x_q$ minimized.

Conversely, in order to let a $q$-tuple $(x_1, x_2, \ldots, x_q)$ to describe a real pin packing for $\alpha$, we need to make sure that the $q$-tuple uses exactly the input items given in $\alpha$. For each feasible bin configuration $T_j$, there are $b_{jh}$ items of size $t_h$. Therefore, if $x_j$ bins are of bin configuration $T_j$, then for the bin configuration $T_j$, the $q$-tuple assumes $x_j b_{jh}$ items of size $t_h$. Now adding these over all bin configurations, we conclude that the total number of items of size $t_h$ assumed by the $q$-tuple $(x_1, x_2, \ldots, x_q)$ is

$$x_1 b_{1h} + x_2 b_{2h} + \cdots + x_q b_{qh}$$

This should match the number $n_h$ of items of size $t_h$ in the input instance $\alpha$. This formulates the conditions into the following linear programming problem.

$$\begin{array}{rcl} \min \quad x_1 + x_2 + \cdots + x_q & & \\ x_1 b_{11} + x_2 b_{21} + \cdots + x_q b_{q1} & = & n_1 \\ x_1 b_{12} + x_2 b_{22} + \cdots + x_q b_{q2} & = & n_2 \\ \vdots & & \\ x_1 b_{1\pi} + x_2 b_{2\pi} + \cdots + x_q b_{q\pi} & = & n_\pi \\ x_i \geq 0, \ \text{ for } \ i = 1, \ldots, q & & \end{array} \tag{7.2}$$

Since all $x_i$s must be integers, this is an instance of the INTEGER LINEAR PROGRAMMING problem. It is easy to see that if a $q$-tuple $(x_1, \ldots, x_q)$ corresponds to a valid bin packing of the input instance $\alpha$, then the vector $(x_1, \ldots, x_q)$ satisfies the constraints in the system (7.2). Conversely, any vector $(x_1, \ldots, x_q)$ satisfying the constraints in the system (7.2) describes a valid bin packing for the input instance $\alpha$. Moreover, it is easy to see that if a vector $(x_1, \ldots, x_q)$ satisfying the constraints in the system (7.2) is given, the corresponding bin packing can be constructed in linear time.

Therefore, to construct an optimal solution for the input instance $\alpha$ for the $(\delta, \pi)$-BIN PACKING problem, we only need to construct an optimal solution for the instance (7.2) for the INTEGER LINEAR PROGRAMMING problem. According to Theorem 5.1.4, the INTEGER LINEAR PROGRAMMING problem in general is NP-hard. But here the nice thing is that both the number $q$ of variables and the number $q + \pi$ of constraints in the system (7.2) are independent of $n = \sum_{i=1}^{\pi} n_i$. However, this does not immediately imply that the system can be solved in time independent of $n$ — the numbers $n_i$ appearing on the right side of the system may be as large as $n$. Thus, the

**Algorithm.** $(\delta, \pi)$**-Precise2**

Input: $\alpha = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$, where $t_i \geq \delta B$ for all $i$
Output: a bin packing of $\alpha$ using the minimum number of bins

1. construct the list (7.1) of all feasible configurations $T_1$, $T_2$, ..., $T_q$;
2. solve the system (7.2) using Lenstra's algorithm;
3. return the solution $(x_1, \ldots, x_q)$ of step 2.

Figure 7.4: The algorithm $(\delta, \pi)$**-Precise2**

vector $(x_1, \ldots, x_q)$ that is an optimal solution to the instance (7.2) can have elements as large as a polynomial of $n$ (see Theorem 5.2.9). In consequence, constructing an optimal solution to the instance (7.2) by enumerating all possible solutions may still take time dependent of $n$.

Anyway, the above system has at least suggested a polynomial time algorithm for solving the problem: we know that an optimal solution must satisfy $x_1 + \cdots + x_q \leq n$. Thus, $0 \leq x_i \leq n$ for all $i = 1, \ldots, q$ in an optimal solution. Therefore, we could enumerate all vectors $(x_1, \ldots, x_q)$ satisfying $0 \leq x_i \leq n$ and solve the system (7.2). Note that there are totally $(n+1)^q$ such vectors and $q$ is independent of $n$. However, since $q$ is of order $(1/\delta)^\pi$, this enumerating algorithm gives a polynomial time algorithm whose complexity is even worse than that of the algorithm $(\delta, \pi)$**-Precise**.

Fortunately, Lenstra [89] has developed an algorithm that solves the system (7.2) in time $h(q, \pi)$, where $h(q, \pi)$ is a function depending only on $q$ and $\pi$. Since the algorithm involves complicated analysis on the INTEGER LINEAR PROGRAMMING problem, we omit the description of the algorithm.

The above discussion results in a second polynomial time algorithm, $(\delta, \pi)$**-Precise2**, for the $(\delta, \pi)$-BIN PACKING problem, which is presented in Figure 7.4.

The algorithm $(\delta, \pi)$**-Precise2**, as discussed above, runs in time $h_1(q, \pi) = h_2(\pi, \delta)$, where $h_2$ is a function depending only on $\delta$ and $\pi$. This may seem a bit surprising since the algorithm packs $n = \sum_{i=1}^{\pi} n_i$ items in time independent of $n$. This is really a matter of coding. Note that the input $\alpha = \langle t_1 : n_1, \ldots, t_\pi : n_\pi; B\rangle$ of the algorithm $(\delta, \pi)$**-Precise2** actually consists of $2\pi + 1$ integers, and the solution $(x_1, \ldots, x_q)$ given by the algorithm consists of $q \leq (1/\delta)^\pi$ integers. To convert the vector $(x_1, \ldots, x_q)$ into an actual packing of the $n = \sum_{i=1}^{\pi} n_i$ input items, an extra step of time $O(n)$ should be added.

**Theorem 7.1.6** *The* $(\delta, \pi)$-Bin Packing *problem can be solved in time* $O(n) + h(\delta, \pi)$, *where* $h(\delta, \pi)$ *is a function independent of* $n$.

### 7.1.3 Asymptotic approximation schemes

In the last subsection, we have shown that the $(\delta, \pi)$-Bin Packing problem can be solved in time $O(n) + h(\delta, \pi)$, where $h(\delta, \pi)$ is a function independent of the number $n$ of items in the input instance (Theorem 7.1.6). In this subsection, we use the solution for the $(\delta, \pi)$-Bin Packing problem to develop an approximation algorithm for the general Bin Packing problem. We first roughly describe the basic idea of the approximation algorithm.

An input instance of the general Bin Packing problem may contain items of small size and items of many different sizes. To convert an input instance $\alpha = \langle t_1, \ldots, t_n; B\rangle$ of the general Bin Packing problem to an input instance of the $(\delta, \pi)$-Bin Packing problem, we perform two preprocessing steps:

1. ignore the items of small size, i.e., the items of size less than $\delta B$; and
2. sort the rest of the items by their sizes in decreasing order, then evenly partition the sorted list according to item size into $\pi$ groups $G_1$, ..., $G_\pi$. For each group $G_i$, replace every item by the one with the largest size $t'_i$ in $G_i$.

After the preprocessing steps, we obtain an instance $\alpha' = \langle t'_1 : m, \ldots, t'_\pi : m; B\rangle$ of the $(\delta, \pi)$-Bin Packing problem, where $m \le \lceil n/\pi \rceil$. Now we use the algorithm we have developed to construct a solution $Y'$, which is a packing, for the instance $\alpha'$. To obtain a solution $Y$ to the original input instance $\alpha$ of the Bin Packing problem, we first replace each item in $Y'$ by the corresponding item in $\alpha$, then add the items in $\alpha$ that have size smaller than $\delta B$ using the greedy method as described in the algorithm **First-Fit** in Figure 7.1.

An optimal solution to the instance $\alpha'$ is an over-estimation of the optimal solution to the instance $\alpha$ since each item in $\alpha$ is replaced by a larger item in $\alpha'$, the number of bins used by an optimal packing of $\alpha'$ is at least as large as the number of bins used by an optimal packing of $\alpha$. To see that an optimal solution to the instance $\alpha'$ is a good approximation to optimal solutions to the instance $\alpha$, consider the instance $\alpha'' = \langle t'_2 : m, \ldots, t'_\pi : m; B\rangle$, which is an instance to the $(\delta, \pi-1)$-Bin Packing problem. The instance $\alpha''$ can be obtained from $\alpha$ by replacing each item in the group $G_i$ by a smaller item of size $t'_{i+1}$ for $1 \le i \le \pi - 1$ and deleting the items in the last group

**Algorithm. ApxBinPack**

Input: $\alpha = \langle t_1, \ldots, t_n; B\rangle$ and $\epsilon > 0$
Output: a packing of the items $t_1$, $t_2$, ..., $t_n$

1. sort $t_1, \ldots, t_n$; without loss of generality, let $t_1 \geq t_2 \geq \cdots \geq t_n$;
2. let $h$ be the largest index such that $t_h \geq \epsilon B/2$; $\alpha_0 = \langle t_1, t_2, \ldots, t_h; B\rangle$;
3. let $\pi = \lceil 2/\epsilon^2 \rceil$, partition the items in $\alpha_0$ into $\pi$ groups $G_1$, ..., $G_\pi$, such that the group $G_i$ consists of the items $t_{(i-1)m+1}$, $t_{(i-1)m+2}$, ..., $t_{im}$; where $m = \lceil h/\pi \rceil$ (the last group $G_\pi$ contains $m' \leq m$ items);
4. construct an optimal solution $Y'$ to the instance
   $\alpha' = \langle t_1 : m, t_{m+1} : m, t_{2m+1} : m, \ldots, t_{(\pi-1)m+1} : m'; B\rangle$
   for the $(\epsilon/2, \pi)$-Bin Packing problem;
5. replace each item in $Y'$ of size $t_{jm+1}$ by a proper item in the group $G_{j+1}$ of $\alpha_0$, for $j = 0, \ldots, \pi - 1$, to construct a packing $Y_0$ for the instance $\alpha_0$;
6. add the items $t_{h+1}, \ldots, t_n$ in $\alpha$ to the packing $Y_0$ by greedy method (i.e., no new bin will be used until no used bin has enough space for the current item). This results in a packing for the instance $\alpha$.

Figure 7.5: The **ApxBinPack** algorithm

$G_\pi$. Thus, an optimal solution to the instance $\alpha''$ is an under-estimation of the optimal solution to $\alpha$. Since the instance $\alpha''$ can also be obtained from $\alpha'$ by deleting the $m$ largest items, an optimal packing of $\alpha'$ uses at most $m$ more bins than an optimal packing of $\alpha''$ (the $m$ bins are used to pack the $m$ largest items in $\alpha'$). Therefore, an optimal packing of $\alpha'$ uses at most $m$ more bins than an optimal packing of $\alpha$, with the items of size less than $\delta B$ ignored. When the value $\pi$ is sufficiently large, the value $m \leq \lceil n/\pi \rceil$ is small so that an optimal solution to $\alpha'$ will be a good approximation to the optimal solution to $\alpha$ with items of size less than $\delta B$ ignored.

Finally, after a good approximation of the optimal solution to the instance $\alpha$ minus the small items is obtained, we add the small items to this solution using greedy method. Since the small items have small size, the greedy method will not leave much room in each bin. Thus, the resulting packing will be a good approximation for the input instance $\alpha$ of the general Bin Packing problem.

We present the formal algorithm in Figure 7.5 and give formal analysis as follows.

According to Theorem 7.1.6 and note that $\pi = \lceil 2/\epsilon^2 \rceil$, the $(\epsilon/2, \pi)$-Bin Packing problem can be solved in time $O(n) + h(\epsilon/2, \pi) = O(n) + h_0(\epsilon)$, where $h_0(\epsilon)$ is a function depending only on $\epsilon$, we conclude that the algorithm **ApxBinPack** runs in time $O(n \log n) + h_0(\epsilon)$, if an $O(n \log n)$ time sorting algorithm is used for step 1.

We discuss the approximation ratio for the algorithm **ApxBinPack**. As before, we denote by $Opt(\alpha)$ the optimal value, i.e., the number of bins used by an optimal packing, of the input instance $\alpha$ of the Bin Packing problem.

**Lemma 7.1.7** *Let $\alpha_0$ be the input instance constructed by step 2 of the algorithm* **ApxBinPack**. *Then* $Opt(\alpha_0) \leq Opt(\alpha)$.

Proof. This is because the items in the instance $\alpha_0$ form a subset of the items in the instance $\alpha$ so $\alpha$ takes at least as many bins as $\alpha_0$. □

**Lemma 7.1.8** *Let $\alpha_0$ and $\alpha'$ be the input instances constructed by step 2 and step 4 of the algorithm* **ApxBinPack**, *respectively. Then*

$$Opt(\alpha') \leq Opt(\alpha_0)(1 + \epsilon) + 1$$

Proof. Note that the instance $\alpha'$ is obtained from the instance $\alpha_0$ by replacing each item in a group $G_i$ by the largest item $t_{(i-1)m+1}$ in the group. Therefore, an optimal packing for the instance $\alpha'$ uses at least as many bins as that used by an optimal packing for the instance $\alpha_0$. This gives

$$Opt(\alpha_0) \leq Opt(\alpha')$$

Now let

$$\alpha'' = \langle t_{m+1} : m, t_{2m+1} : m, \ldots, t_{(\pi-1)m+1} : m'; B \rangle$$

$\alpha''$ can be regarded as an instance obtained from $\alpha_0$ by (i) replacing each item in the group $G_i$ by a smaller item $t_{im+1}$ (recall that $t_{im+1}$ is the largest item in group $G_{i+1}$), for all $i = 1, \ldots, \pi - 2$; (ii) replacing $m'$ items in group $G_{\pi-1}$ by a smaller item $t_{(\pi-1)m+1}$; and (iii) eliminating rest of the items in group $G_{\pi-1}$ and all items in group $G_\pi$. Therefore, an optimal packing for $\alpha_0$ uses at least as many bins as an optimal packing for $\alpha''$. This gives

$$Opt(\alpha'') \leq Opt(\alpha_0)$$

Finally, the difference between the instances $\alpha'$ and $\alpha''$ are $m$ items of size $t_1$. Since every item can fit into a single bin, we must have

$$Opt(\alpha') \leq Opt(\alpha'') + m$$

Combining all these we obtain

$$Opt(\alpha_0) \le Opt(\alpha') \le Opt(\alpha_0) + m$$

This gives us

$$\begin{aligned} Opt(\alpha') &\le Opt(\alpha_0) + m = Opt(\alpha_0) + \lceil h/\pi \rceil \\ &\le Opt(\alpha_0) + h/\pi + 1 \le Opt(\alpha_0) + h\epsilon^2/2 + 1 \end{aligned} \tag{7.3}$$

Now since each item of $\alpha_0$ has size at least $\epsilon B/2$, each bin can hold at most $\lfloor 2/\epsilon \rfloor$ items. Thus, the number of bins $Opt(\alpha_0)$ used by an optimal packing for the instance $\alpha_0$ is at least as large as $h/\lfloor 2/\epsilon \rfloor \ge \epsilon h/2$:

$$\epsilon h/2 \le Opt(\alpha_0)$$

Use this in Equation (7.3), we get

$$Opt(\alpha') \le Opt(\alpha_0) + \epsilon \cdot Opt(\alpha_0) + 1 = Opt(\alpha_0)(1+\epsilon) + 1$$

The lemma is proved. □

From the packing $Y'$ to the instance $\alpha'$, we get a packing $Y_0$ for the instance $\alpha_0$ in step 5 of the algorithm **ApxBinPack**, as shown by the following lemma.

**Lemma 7.1.9** *The solution $Y_0$ constructed by step 5 of the algorithm* **ApxBinPack** *is a packing for the instance $\alpha_0$. Moreover, the number of bins used by $Y_0$ is at most $Opt(\alpha_0)(1+\epsilon)+1$.*

PROOF. First note that the instances $\alpha_0$ and $\alpha'$ have the same number of items. The solution $Y_0$ to $\alpha_0$ is obtained from the optimal solution $Y'$ to the instance $\alpha'$ by replacing each of the $m$ items of size $t_{(i-1)m+1}$ in $\alpha'$ by an item in group $G_i$ of $\alpha_0$. Since no item in group $G_i$ has size larger than $t_{(i-1)m+1}$, we actually replace items in the bins in $Y'$ by items of the same or smaller size. Therefore, no bin would get content more than $B$ in the packing $Y_0$. This shows that $Y_0$ is a packing for the instance $\alpha_0$.

Finally, since $Y_0$ uses exactly the same number of bins as $Y'$ and $Y'$ is an optimal packing for $\alpha'$. By Lemma 7.1.8, the number of bins used by $Y_0$, which is $Opt(\alpha')$, is at most $Opt(\alpha_0)(1+\epsilon)+1$. □

Now we are ready for deriving our main theorem.

**Theorem 7.1.10** *For any input instance $\alpha = \langle t_1, \ldots, t_n; B \rangle$ of the* Bin Packing *problem and for any $0 < \epsilon \leq 1$, the algorithm* **ApxBinPack** *constructs a bin packing of $\alpha$ that uses at most $Opt(\alpha)(1 + \epsilon) + 1$ bins.*

Proof. According to Lemma 7.1.9, the solution $Y_0$ constructed by step 5 of the algorithm **ApxBinPack** is a packing for the instance $\alpha_0$. Now step 6 of the algorithm simply adds the items in $\alpha - \alpha_0$ to $Y_0$ using greedy method. Therefore, the algorithm **ApxBinPack** constructs a packing for the input instance $\alpha$. Let $Y$ be the packing constructed by the algorithm **ApxBinPack** for $\alpha$ and let $r$ be the number of bins used by $Y$. There are two cases.

If in step 6 of the algorithm **ApxBinPack**, no new bin is introduced. Then $r$ equals the number of bins used by $Y_0$. According to Lemma 7.1.9 and Lemma 7.1.7, we get

$$r \leq Opt(\alpha_0)(1 + \epsilon) + 1 \leq Opt(\alpha)(1 + \epsilon) + 1$$

and the theorem is proved.

Thus, we assume that in step 6 of the algorithm **ApxBinPack**, new bins are introduced. According to our greedy strategy, no new bin is introduced unless no used bin has enough room for the current item. Since all items added by step 6 have size less than $\epsilon B/2$, we conclude that all of the $r$ bins in $Y$, except maybe one, have content larger than $B(1 - \epsilon/2)$. This gives us

$$t_1 + \cdots + t_n > B(1 - \epsilon/2)(r - 1)$$

Thus, an optimal packing of the instance $\alpha$ uses more than $(1 - \epsilon/2)(r - 1)$ bins. From

$$Opt(\alpha) > (1 - \epsilon/2)(r - 1)$$

we derive

$$r < Opt(\alpha)/(1 - \epsilon/2) + 1 \leq Opt(\alpha)(1 + \epsilon) + 1$$

The last inequality is because $\epsilon \leq 1$.

Therefore, in any case, the packing $Y$ constructed by the algorithm **ApxBinPack** for the input instance $\alpha$ of the Bin Packing problem uses at most $Opt(\alpha)(1 + \epsilon) + 1$ bins. The theorem is proved. □

Note that the condition that $\epsilon$ must be less than or equal to 1 loses no generality. In particular, if we are interested in an approximation algorithm for the Bin Packing problem with approximation ratio $1 + \epsilon$ with $\epsilon > 1$, we simply use the algorithm **First-Fit** (see Figure 7.1).

Let $Apx(\alpha)$ be the number of bins used by the algorithm **ApxBinPack** for the instance $\alpha$ for the BIN PACKING problem. Then from Theorem 7.1.10, we get

$$\frac{Apx(\alpha)}{Opt(\alpha)} \leq 1 + \epsilon + \frac{1}{Opt(\alpha)}$$

Therefore, the asymptotic approximation ratio of the algorithm **ApxBinPack** is bounded by $1 + \epsilon$, for any given constant $\epsilon > 0$.

The algorithm **ApxBinPack** is not yet quite an asymptotic fully polynomial time approximation scheme for the BIN PACKING problem — the running time of the algorithm is bounded by $O(n \log n) + h_0(\epsilon)$, where $h_0(\epsilon)$ does not seem to be bounded by a polynomial of $1/\epsilon$. In fact, the value $h_0(\epsilon)$ can be huge when $\epsilon$ is small. Therefore, a possible further improvement to the algorithm **ApxBinPack** is an asymptotic fully polynomial time approximation scheme for the BIN PACKING problem. Such an improvement has been achieved by Karmakar and Karp [77], who use a similar approach that reduces the BIN PACKING problem to the LINEAR PROGRAMMING problem. The algorithm is based on some deep observations on the LINEAR PROGRAMMING problem. We omit the detailed description here. Instead, we state the result directly.

**Theorem 7.1.11** (Karmakar and Karp) *There is an approximation algorithm $A$ for the* BIN PACKING *problem such that for any $\epsilon > 0$, the algorithm $A$ produces in time polynomial in $n$ and $1/\epsilon$ a packing in which the number of bins used is bounded by*

$$Opt(x)(1 + \epsilon) + 1/\epsilon^2 + 3$$

**Corollary 7.1.12** *The* BIN PACKING *problem has an asymptotic fully polynomial time approximation scheme.*

PROOF. For any $\epsilon > 0$, let $N_\epsilon = (8 + 6\epsilon^2)/\epsilon^3$. For each input instance $\alpha$ of the BIN PACKING problem, let the Karmakar-Karp algorithm construct a packing that uses at most

$$r \leq Opt(\alpha)(1 + \epsilon/2) + (2/\epsilon)^2 + 3$$

bins. Now for input instances $\alpha$ with

$$Opt(\alpha) \geq N_\epsilon = (8 + 6\epsilon^2)/\epsilon^3$$

we have

$$\frac{r}{Opt(\alpha)} \leq 1 + \frac{\epsilon}{2} + \frac{(2/\epsilon)^2 + 3}{Opt(\alpha)} \leq 1 + \epsilon$$

Moreover, the algorithm runs in time polynomial in $n$ and $2/\epsilon$, which is also in polynomial in $n$ and $1/\epsilon$. □

### 7.1.4 Further work and extensions

—— Don: would you write this, please. ——

## 7.2 Graph edge coloring problem

The BIN PACKING problem is a typical optimization problem that has an asymptotic fully polynomial time approximation scheme. In fact, the concept of asymptotic fully polynomial time approximation scheme was introduced based on the study of approximation algorithms for the BIN PACKING problem.

There is another class of optimization problems that have asymptotic fully polynomial time approximation schemes. The asymptotic fully polynomial time approximation schemes for this class of optimization problems are derived from polynomial time approximation algorithms for the problems that deliver solutions such that the difference of the value of these solutions and the optimal solution value is very small.

**Definition 7.2.1** Let $Q = \langle I_Q, S_Q, f_Q, opt_Q\rangle$ be an optimization problem and let $d(n)$ be a function. We say that $Q$ can be approximated with an *additive difference* $d(n)$ in polynomial time if there is a polynomial time approximation algorithm $A$ for $Q$ such that for any input instance $x$ of $Q$, the algorithm $A$ produces a solution $y$ to $x$ such that

$$|Opt(x) - f_Q(x, y)| \leq d(|x|)$$

We are most interested in the case that the additive difference $d(n)$ is bounded by a constant.

We start the discussion with the famous planar graph coloring problem.

PLANAR GRAPH COLORING

INPUT: a planar graph $G$

OUTPUT: a coloring of the vertices of $G$ such that no two adjacent vertices are colored with the same color and the number of colors used is minimized.

**Theorem 7.2.1** *The* PLANAR GRAPH COLORING *problem can be approximated in polynomial time with an additive difference 1.*

PROOF. Consider the following approximation algorithm for the PLANAR GRAPH COLORING problem: let $G$ be an input planar graph. First note that $G$ can be colored with a single color if and only if $G$ has no edges. If $G$ contains edges, then we check whether $G$ is 2-colorable — this is equivalent to checking whether $G$ is a bipartite graph and can be done in linear time using the depth first search process. If the graph $G$ is 2-colorable, then we color $G$ with two colors and obtain an optimal solution. Otherwise, the graph $G$ needs at least three colors. According to the famous Four-Color Theorem, every planar graph can be colored by four colors. Moreover, there is an algorithm that colors any given planar graph with four colors in time $O(n^2)$ [3, 4, 108]. Therefore, for any planar graph that needs at least three colors, we can color it with four colors in polynomial time. □

Therefore, the PLANAR GRAPH COLORING problem can be approximated very well in terms of the additive difference. On the other hand, however, the PLANAR GRAPH COLORING problem cannot be approximated in polynomial time with an approximation ratio arbitrarily close to 1, as shown in the following theorem.

**Theorem 7.2.2** *No polynomial time approximation algorithm for the* PLANAR GRAPH COLORING *problem has an optimization ratio less than* $4/3$ *unless* $P = NP$.

PROOF. The decision problem PLANAR GRAPH 3-COLORABILITY: "given a planar graph $G$, can $G$ be colored with at most 3 colors?" is NP-complete [51]. An approximation algorithm for the PLANAR GRAPH COLORING problem with approximation ratio less than $4/3$ would always color a 3-colorable planar graph with 3 colors. Thus, if such an approximation algorithm runs in polynomial time, then the NP-complete problem PLANAR GRAPH 3-COLORABILITY would be solvable in polynomial time, which would imply P = NP. □

The PLANAR GRAPH COLORING problem is not an example that admits an asymptotic fully polynomial time approximation scheme because of a good approximation algorithm with a small additive difference. In fact, by the Four-Color Theorem, the optimal solution value for any instance of the PLANAR GRAPH COLORING problem is bounded by 4. Therefore, it makes no sense to talk about the asymptotic approximation ratio of an algorithm in terms of the optimal solution values. However, the PLANAR GRAPH COLORING problem does give a good example that is an NP-hard optimization problem but can be approximated in polynomial time with a very small additive difference.

Now we consider another example for which optimal solution values are not bounded while the problem still has very good approximation algorithm in terms of the additive difference. Let $G$ be a graph. $G$ is a *simple graph* if it contains neither self-loops nor multiple edges. We say that two edges of $G$ are *adjacent* if they share a common endpoint.

> GRAPH EDGE COLORING
>
> INPUT: a simple graph $G$
>
> OUTPUT: a coloring of the edges of $G$ such that no two adjacent edges are colored with the same color and the number of colors used is minimized.

As for the PLANAR GRAPH COLORING problem, the GRAPH EDGE COLORING problem has no polynomial time approximation algorithm with approximation ratio arbitrarily close to 1, as shown below.

**Theorem 7.2.3** *No polynomial time approximation algorithm for the* GRAPH EDGE COLORING *problem has an approximation ratio less than* 4/3 *unless P = NP.*

PROOF. The decision problem GRAPH EDGE 3-COLORABILITY: "given a simple graph $G$, can the edges of $G$ be colored with at most 3 colors?" is NP-complete [66]. A polynomial time approximation algorithm for the GRAPH EDGE COLORING problem with approximation ratio less that 4/3 would imply that the GRAPH EDGE 3-COLORABILITY problem can be solved in polynomial time, which would imply P = NP. □

Given a graph $G$, let $v$ be a vertex of $G$. Define $deg(v)$ to be the degree of the vertex and define $deg(G)$ to be the maximum $deg(v)$ over all vertices $v$ of $G$.

The following lemma follows directly from the definition.

**Lemma 7.2.4** *Every edge coloring of a graph $G$ uses at least $deg(G)$ colors.*

**Algorithm. Edge-Coloring**

Input: a simple graph $G$
Output: an edge coloring of $G$

1. let $G_0 = G$ with all edges deleted, and suppose that the edges of $G$ are $e_1, \ldots, e_m$;
2. **for** $i = 1$ **to** $m$ **do**
   $G_i = G_{i-1} \cup \{e_i\}$;
   color the edges of $G_i$ using at most $d + 1$ colors;

Figure 7.6: Edge coloring a graph $G$ with $deg(G) + 1$ colors

Since $deg(G)$ can be arbitrarily large, the optimal solution value for an instance of the Graph Edge Coloring problem is not bounded by any constant. This is the difference to the case of the Planar Graph Coloring problem.

**Theorem 7.2.5** *There is a polynomial time algorithm that colors the edges of any given graph $G$ with at most $deg(G) + 1$ colors.*

Proof. Let $G$ be the input graph. To simplify expressions, let $d = deg(G)$. We present an algorithm that colors the edges of $G$ using at most $d+1$ colors.

The algorithm is given in Figure 7.6.

We need to explain how the graph $G_i$, for each $i$, can be colored with at most $d + 1$ colors. The graph $G_0$, which has no edges, can certainly be colored with $d + 1$ colors. Inductively, suppose that we have colored the edges of $G_{i-1}$ using at most $d + 1$ colors. Now $G_i = G_{i-1} \cup \{e_i\}$, where we suppose $e_i = [v_1, w]$. Thus, we have all edges of $G_i$ except $e_i$ colored properly using at most $d + 1$ colors.

We say that a vertex $u$ in $G_i$ *misses* a color $c$ if no edge incident on $u$ is colored with $c$. Since we have $d + 1$ colors and each vertex of $G_i$ has degree at most $d$, every vertex of $G_i$ misses at least one color.

If both vertices $v_1$ and $w$ miss a common color $c$, then we simply color the edge $e_i = [v_1, w]$ with the color $c$ and we obtain a valid edge coloring for the graph $G_i$ by $d + 1$ colors.

So we suppose that there is no color that is missed by both $v_1$ and $w$. Let $c_1$ be the color missed by $v_1$ and $c_0$ be the color missed by $w, c_1 \neq c_0$.

Since $c_1$ is not missed by $w$, there is an edge $[v_2, w]$ colored with $c_1$. Now if $v_2$ and $w$ have a common missed color, we stop. If $v_2$ and $w$ have no

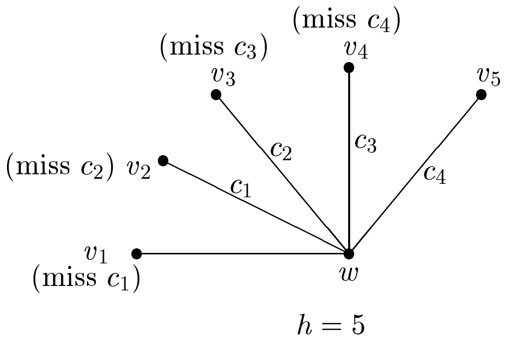



Figure 7.7: A fan structure

common missed color, then let $c_2$ be a color missed by $v_2$ — $c_2$ is not missed by $w$. Now let $[v_3, w]$ be the edge colored with $c_2$.

Inductively, suppose that we have constructed a "fan" that consists of $h$ neighbors $v_1, \ldots, v_h$ of $w$ and $h-1$ different colors $c_1, \ldots, c_{h-1}$, such that (see Figure 7.7)

- for all $j = 1, \ldots, h-1$, the vertex $v_j$ misses color $c_j$ and the edge $[v_{j+1}, w]$ is colored with the color $c_j$;
- none of the vertices $v_1, \ldots, v_{h-1}$ have a common missed color with $w$;
- for all $j = 1, \ldots, h-1$, the vertex $v_j$ does not miss any of the colors $c_1, \ldots, c_{j-1}$.

Now we consider the vertex $v_h$. There are three possible cases.

**Case 1**. the vertex $v_h$ does not miss any of the colors $c_1, \ldots, c_{h-1}$ and $v_h$ has no common missed color with $w$.

Then let $c_h$ be a color missed by $v_h$. Since $c_h$ is not missed by $w$, there is an edge $[v_{h+1}, w]$ colored with $c_h$. Thus, we have expanded the fan structure by one more edge.

Since the degree of the vertex $w$ is bounded by $d$, **Case 1** must fail at some stage and one of the following two cases should happen.

**Case 2**. the vertex $v_h$ has a common missed color $c_0$ with $w$.

Then we change the coloring of the fan by coloring the edge $[v_h, w]$ with $c_0$, and coloring the edge $[v_i, w]$ with $c_i$, for $i = 1, \ldots, h-1$ (see Figure 7.8). It is easy to verify that this gives a valid edge coloring for the graph $G_i = G_{i-1} \cup \{e_i\}$.

**Case 3**. the vertex $v_h$ misses a color $c_s$, $1 \leq s \leq h-1$.

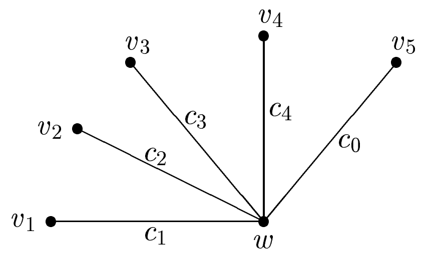



Figure 7.8: In case $v_h$ and $w$ miss a common color $c_0$

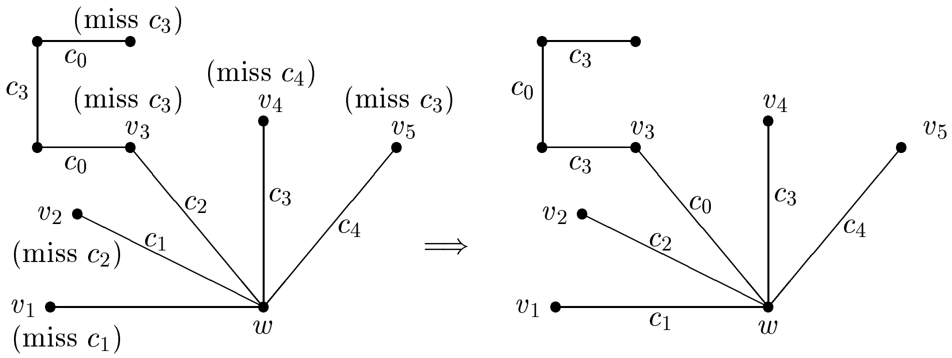
Let $c_0$ be a color missed by $w$. We start from the vertex $v_s$. Since $v_s$ has no common missed color with $w$, there is an edge $[v_s, u_1]$ colored with $c_0$. Now if $u_1$ does not miss $c_s$, there is an edge $[u_1, u_2]$ colored with $c_s$, now we look at vertex $u_2$ and see if there is an edge colored with $c_0$, and so on. By this, we obtained a path $P_s$ whose edges are alternatively colored by $c_0$ and $c_s$. The path has the following properties: (1) the path $P_s$ must be a simple path since each vertex of the graph $G_{i-1}$ has at most two edges colored with $c_0$ and $c_s$. Thus, the path $P_s$ must be a finite path; (2) the path $P_s$ cannot be a cycle since the vertex $v_s$ misses the color $c_s$; and (3) the vertex $w$ is not an interior vertex of $P_s$ since $w$ misses the color $c_0$.

Let $P_s = \{v_s, u_1, \ldots, u_t\}$, where $v_s$ misses the color $c_s$, $u_t$ misses one of colors $c_s$ and $c_0$, and $u_j$, $j = 1, \ldots, t-1$, misses neither $c_s$ nor $c_0$.

If $u_t \neq w$, then interchange the colors $c_0$ and $c_s$ on the path $P_s$ to make the vertex $v_s$ miss $c_0$. Then color $[v_s, w]$ with $c_0$ and color $[v_j, w]$ with $c_j$, for $j = 1, \ldots, s-1$ (see Figure 7.9). It is easy to verify that this gives a valid edge coloring for the graph $G_i = G_{i-1} \cup \{e_i\}$.

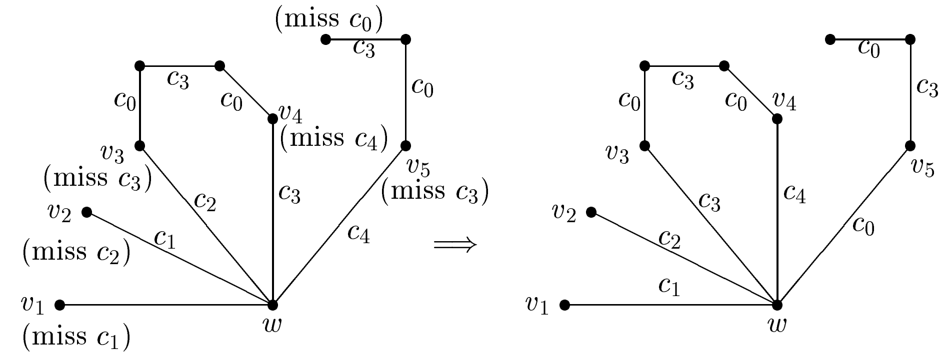
If $u_t = w$, we must have $u_{t-1} = v_{s+1}$. Then we grow a $c_0$-$c_s$ alternating path $P_h$ starting form the vertex $v_h$, which also misses the color $c_s$. Again $P_h$ is a finite simple path. Moreover, the path $P_h$ cannot end at the vertex $w$ since no vertex in $G_{i-1}$ is incident on more than two edges colored with $c_0$ and $c_s$ and the vertex $w$ misses the color $c_0$. Therefore, similar to what we did for vertex $v_s$, we interchange the colors $c_0$ and $c_s$ on the path $P_h$ to make $v_h$ miss $c_0$. Then color $[v_h, w]$ with $c_0$ and color $[v_j, w]$ with $c_j$ for $j = 1, \ldots, h-1$ (see Figure 7.10). It is easy to verify that this gives a valid edge coloring for the graph $G_i = G_{i-1} \cup \{e_i\}$.

Therefore, starting with an edge coloring of the graph $G_{i-1}$ using at most $d+1$ colors, we can always derive a valid edge coloring for the graph




$h = 5$ and $s = 3$

Figure 7.9: Extending a $c_0$-$c_s$ alternating path $P_s$ from $v_s$ not ending at $w$




$h = 5$ and $s = 3$

Figure 7.10: Extending a $c_0$-$c_s$ alternating path $P_h$ from $v_h$ not ending at $w$

$G_i = G_{i-1} \cup \{e_i\}$ using at most $d+1$ colors.

To show that the algorithm **Edge-Coloring** runs in polynomial time, we only need to show that the edge coloring of the graph $G_i$ can be constructed from the edge coloring of the graph $G_{i-1}$ in polynomial time. Suppose that the $d+1$ colors we use to color the graph $G$ are $c_1, \ldots, c_{d+1}$. Note that $d+1 \leq n$, where $n$ is the number of vertices in $G$. For each vertex $v$, we maintain an array of $d+1$ elements such that the $i$th element is a vertex $w$ if the edge $[v,w]$ is colored with the color $c_i$. Moreover, in the linked list structure of the graph $G$, we also record the color for each edge. Therefore, the fan structure for a vertex $w$ can be constructed in time $O(deg(w)) = O(n)$. It is also easy to see that the alternating paths in **Case 3** can also be constructed in time $O(n)$. Hence, the edge coloring of the graph $G_i$ based on an edge coloring of the graph $G_{i-1}$ can be constructed in time $O(n)$. Since $G_m = G$, where $m$ is the number of edges in the graph $G$, we conclude that an edge coloring of the graph $G$ with $d+1$ colors can be constructed in time $O(nm)$. □

**Corollary 7.2.6** *The* GRAPH EDGE COLORING *problem can be approximated within an additive difference 1 in polynomial-time.*

PROOF. The proof follows directly from Lemmas 7.2.4 and 7.2.5. □

**Corollary 7.2.7** *The* GRAPH EDGE COLORING *problem has an asymptotic fully polynomial time approximation scheme.*

PROOF. For any graph $G$, let $A(G)$ be the number of colors used by the algorithm **Edge-Coloring** to color the edges of $G$, and let $Opt(G)$ be the minimum number of colors that can be used to color the edges of $G$. By Corollary 7.2.6, we have

$$A(G) \leq Opt(G) + 1$$

Thus, the approximation ratio of the algorithm **Edge-Coloring** is bounded by

$$A(G)/Opt(G) \leq 1 + 1/Opt(G)$$

Therefore, for any given $\epsilon > 0$, if we let $N_\epsilon = 1/\epsilon$, then when $Opt(G) \geq N_\epsilon$, the approximation ratio of the algorithm **Edge-Coloring** is bounded by $1+\epsilon$. Moreover, the running time of algorithm **Edge-Coloring** is bounded by a polynomial of the input length $|G|$ (and independent of $\epsilon$). □

## 7.3 On approximation for additive difference

Corollary 7.2.6 shows that the GRAPH EDGE COLORING problem, which is NP-hard, can be approximated in polynomial time to an additive difference 1. Another optimization problem of this kind is the MINIMUM-DEGREE SPANNING TREE problem, which asks the construction of a spanning tree for a given graph such that the maximum vertex degree in the tree is minimized. It has been shown [44] that the MINIMUM-DEGREE SPANNING TREE problem can be approximated in polynomial time to an additive difference 1. In consequence, the MINIMUM DEGREE SPANNING TREE problem has an asymptotic fully polynomial time approximation scheme.

Not many optimization problems are known that can be approximated in polynomial time to a small additive difference. It is also not clear how we can recognize when an optimization problem can be approximated in polynomial time to a small additive difference. Nevertheless, there are a few known techniques that can be used to show the impossibility of polynomial time approximation within a small additive difference. In this section, we will introduce these techniques.

We say that the optimal solution values of an optimization problem $Q$ is *unbounded* if for any constant $N$, there is an instance $x$ of $Q$ such that $Opt(x) \geq N$.

**Theorem 7.3.1** *Let $Q$ be an optimization problem with unbounded optimal solution values. If $Q$ has no asymptotic fully polynomial time approximation scheme, then $Q$ cannot be approximated in polynomial time to a constant additive difference.*

PROOF. Without loss of generality, suppose that $Q$ is a minimization problem. Assume the contrary that $A$ is a polynomial time algorithm that approximates $Q$ to a constant additive difference $c$. Then for any $\epsilon > 0$, let $N_\epsilon = c/\epsilon$. For any instance $x$ of $Q$ such that $Opt(x) \geq N_\epsilon$, let $f_Q(x, y)$ be the value of the solution $y$ to $x$ constructed by the algorithm $A$. Then we have

$$f_Q(x, y) \leq Opt(x) + c$$

which gives the approximation ratio

$$f_Q(x, y)/Opt(x) \leq 1 + c/Opt(x) \leq 1 + \epsilon$$

Moreover, the running time of the algorithm $A$ is bounded by a polynomial of $|x|$ and is independent of $\epsilon$. That is, the optimization problem $Q$ has an asymptotic fully polynomial time approximation scheme. □

Therefore, having an asymptotic fully polynomial time approximation scheme is a necessary condition for an optimization problem to have a polynomial time approximation algorithm with a constant additive difference.

In the following, we describe another technique that shows that for a class of optimization problems, it is even impossible to have a polynomial time approximation algorithm with a relatively large additive difference.

Recall the KNAPSACK problem.

KNAPSACK = $\langle I_Q, S_Q, f_Q, opt_Q \rangle$

$I_Q$: the set of tuples $\alpha = (s_1, \ldots, s_n; v_1, \ldots, v_n; B)$, where all $s_i$, $v_j$, $B$ are integers

$S_Q$: $S_Q(\alpha)$ is the collection of subsets $T$ of $\{1, \ldots, n\}$ such that $\sum_{i \in T} s_i \leq B$

$f_Q$: $f_Q(\alpha, T) = \sum_{i \in T} v_i$

$opt_Q$: max

**Theorem 7.3.2** *There is no polynomial time approximation algorithm for the* KNAPSACK *problem that guarantees an additive difference* $2^n$ *unless P = NP.*

PROOF. Suppose that $A$ is a polynomial time approximation algorithm for the KNAPSACK problem KNAPSACK = $\langle I_Q, S_Q, f_Q, opt_Q \rangle$ such that for any input instance $\alpha \in I_Q$, the algorithm $A$ produces a solution $T_\alpha \in S_Q(\alpha)$ such that $|Opt(\alpha) - f_Q(\alpha, T_\alpha)| \leq 2^n$. We show how we can use this algorithm to solve the KNAPSACK problem precisely in polynomial time.

Given an input instance $\alpha = (s_1, \ldots, s_n; v_1, \ldots, v_n; B)$ for the KNAPSACK problem, we construct another input instance for the problem: $\beta = (s_1, \ldots, s_n; v_1 2^{n+1}, \ldots, v_n 2^{n+1}; B)$ (i.e. we scale the values $v_i$ in $\alpha$ to be a multiple of $2^{n+1}$ so that the difference of any two different values is larger than $2^n$). Then we apply the algorithm $A$ to $\beta$ to get a solution $T_\beta$. According to our assumption, $|Opt(\beta) - f_Q(\beta, T_\beta)| \leq 2^n$. Since both $Opt(\beta)$ and $f_Q(\beta, T_\beta)$ are multiples of $2^{n+1}$, we conclude that $Opt(\beta) = f_Q(\beta, T_\beta)$, that is, the solution $T_\beta$ is an optimal solution to the instance $\beta$. Moreover, it is easy to see that $T_\beta$ is also a solution to the instance $\alpha$ and $Opt(\beta) = 2^{n+1} Opt(\alpha)$ and $f_Q(\beta, T_\beta) = 2^{n+1} f_Q(\alpha, T_\beta)$. Therefore, $T_\beta$ is also an optimal solution for the instance $\alpha$.

By our assumption, the algorithm $A$ runs in polynomial time. Moreover, since the number $2^{n+1}$ has $O(n) = O(|\alpha|)$ bits, the instance $\beta$ can be constructed from the instance $\alpha$ in polynomial time. Therefore, the above

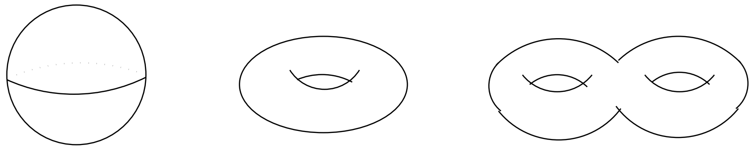



Figure 7.11: Surfaces of genus 0, 1, and 2

process constructs an optimal solution for the given instance $\alpha$ in polynomial time. Consequently, the KNAPSACK problem can be solved in polynomial time. Since the KNAPSACK problem is NP-hard, it follows that P = NP. □

The proof for Theorem 7.3.2 can be easily extended to other optimization problems in which instances may contain very large numbers and optimal solution values are calculated based on these numbers. Examples of this kind of problems include the $c$-MAKESPAN problem and the TRAVELING SALESMAN problem. Note that some of these problems, such as the KNAPSACK problem and the $c$-MAKESPAN problem, have very good approximation algorithms (fully polynomial time approximation schemes) in terms of the approximation ratio.

The main reason that Theorem 7.3.2 holds for many optimization problems is that we can scale the numbers in the input instances so that a small additive difference would make no difference for the scaled instances. However, what about the optimization problems in which no large numbers are allowed in its input instances? In particular, is there a similar theorem for optimization problems whose instances contain no number at all? We demonstrate a technique for this via the study of an optimization problem related to graph embeddings.

A *surface* $S_k$ *of genus* $k$ is obtained from the sphere by adding $k$ "handles". Therefore, $S_0$ is the sphere, $S_1$ is the torus, $S_2$ is the double torus, and so on (see Figure 7.11).

An *embedding* of a graph $G$ on a surface $S_k$ is a one-to-one continuous mapping from vertices and edges of $G$ to points and curves of the surface $S_k$, respectively. By the definition, no "edge crossing" is allowed in a graph embedding. Figure 7.12 illustrates an embedding of the complete graph $K_5$ on the surface $S_1$.

The *genus of a graph* $G$, denoted $\gamma_{\min}(G)$, is the minimum integer $k \geq 0$ such that the graph $G$ can be embedded in the surface $S_k$. In particular, the class of graphs of genus 0 is exactly the class of planar graphs, and the class

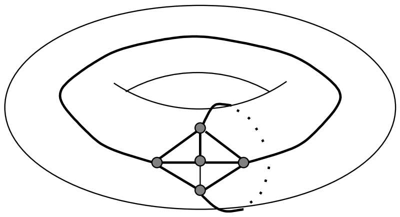


Figure 7.12: An embedding of $K_5$ on $S_1$

of graphs of genus 1 is the class of non-planar graphs that can be embedded on the torus. For example, by the well-known Kuratowski Theorem [58], the complete graph $K_5$ is non-planar. Thus, the embedding of $K_5$ given in Figure 7.12 shows that the graph $K_5$ has genus 1.

Now we are ready to formulate the following problem.

GRAPH GENUS

INPUT: a graph $G$

OUTPUT: an embedding $\rho(G)$ of $G$ on the surface $S_k$,
where $k = \gamma_{\min}(G)$

The GRAPH GENUS problem has applications in many areas such as VLSI layouts and graph isomorphism testing. Unfortunately, it has been recently shown that the GRAPH GENUS problem is NP-hard [113]. In the rest of this section, we discuss polynomial time approximability of the GRAPH GENUS problem in terms of the additive difference.

Graph embeddings can be studied using *graph rotation systems*. A *rotation* at a vertex $v$ is a cyclic permutation of the edge-ends incident on $v$. A list of rotations, one for each vertex of the graph, is called a *rotation system*.

An embedding of a graph $G$ on a surface induces a rotation system, as follows: the rotation at vertex $v$ is the cyclic permutation corresponding to the order in which the edge-ends are traversed in an orientation-preserving tour around $v$. Conversely, it can be shown [58] that every rotation system induces a unique embedding of $G$ on a surface. Therefore, a rotation system of a graph is a combinatorial representation of an embedding of the graph. In the following, we will interchangeably use the phrases "an embedding of a graph" and "a rotation system of a graph".

Suppose that $\rho(G)$ is a rotation system of the graph $G = (V, E)$ on the surface $S_k$. Then $k$ is called the *genus of the rotation system* $\rho(G)$, denoted

$\gamma(\rho(G))$. By the Euler polyhedral equation [58]

$$|V| - |E| + |F| = 2 - 2\gamma(\rho(G))$$

where $|F|$ is the number of faces in the embedding $\rho(G)$. There is a linear time algorithm that, given a rotation system $\rho(G)$ for a graph $G$, traces the boundary walks of all faces in the rotation system [58]. Therefore, given a rotation system $\rho(G)$, the genus $\gamma(\rho(G))$ of $\rho(G)$ can be computed in linear time.

Let $G$ and $G'$ be two graphs. The *bar-amalgamation* of $G$ and $G'$, denoted $G * G'$, is the result of running a new edge (called the "bar") from a vertex of $G$ to a vertex of $G'$. The definition of bar-amalgamation on two graphs can be extended to more than two graphs. Inductively, a *bar-amalgamation* of $r$ graphs $G_1, \ldots, G_r$, written $G_1 * G_2 * \cdots * G_r$, is the bar-amalgamation of the graph $G_1$ and the graph $G_2 * \cdots * G_r$.

Let $G$ be a graph and let $H$ be a subgraph of $G$. Let $\rho(G)$ be a rotation system of $G$. A rotation system $\rho'(H)$ of $H$ can be obtained from $\rho(G)$ by deleting all edges that are not in $H$. The rotation system $\rho'(H)$ of $H$ will be called an *induced rotation system* of $H$ from the rotation system $\rho(G)$.

The proofs for the following theorem and corollary are omitted.

**Theorem 7.3.3** *Let $G_1, \ldots, G_r$ be graphs and let $\rho(G_1 * \cdots * G_r)$ be a rotation system of a bar-amalgamation $G_1 * \cdots * G_r$ of $G_1, \ldots, G_r$. Then*

$$\gamma(\rho(G_1 * \cdots * G_r)) = \sum_{i=1}^{r} \gamma(\rho_i(G_i))$$

*where $\rho_i(G_i)$ is the induced rotation system of $G_i$ from $\rho(G_1 * \cdots * G_r)$, $1 \le i \le r$.*

**Corollary 7.3.4** *Let $G_1, \ldots, G_r$ be graphs and let $G'$ be an arbitrary bar-amalgamation of $G_1, \ldots, G_r$. Then*

$$\gamma_{\min}(G') = \sum_{i=1}^{r} \gamma_{\min}(G_i)$$

Now we are ready for the main theorem.

**Theorem 7.3.5** *For any fixed constant $\epsilon$, $0 \le \epsilon < 1$, the* GRAPH GENUS *problem cannot be approximated in polynomial time to an additive difference $n^\epsilon$ unless $P = NP$.*

PROOF. Suppose that $A$ is a polynomial time approximation algorithm that, given a graph $G$ of $n$ vertices, constructs an embedding of $G$ of genus at most $\gamma_{\min}(G) + n^{\epsilon}$.

Let $k$ be a fixed integer such that $\epsilon < \frac{k}{k+1}$. Then for sufficiently large $n$, we have $n^{\epsilon} < n^{\frac{k}{k+1}}$. Thus $n^{\epsilon(k+1)} \leq n^k - 1$.

Let $n^k G$ be a graph that is an arbitrary bar amalgamation of $n^k$ copies of $G$. Then the number of vertices of $n^k G$ is $N = n^{k+1}$. The graph $n^k G$ can be obviously constructed from $G$ in polynomial time. Moreover, by Corollary 7.3.4

$$\gamma_{\min}(n^k G) = n^k \cdot \gamma_{\min}(G)$$

Now running the algorithm $A$ on the graph $n^k G$ gives us an embedding $\rho(n^k G)$ of $n^k G$, which has genus at most $\gamma_{\min}(n^k G) + N^{\epsilon}$. Therefore,

$$\begin{aligned} \gamma(\rho(n^k G)) &\leq \gamma_{\min}(n^k G) + N^{\epsilon} \\ &= n^k \gamma_{\min}(G) + n^{\epsilon(k+1)} \\ &\leq n^k \gamma_{\min}(G) + n^k - 1 \end{aligned} \tag{7.4}$$

On the other hand, if we let $\rho_1(G), \ldots, \rho_{n^k}(G)$ be the $n^k$ induced rotation systems of $G$ from $\rho(n^k G)$, then by Theorem 7.3.3

$$\gamma(\rho(n^k G)) = \sum_{i=1}^{n^k} \gamma(\rho_i(G)) \tag{7.5}$$

Combining Equations (7.4) and (7.5) and noticing that the genus of $\rho_i(G)$ is at least as large as $\gamma_{\min}(G)$ for all $1 \leq i \leq n^k$, we conclude that at least one induced rotation system $\rho_i(G)$ of $G$ achieves the minimum genus $\gamma_{\min}(G)$. This rotation system of $G$ can be found by calculating the genus for each induced rotation system $\rho_i(G)$ from $\rho(n^k G)$ and selecting the one with the smallest genus. This can be accomplished in polynomial time.

Therefore, using the algorithm $A$, we would be able to construct in polynomial time a minimum genus embedding for the graph $G$. Consequently, the GRAPH GENUS problem can be solved in polynomial time. Since the GRAPH GENUS problem is NP-hard, we would derive P = NP. □

The technique of Theorem 7.3.5 can be summarized as follows. Let $Q = \langle I_Q, S_Q, f_Q, opt_Q \rangle$ be an optimization problem such that there is an operator $\oplus$ implementable in polynomial time that can "compose" input instances, i.e., for any two input instances $x$ and $y$ of $Q$, $x \oplus y$ is also an input instance of $Q$ such that $|x \oplus y| = |x| + |y|$ (in the case of Theorem 7.3.5, $\oplus$ is the bar-amalgamation). Moreover, suppose that from a solution $s_{x \oplus y}$ to the instance $x \oplus y$, we can construct in polynomial time solutions $s_x$ and $s_y$ for the instances $x$ and $y$, respectively such that

$$f_Q(x \oplus y, s_{x \oplus y}) = f_Q(x, s_x) + f_Q(y, s_y)$$

(this corresponds to Theorem 7.3.3) and

$$Opt(x \oplus y) = Opt(x) + Opt(y)$$

(this corresponds to Corollary 7.3.4), then using the technique of Theorem 7.3.5, we can prove that the problem $Q$ cannot be approximated in polynomial time with an additive difference $n^\epsilon$ for any constant $\epsilon < 1$ unless $Q$ can be solved precisely in polynomial time. In particular, if $Q$ is NP-hard, then $Q$ cannot be approximated in polynomial time with an additive difference $n^\epsilon$ for any constant $\epsilon < 1$ unless P = NP.

# Chapter 8

# Polynomial Time Approximation Schemes

A fully polynomial time approximation scheme for an NP-hard optimization problem $Q$ seems the best we can hope in approximation of optimal solutions to the problem $Q$: we can approximate an optimal solution to $Q$ with an approximation ratio $1 + \epsilon$ arbitrarily close to 1 in time polynomial in the input length and in the reciprocal of the error bound $\epsilon$. Unfortunately, Theorem 6.4.1 immediately excludes the possibility of having fully polynomial time approximation schemes for many NP-hard optimization problems. In particular, a large class of NP-hard optimization problems is of the type of "subset problem", which ask to select a largest or smallest subset satisfying certain properties from a given set of objects. Most optimization problems related to graphs, such as the INDEPENDENT SET problem and the VERTEX COVER problem, belong to this class. Note that any such an NP-hard subset problem automatically satisfies the conditions in Theorem 6.4.1, thus has no fully polynomial time approximation scheme unless P = NP.

With the understanding that a given optimization problem $Q$ is unlikely to have a fully polynomial time approximation scheme, we are still interested in whether we can approximate in polynomial time optimal solutions to $Q$ with approximation ratio arbitrarily close to 1. More precisely, for each *fixed* constant $\epsilon$, we are interested in knowing whether there is an approximation algorithm for $Q$ with approximation ratio bounded by $1 + \epsilon$ whose running time is bounded by a polynomial (of the input length but not necessarily of $1/\epsilon$). Note that neither Theorem 6.4.1 nor Theorem 6.4.8 excludes the possibility of having this kind of approximation algorithms for an optimization problem $Q$ if even $Q$ satisfies the conditions in the corresponding theorem.

**Definition 8.0.1** An optimization problem $Q$ has a *polynomial time approximation scheme* (PTAS), if for any fixed constant $\epsilon > 0$, there is a polynomial time approximation algorithm for $Q$ with approximation ratio bounded by $1 + \epsilon$.

In particular, an optimization problem with a fully polynomial time approximation scheme has polynomial time approximation schemes.

In this chapter, we present polynomial time approximation schemes for a number of well-known optimization problems, including the MAKESPAN problem, the PLANAR GRAPH INDEP-SET problem, and the EUCLIDEAN TRAVELING SALESMAN problem. These problems, according to Theorem 6.4.1 and Theorem 6.4.8, have no fully polynomial time approximation schemes unless P = NP. Therefore, polynomial time approximation schemes seem the best approximation we can expect for these problems.

Two techniques are introduced in developing polynomial time approximation schemes for these optimization problems. In Section 8.1, we introduce the technique of *dual approximation* that leads to a polynomial time approximation scheme for the MAKESPAN problem. In the next two sections, we take the advantage of balanced separability of planar graphs and use a "separate-approximate" technique to develop polynomial time approximation schemes for the PLANAR GRAPH INDEP-SET problem and for the EUCLIDEAN TRAVELING SALESMAN problem.

We should point out that though most (non-fully) polynomial time approximation schemes for optimization problems are of great theoretical importance, they are not very practical for small error bound $\epsilon$. Typically, the running time of this kind of algorithms is proportional to at least $2^{1/\epsilon}$ or even to $n^{1/\epsilon}$, which is an enormous number when $\epsilon$ is small.

## 8.1 The Makespan problem

Recall that the MAKESPAN problem is defined 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,

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

$opt_Q$: min

We review a few properties for the MAKESPAN problem:

- The problem is NP-hard;
- Even if we fix the number $m$ of processors to be any constant larger than 1, the problem remains NP-hard (Theorem 5.1.3);
- If the number $m$ of processors is a fixed constant, then the problem has a fully polynomial time approximation scheme (Corollary 6.2.3).
- The general MAKESPAN problem in which the number of processors $m$ is given as a variable in the input instances is NP-hard in the strong sense and has no fully polynomial time approximation scheme unless P = NP (Theorem 6.4.9).

In the following, we first study a variation of the BIN PACKING problem, which is "dual" to the MAKESPAN problem. Based on an efficient algorithm for this problem, we will develop a polynomial time approximation scheme for the MAKESPAN problem.

### 8.1.1 The $(1 + \epsilon)$-BIN PACKING problem

Recall that the BIN PACKING problem is to pack $n$ given objects of sizes $s_1$, $s_2$, ..., $s_n$, respectively, into the minimum number of bins of a given size $B$.

The MAKESPAN problem can be regarded as a variant version of the BIN PACKING problem in which we are given $n$ objects of sizes $t_1$, ..., $t_n$, respectively, and the number $m$ of bins, and we are asked to pack the objects into the $m$ bins such that the bin size is minimized. Therefore, there are two parameters: the number of bins and the bin size. Each of the MAKESPAN problem and the BIN PACKING problem fixes one parameter and optimizes the other parameter. In this sense, the MAKESPAN problem is "dual" to the BIN PACKING problem. Therefore, it is not very surprising that the techniques developed for approximation algorithms for the BIN PACKING problem can be useful in deriving approximation algorithms for the MAKESPAN problem.

Consider the following variation of the BIN PACKING problem, which we call the $(1 + \epsilon)$-BIN PACKING problem. Each instance $\alpha$ of this problem is also an instance of the BIN PACKING problem, where we use $Opt_B(\alpha)$ to denote the optimal value of $\alpha$ as an instance for the BIN PACKING problem.

> $(1+\epsilon)$-BIN PACKING
>
> INPUT: $\alpha = \langle t_1, t_2, \ldots, t_n; B\rangle$, all integers
>
> OUTPUT: a packing of the $n$ objects of sizes $t_1$, $t_2$, …, $t_n$, respectively, into at most $Opt_B(\alpha)$ bins such that the content of each bin is bounded by $(1+\epsilon)B$

Therefore, instead of asking to minimize the number of bins used in the packing, the $(1+\epsilon)$-BIN PACKING problem fixes the number of bins to $Opt_B(\alpha)$ (note that the number $Opt_B(\alpha)$ is *not* given in the input instance $\alpha$) and allows the size of the bins to "exceed" by an $\epsilon$ factor. Note that any optimal solution $Y_B$ to $\alpha$ as an instance for the BIN PACKING problem is also a solution to $\alpha$ as an instance for the $(1+\epsilon)$-BIN PACKING since $Y_B$ uses exactly $Opt_B(\alpha)$ bins and no bin has content exceeding $B$.

We first show that the $(1+\epsilon)$-BIN PACKING problem can be solved in polynomial time for a fixed constant $\epsilon > 0$. Then we show how this solution can be used to derive a polynomial time approximation scheme for the MAKESPAN problem.

The idea for solving the $(1+\epsilon)$-BIN PACKING problem is very similar to the one for the approximation algorithm **ApxBinPack** for the general BIN PACKING problem (see subsection 7.1.3, Figure 7.5). We first perform two preprocessing steps:

1. ignore the objects of size less than $\epsilon B$; and
2. partition the rest of the objects into $\pi$ groups $G_1$, …, $G_\pi$ so that the objects in each group have a very small difference in size. For each group $G_i$, replace every object by the one with the *smallest* size in $G_i$.

The preprocessing steps give us an instance $\alpha'$ of the $(\epsilon, \pi)$-BIN PACKING problem (see subsection 7.1.2), for which an optimal solution can be constructed in polynomial time for fixed constants $\epsilon$ and $\pi$ (see Theorem 7.1.6). Note that the optimal solution for $\alpha'$ is an "under-estimation" of the optimal solution for $\alpha$ and thus it uses no more than $Opt_B(\alpha)$ bins. Then we restore the object sizes and add the small objects by greedy method to get a packing for the instance $\alpha$. Since the difference in sizes of the objects in each group $G_i$ is very small, the restoring of object sizes will not increase the content for each bin very much. Moreover, adding small objects using greedy method will not induce much unbalancing in the packing.

The formal algorithm is given in Figure 8.1.

According to Theorem 7.1.6 and note that $\pi = \lceil 1/\epsilon^2 \rceil$, the $(\epsilon, \pi)$-BIN PACKING problem can be solved in time $O(n) + h_0(\epsilon)$, where $h_0(\epsilon)$ is a function depending only on $\epsilon$. We conclude that the algorithm **VaryBinPack**

**Algorithm. VaryBinPack**

Input: $\alpha = \langle t_1, \ldots, t_n; B\rangle$ and $\epsilon > 0$
Output: a packing of $t_1$, $t_2$, ..., $t_n$ into $Opt_B(\alpha)$ bins of size $(1+\epsilon)B$

1. sort $t_1, \ldots, t_n$; without loss of generality, let $t_1 \geq t_2 \geq \cdots \geq t_n$;
2. let $h$ be the largest index such that $t_h > \epsilon B$; $\alpha_0 = \langle t_1, t_2, \ldots, t_h; B\rangle$;
3. $\pi = \lceil 1/\epsilon^2 \rceil$, divide $(\epsilon B, B]$ into $\pi$ subsegments of equal length $(l_1, h_1]$, $(l_2, h_2]$, ..., $(l_\pi, h_\pi]$, where $h_i = l_{i+1}$ and $h_i - l_i = (B - \epsilon B)/\pi$ for all $i$, and $l_1 = \epsilon B$ and $h_\pi = B$;
4. partition the objects in $\alpha_0$ into $\pi$ groups $G_1$, ..., $G_\pi$, such that an object is in group $G_i$ if and only if its size is in the range $(l_i, h_i]$; let $m_i$ be the number of objects in $G_i$;
5. construct an optimal solution $Y'$ to the instance $\alpha' = \langle l_1 : m_1, \ldots, l_\pi : m_\pi; B\rangle$ for the $(\epsilon, \pi)$-Bin Packing problem;
6. **for** $i = 1$ **to** $\pi$ **do**
   replace the $m_i$ objects of size $l_i$ in the packing $Y'$ by the $m_i$ objects in group $G_i$ in the instance $\alpha_0$ (in an arbitrary order);
   This gives a packing $Y_0$ for the instance $\alpha_0$;
7. add the objects $t_{h+1}, \ldots, t_n$ in $\alpha$ to the packing $Y_0$ by greedy method (i.e., no new bin will be used until adding the current object to any used bins would exceed the size $(1+\epsilon)B$). This results in a packing $Y$ for $\alpha$ as an instance for the $(1+\epsilon)$-Bin Packing problem.

Figure 8.1: The **VaryBinPack** algorithm

runs in time $O(n \log n) + h_0(\epsilon)$, if an $O(n \log n)$ time sorting algorithm is used for step 1.

Recall that we denote by $Opt_B(\alpha)$ the optimal value, i.e., the number of bins used by an optimal packing, of the input instance $\alpha$ for the general Bin Packing problem. We first show that the algorithm **VaryBinPack** results in a packing for the objects in $\alpha$ such that the packing uses no more than $Opt_B(\alpha)$ bins.

**Lemma 8.1.1** *The packing $Y'$ for the instance $\alpha'$ constructed by step 5 of the algorithm* **VaryBinPack** *uses no more than $Opt_B(\alpha)$ bins.*

Proof. The instance $\alpha_0$ is a subset of the instance $\alpha$. Thus, $Opt_B(\alpha_0) \leq Opt_B(\alpha)$. The instance $\alpha'$ can be regarded as obtained from the instance $\alpha_0$ by replacing each object in group $G_i$ by an object of smaller size $l_i$, for all $i = 1, \ldots, \pi$. Thus, $Opt_B(\alpha') \leq Opt_B(\alpha_0) \leq Opt_B(\alpha)$. Since $Y'$ is an optimal packing for the instance $\alpha'$, $Y'$ uses $Opt_B(\alpha') \leq Opt_B(\alpha)$ bins. □

**Lemma 8.1.2** *In the packing $Y_0$ for the instance $\alpha_0$ constructed by step 6 of the algorithm* **VaryBinPack**, *no bin has content larger than $(1+\epsilon)B$, and $Y_0$ uses no more than $Opt_B(\alpha)$ bins.*

Proof. According to step 6 of the algorithm **VaryBinPack**, the number of bins used by the packing $Y_0$ for the instance $\alpha_0$ is the same as that used by $Y'$ for the instance $\alpha'$. By Lemma 8.1.1, the packing $Y_0$ uses no more than $Opt_B(\alpha)$ bins.

Each object of size $l_i$ in the instance $\alpha'$ corresponds to an object of size between $l_i$ and $h_i$ in group $G_i$ in the instance $\alpha_0$. The packing $Y_0$ for $\alpha_0$ is obtained from the packing $Y'$ for $\alpha'$ by replacing each object of $\alpha'$ by the corresponding object in $\alpha_0$. Since no object in group $G_i$ has size larger than $h_i$ and

$$h_i - l_i = (B - \epsilon B)/\pi$$

the size increase for each object from $Y'$ to $Y_0$ is bounded by $(B - \epsilon B)/\pi$.

Moreover, since all objects in $\alpha'$ have size larger than $\epsilon B$, and the packing $Y'$ has bin size $B$, each bin in the packing $Y'$ holds at most $\lfloor 1/\epsilon \rfloor$ objects. Therefore, the size increase for each bin from $Y'$ to $Y_0$ is bounded by

$$\begin{aligned}
& ((B - \epsilon B)/\pi) \cdot \lfloor 1/\epsilon \rfloor \\
= \; & ((B - \epsilon B)/\lceil 1/\epsilon^2 \rceil) \cdot \lfloor 1/\epsilon \rfloor \\
\leq \; & (B/(1/\epsilon^2)) \cdot (1/\epsilon) \\
= \; & \epsilon B
\end{aligned}$$

Since the content of each bin of the packing $Y'$ is at most $B$, we conclude that the content of each bin of the packing $Y_0$ is at most $(1+\epsilon)B$. $\square$

Now we can show that the algorithm **VaryBinPack** constructs a solution to the $(1+\epsilon)$-BIN PACKING problem.

**Lemma 8.1.3** *The packing $Y$ for the instance $\alpha$ constructed by step 7 of the algorithm* **VaryBinPack** *uses no more than $Opt_B(\alpha)$ bins, and each bin of $Y$ has content at most $(1+\epsilon)B$.*

PROOF. By Lemma 8.1.2, each bin of the packing $Y_0$ has content at most $(1+\epsilon)B$. The packing $Y$ is obtained from $Y_0$ by adding the objects of size bounded by $\epsilon B$ using greedy method. That is, suppose we want to add an object of size not larger than $\epsilon B$ and there is a used bin whose content will not exceed $(1+\epsilon)B$ after adding the object to the bin, then we add the object to the bin. A new bin is introduced only if no used bin can have the object added without exceeding the content $(1+\epsilon)B$. The greedy method ensures that the content of each bin in $Y$ is bounded by $(1+\epsilon)B$. Note that since all added objects have size bounded by $\epsilon B$, when a new bin is introduced, all used bins have content larger than $B$.

If no new bin was introduced in the process of adding small objects in step 7, then the number of bins used by the packing $Y$ is the same as the number of bins used by the packing $Y_0$. By Lemma 8.1.2, in this case the packing $Y$ uses no more than $Opt_B(\alpha)$ bins.

Now suppose that new bins were introduced in the process of adding small objects in step 7. Let $r$ be the number of bins used by the packing $Y$. By the above remark, at least $r-1$ bins in the packing $Y$ have content larger than $B$. Therefore, we have

$$t_1 + \cdots + t_n > B(r-1)$$

This shows that we need more than $r-1$ bins of size $B$ to pack the objects in $\alpha$ in any packing. Consequently, the value $Opt_B(\alpha)$ is at least $(r-1)+1 = r$. That is, the packing $Y$ uses no more than $Opt_B(\alpha)$ bins. $\square$

We conclude this subsection with the following theorem.

**Theorem 8.1.4** *Given an instance $\alpha = \langle t_1, \ldots, t_n; B\rangle$ for the* BIN PACKING *problem and a constant $\epsilon > 0$, The algorithm* **VaryBinPack** *constructs in time $O(n \log n) + h_0(\epsilon)$ a packing for $\alpha$ that uses no more than $Opt_B(\alpha)$ bins and the content of each bin is bounded by $(1+\epsilon)B$, where $h_0(\epsilon)$ is a* *function depending only on $\epsilon$ and $Opt_B(\alpha)$ is the optimal value for $\alpha$ as an instance for the* Bin Packing *problem.*

**Corollary 8.1.5** *The $(1+\epsilon)$-*Bin Packing *problem can be solved in polynomial time for a fixed constant $\epsilon$.*

### 8.1.2 A PTAS for Makespan

We use the algorithm **VaryBinPack** to develop a polynomial time approximation scheme for the Makespan problem. We first re-formulate the Makespan problem in the language of bin packing.

> Makespan (Bin Packing version)
>
> Input: $\langle t_1, t_2, \ldots, t_n; m\rangle$, all integers, where $t_i$ is the size of the $i$th object, for $i = 1, 2, \ldots, n$
>
> Output: a packing of the $n$ objects into $m$ bins of size $B$ with $B$ minimized

We use the idea of binary search to find the minimum bin size $B$. In general, suppose that we try bin size $B$, and find out that the input instance $\langle t_1, \ldots, t_n; B\rangle$ for the Bin Packing problem needs more than $m$ bins of size $B$ in its optimal packing, then the tried bin size $B$ is too small. So we will try a larger bin size. On the other hand, if the instance $\langle t_1, \ldots, t_n; B\rangle$ needs no more than $m$ bins of size $B$, then we may want to try a smaller bin size because we are minimizing the bin size. Note that the algorithm **VaryBinPack** can be used to estimate the number of bins used by an optimal packing for the instance $\langle t_1, \ldots, t_n; B\rangle$ for the Bin Packing problem.

We first discuss the initial bounds for the bin size in the binary search. Fix an input instance $\langle t_1, \ldots, t_n; m\rangle$ for the Makespan problem. Let

$$B_{avg} = \max\{\sum_{i=1}^{n} t_i/m, t_1, t_2, \ldots, t_n\}$$

**Lemma 8.1.6** *The minimum bin size of the input instance $\langle t_1, \ldots, t_n; m\rangle$ for the* Makespan *problem is at least $B_{avg}$.*

Proof. Since $\sum_{i=1}^{n} t_i/m$ is the average content of the $m$ bins for packing the $n$ objects of size $t_1$, ..., $t_n$, any packing of the $n$ objects into the $m$ bins has at least one bin with content at least $\sum_{i=1}^{n} t_i/m$. That is, the bin size of the packing is at least $\sum_{i=1}^{n} t_i/m$.

Moreover, the bin size of the packing should also be at least as large as any $t_i$ since every object has to be packed into a bin in the packing.

This shows that for any packing of the $n$ objects of size $t_1, \ldots, t_n$ into the $m$ bins, the bin size is at least $B_{avg}$. The lemma is proved. $\square$

This gives a lower bound on the bin size for the input instance $\langle t_1, \ldots, t_n; m \rangle$ of the MAKESPAN problem. We also have the following upper bound.

**Lemma 8.1.7** *The minimum bin size of the input instance $\langle t_1, \ldots, t_n; m \rangle$ for the* MAKESPAN *problem is bounded by $2B_{avg}$.*

PROOF. Suppose that the lemma is false. Let $B$ be the minimum bin size for packing the objects $t_1, \ldots, t_n$ into $m$ bins, and $B > 2B_{avg}$.

Let $Y$ be a packing of the objects $t_1, \ldots, t_n$ into $m$ bins such that the bin size of $Y$ is $B$. Furthermore, we suppose that $Y$ is the packing in which the fewest number of bins have content $B$. Let $\beta_1, \beta_2, \ldots, \beta_m$ be the bins used by $Y$, where the bin $\beta_1$ has content $B > 2B_{avg}$. Then at least one of the bins $\beta_2, \ldots, \beta_m$ has content less than $B_{avg}$ — otherwise, the sum of total contents of the bins $\beta_1, \beta_2, \ldots, \beta_m$ would be larger than $mB_{avg} \geq \sum_{i=1}^{n} t_i$. Without loss of generality, suppose that the bin $\beta_2$ has content less than $B_{avg}$. Now remove any object $t_i$ from the bin $\beta_1$ and add $t_i$ to the bin $\beta_2$. We have

1. the content of the bin $\beta_1$ in the new packing is less than $B$;

2. the content of the bin $\beta_2$ in the new packing is less than

$$B_{avg} + t_i \leq 2B_{avg} < B$$

3. the contents of the other bins are unchanged.

Thus, in the new packing, the number of bins that have content $B$ is one less than the number of bins of content $B$ in the packing $Y$. This contradicts our assumption that $Y$ has the fewest number of bins of content $B$.

This contradiction proves the lemma. $\square$

Therefore, the minimum bin size for packing the objects $t_1, \ldots, t_n$ into $m$ bins is in the range $[B_{avg}, 2B_{avg}]$. We apply binary search on this range to find an approximation for the optimal solution to the instance $\langle t_1, \ldots, t_n; m \rangle$ for the MAKESPAN problem. Consider the algorithm given in Figure 8.2.

**Algorithm. ApxMakespan**

Input: $\alpha = \langle t_1, \ldots, t_n; m\rangle$, all integers, and $\epsilon > 0$
Output: a scheduling of the $n$ jobs of processing time $t_1$, $t_2$, ..., $t_n$
on $m$ identical processors

1. $B_{avg} = \max\{\sum_{i=1}^{n} t_i/m, t_1, t_2, \ldots, t_n\}$;
2. $B_L = B_{avg}$; $B_H = 2B_{avg}$;
3. **while** $B_H - B_L > \epsilon B_{avg}/4$ **do**
   $B = \lfloor (B_L + B_H)/2 \rfloor$;
   call the algorithm **VaryBinPack** on input $\alpha = \langle t_1, \ldots, t_n; B\rangle$
   and $\epsilon/4$; suppose the algorithm uses $r$ bins on the input;
   **if** $r > m$ **then** $B_L = B$ **else** $B_H = B$;
4. let $B^* = (1 + \epsilon/4)B_H$;
5. call the algorithm **VaryBinPack** on input $\alpha = \langle t_1, \ldots, t_n; B^*\rangle$
   and $\epsilon/4$ to construct a scheduling for the instance $\alpha$ for the
   MAKESPAN problem.

Figure 8.2: The algorithm **ApxMakespan**

We first study the complexity of the algorithm **ApxMakespan**. The running time of the algorithm is dominated by step 3. We start with

$$B_H - B_L = 2B_{avg} - B_{avg} = B_{avg}$$

Since we are using binary search, each execution of the body of the **while** loop in step 3 will half the difference $(B_H - B_L)$. Therefore, after $O(\log(1/\epsilon))$ executions of the body of the **while** loop in step 3, we must have

$$B_H - B_L \le \epsilon B_{avg}/4$$

That is, the body of the **while** loop in step 3 is executed at most $O(\log(1/\epsilon))$ times.

In each execution of the body of the **while** loop in step 3, we call the algorithm **VaryBinPack** on input $\langle t_1, \ldots, t_n; B\rangle$ and $\epsilon/4$, which takes time $O(n \log n) + h_0(\epsilon/4) = O(n \log n) + h_1(\epsilon)$, where $h_1(\epsilon)$ is a function depending only on $\epsilon$. Therefore, the running time of the algorithm **ApxMakespan** is bounded by

$$O(\log(1/\epsilon))(O(n \log n) + h_1(\epsilon)) = O(n \log n \log(1/\epsilon)) + h_2(\epsilon)$$

where $h_2(\epsilon)$ is a function depending only on $\epsilon$. This concludes the following lemma.

**Lemma 8.1.8** *The running time of the algorithm* **ApxMakespan** *on input instance* $\alpha = \langle t_1, \ldots, t_n; m \rangle$ *and* $\epsilon > 0$ *is bounded by* $O(n \log n \log(1/\epsilon)) + h_2(\epsilon)$*, where* $h_2(\epsilon)$ *is a function depending only on* $\epsilon$*. In particular, for a fixed constant* $\epsilon > 0$*, the algorithm* **ApxMakespan** *runs in polynomial time.*

Now we discuss the approximation ratio of the algorithm **ApxMakespan**.

Fix an input instance $\alpha = \langle t_1, \ldots, t_n; m \rangle$ for the MAKESPAN problem. Let $Opt(\alpha)$ be the optimal solution, i.e., the parallel finish time of an optimal scheduling, of the instance $\alpha$ for the MAKESPAN problem.

**Lemma 8.1.9** *For any input instance* $\alpha$*, the following relations hold in the entire execution of the algorithm* **ApxMakespan**

$$B_L \leq Opt(\alpha) \leq (1 + \epsilon/4) B_H$$

PROOF. Initially, $B_L = B_{avg}$ and $B_H = 2B_{avg}$. By Lemmas 8.1.6 and 8.1.7, we have $B_L \leq Opt(\alpha) \leq B_H < (1 + \epsilon/4) B_H$.

Now for each execution of the **while** loop in step 3, we start with a bin size $B$ and call the algorithm **VaryBinPack** on the input $\langle t_1, \ldots, t_n; B \rangle$ and $\epsilon/4$, which uses $r$ bins.

If $r > m$, by the algorithm **VaryBinPack**, the minimum number of bins used by a packing to pack the objects $t_1, \ldots, t_n$ into bins of size $B$ is at least as large as $r$. Therefore, if the bin size is $B$, then we need more than $m$ bins to pack the objects $t_1, \ldots, t_n$. Thus, for packing the objects $t_1, \ldots, t_n$ into $m$ bins, the bin size $B$ is too small. That is, $Opt(\alpha) > B$. Since in this case we set $B_L = B$ and unchange $B_H$, the relations $B_L \leq Opt(\alpha) \leq (1+\epsilon/4) B_H$ still hold.

If $r \leq m$, then the objects $t_1, \ldots, t_n$ can be packed in $r$ bins of size $(1 + \epsilon/4)B$. Certainly, the objects can also be packed in $m$ bins of size $(1 + \epsilon/4)B$. This gives $Opt(\alpha) \leq (1 + \epsilon/4)B$. Thus, setting $B_H = B$ and unchanging $B_L$ still keep the relations $B_L \leq Opt(\alpha) \leq (1 + \epsilon/4) B_H$.

This proves the lemma. □

Now we are ready to show that the algorithm **ApxMakespan** is a polynomial time approximation scheme for the MAKESPAN problem.

**Theorem 8.1.10** *On any input instance* $\alpha = \langle t_1, \ldots, t_n; m\rangle$ *for the* MAKESPAN *problem and for any* $\epsilon$, $0 < \epsilon \leq 1$, *the algorithm* **ApxMakespan** *constructs in time* $O(n \log n \log(1/\epsilon)) + h_2(\epsilon)$ *a solution to* $\alpha$ *with approximation ratio* $1 + \epsilon$, *where* $h_2(\epsilon)$ *is a function depending only on* $\epsilon$.

PROOF. The time complexity of the algorithm **ApxMakespan** is given in Lemma 8.1.8.

By Lemma 8.1.9, the relations

$$B_L \leq Opt(\alpha) \leq (1 + \epsilon/4)B_H$$

always hold. In particular, at step 4 of the algorithm, we have

$$Opt(\alpha) \leq (1 + \epsilon/4)B_H = B^*$$

Therefore, the bin size $B^*$ is at least as large as the bin size $Opt(\alpha)$. Since the objects $t_1, \ldots, t_n$ can be packed into $m$ bins of size $Opt(\alpha)$, we conclude that the objects $t_1, \ldots, t_n$ can also be packed into $m$ bins of size $B^*$. By the property of the algorithm **VaryBinPack**, on input instance $\alpha = \langle t_1, \ldots, t_n; B^*\rangle$ and $\epsilon/4$, the algorithm **VaryBinPack** packs the objects $t_1, \ldots, t_n$ into at most $m$ bins, with each bin of content at most $(1 + \epsilon/4)B^*$. Therefore, the packing is a valid scheduling of the $n$ jobs $t_1, \ldots, t_n$ on $m$ identical processors.

Now let us consider the content bound $(1 + \epsilon/4)B^*$ for the bins in the packing constructed by the algorithm **VaryBinPack**. At step 4, we have

$$B_H - B_L \leq \epsilon B_{avg}/4$$

Since $B_L = B_{avg}$ initially, and $B_L$ is never decreased, we have

$$B_H \leq B_L + \epsilon B_{avg}/4 \leq B_L + \epsilon B_L/4 = (1 + \epsilon/4)B_L$$

By Lemma 8.1.9, we always have

$$B_L \leq Opt(\alpha) \leq (1 + \epsilon/4)B_H$$

Thus

$$B_H \leq (1 + \epsilon/4)B_L \leq Opt(\alpha)(1 + \epsilon/4)$$

Therefore, the content bound $(1 + \epsilon/4)B^*$ is bounded by (note that $B^* = (1 + \epsilon/4)B_H$)

$$(1 + \epsilon/4)B^* = (1 + \epsilon/4)^2 B_H \leq Opt(\alpha)(1 + \epsilon/4)^3$$

Now $Opt(\alpha)(1+\epsilon/4)^3 \leq Opt(\alpha)(1+\epsilon)$ for $\epsilon \leq 1$. Recall that in the scheduling, the number $m$ of bins corresponds to the number of processors, and the maximum bin content $(1+\epsilon/4)B^*$ is at least as large as the parallel finish time for the scheduling. In conclusion, the scheduling of the $n$ jobs on the $m$ processors constructed by the algorithm **ApxMakespan** has parallel finish time bounded by $Opt(\alpha)(1+\epsilon)$. Since $Opt(\alpha)$ is the parallel finish time of an optimal scheduling for the instance $\alpha$, the algorithm **ApxMakespan** has approximation ratio bounded by $1+\epsilon$. □

We point out that the condition $\epsilon \leq 1$ in Theorem 8.1.10 is not crucial. For any input instance $\alpha = \langle t_1, \ldots, t_n; m\rangle$ and a given constant $\epsilon > 1$, we can apply the algorithm **Graham-Schedule** (see Section 5.3), which runs in time $O(n \log m) = O(n \log n)$ and has approximation ratio $2 - \frac{1}{m} \leq 1+\epsilon$ for $\epsilon > 1$.

Therefore, for any input instance $\alpha$ for the MAKESPAN problem and any constant $\epsilon > 0$, we can first check if $\epsilon > 1$. If so, we apply the algorithm **Graham-Schedule** to construct in time $O(n \log n)$ a solution to $\alpha$ of approximation ratio bounded by $1+\epsilon$. Otherwise, we apply the algorithm **ApxMakespan** to construct in time $O(n \log n \log(1/\epsilon)) + h_2(\epsilon)$ a solution to $\alpha$ of approximation ratio bounded by $1+\epsilon$, where $h_2(\epsilon)$ is a function depending only on $\epsilon$. The following theorem summarizes this discussion.

**Theorem 8.1.11** *The* MAKESPAN *problem has a polynomial time approximation scheme.*

We would like to give two final remarks before we close this section. The algorithm **ApxMakespan**, whose running time is bounded by $O(n \log n \log(1/\epsilon)) + h_2(\epsilon)$, is not a fully polynomial time approximation scheme because the function $h_2(\epsilon)$ is not bounded by any polynomial of $1/\epsilon$. In fact, according to Theorem 6.4.9, it is very unlikely that the MAKESPAN problem has a fully polynomial time approximation scheme.

On the other hand, it is still possible to further improve the running time of the algorithm **ApxMakespan**. For example, based on the same idea of dual approximation, Hochbaum and Shmoys have developed an approximation algorithm for the MAKESPAN problem with approximation ratio $1+\epsilon$ and running time bounded by $O(n) + h_3(\epsilon)$, where the constant in the "big-Oh" is independent of the approximation error bounde $\epsilon$ and the function $h_3(\epsilon)$ depends only on $\epsilon$. Interested readers are referred to [65].

## 8.2 Optimization on planar graphs

A graph is *planar* if it can be embedded into the plane without edge crossing. There is a well-known linear time algorithm by Hopcroft and Tarjan [68] that, given a graph, either constructs a planar embedding of the graph or reports that the graph is not planar. Planar graphs are of great practical interest (e.g, in designing integrated circuits or printed-circuit boards).

In this section, we consider optimization problems on planar graphs. Some NP-hard optimization problems on general graphs become tractable when they are restricted to planar graphs. For example, the CLIQUE problem (given a graph $G$, find a largest clique, i.e., a largest subset $S$ of vertices in $G$ such that every pair of vertices in $S$ are adjacent), which in general is NP-hard, can be solved in polynomial time on planar graphs, as follows. According to Kuratowski's theorem [58], a planar graph contains no clique of size larger than 4. Therefore, we can simply check every set of at most four vertices in a planar graph and find the largest clique. Since for a planar graph of $n$ vertices, there are

$$\binom{n}{4} + \binom{n}{3} + \binom{n}{2} + \binom{n}{1} = O(n^4)$$

different sets of at most 4 vertices, the largest clique in the planar graph can be found in time $O(n^4)$.

On the other hand, some NP-hard optimization problems on general graphs remain NP-hard even when they are restricted on planar graphs. Examples include the PLANAR GRAPH INDEP-SET problem and the PLANAR GRAPH VERTEX-COVER problem. Therefore, finding optimal solutions for these problems on planar graphs seems as hard as finding optimal solutions for the problems on general graphs. However, graph planarity does seem to make some of these problems easier in the sense that for a class of optimization problems on planar graphs, we can derive polynomial time approximation schemes, while the corresponding problem on general graphs have no polynomial time approximation scheme unless P = NP. A general technique has been developed to obtain polynomial time approximation schemes for a class of NP-hard optimization problems on planar graphs. For purposes of discussion, we will focus on the following problem to illustrate the technique, where by an *independent set* $D$ in a graph $G$, we mean a subset of vertices in $G$ in which no two vertices are adjacent:

PLANAR GRAPH INDEP-SET

$I_Q$: the set of planar graphs $G = (V, E)$

$S_Q$: $S_Q(G)$ is the collection of all independent sets in $G$

$f_Q$: $f_Q(G, D)$ is equal to the number of vertices in the independent set $D$ in $G$

$opt_Q$: max

The decision version PLANAR GRAPH INDEP-SET (D) of the PLANAR GRAPH INDEP-SET problem is NP-complete (see Section 1.4). Thus, it is straightforward to derive that the PLANAR GRAPH INDEP-SET problem is NP-hard. Moreover, it is easy to check that the PLANAR GRAPH INDEP-SET problem satisfies the conditions in Theorem 6.4.1. Therefore, the PLANAR GRAPH INDEP-SET problem has no fully polynomial time approximation scheme unless P = NP.

A (non-fully) polynomial time approximation scheme is obtained for the PLANAR GRAPH INDEP-SET problem, and for many other optimization problems on planar graphs, using the popular divide-and-conquer method based on the following *Planar Separator Theorem* by Lipton and Tarjan [91]. Interested readers are referred to the original article [91] for a formal proof for this theorem.

**Theorem 8.2.1** (Planar Separator Theorem) *For any planar graph $G = (V, E)$, $|V| = n$, one can partition the vertex set $V$ of $G$ into three disjoint sets, $A$, $B$, and $S$, such that*

1. $|A|, |B| \leq 2n/3$;
2. $|S| \leq \sqrt{8n}$; *and*
3. *$S$ separates $A$ and $B$, i.e. there is no edge between $A$ and $B$.*

*Moreover, there is a linear time algorithm that, given a planar graph $G$, constructs the triple $(A, B, S)$ as above.*

Let $G = (V, E)$ be a planar graph and let $(A, B, S)$ be a triple satisfying the conditions of Theorem 8.2.1. We will say that *the graph $G$ is split into two smaller pieces $A$ and $B$* (using the separator $S$). Let $G_A$ be the subgraph of $G$ induced by the vertex set $A$, that is, $G_A$ is the subgraph of $G$ that consists of all vertices in the set $A$ and all edges whose both ends are in $A$. Similarly, let $G_B$ be the subgraph of $G$ induced by the vertex set $B$. Based on the fact that there is no edge in $G$ that connects a vertex in $A$ and a vertex in $B$, a simple observation is that if $D_A$ and $D_B$ are independent sets of the subgraphs $G_A$ and $G_B$, respectively, then the union

**Algorithm. PlanarIndSet**$(K)$

Input: a planar graph $G = (V, E)$
Output: an independent Set $D$ in $G$

1. **If** $(|V| \leq K)$ **then**
   find a maximum indepenent set $D$ in $G$ by exhaustive search;
   Return$(D)$;

$\{$At this point $|V| > K.\}$

2. split $V$ into $(A, B, S)$ as in Theorem 8.2.1;
3. recursively find an independent set $D_A$ for the subgraph $G_A$ and an independent set $D_B$ for the subgraph $G_B$;
4. Return$(D_A \cup D_B)$.

Figure 8.3: The algorithm **PlanarIndSet**

$D_A \cup D_B$ of the sets $D_A$ and $D_B$ is an independent set of the graph $G$. Moreover, since the size of a maximum independent set of the planar graph $G$ is of order $\Omega(n)$ (this will be formally proved later) while the size of the separator $S$ is of order $O(\sqrt{n})$, ignoring the vertices in the separator $S$ does not seem to lose too much precision. Based on this observation, our divide-and-conquer method first recursively finds a large independent set $D_A$ for the subgraph $G_A$ and a large independent set $D_B$ for the subgraph $G_B$ (note that the subgraphs $G_A$ and $G_B$ are also planar graphs), then uses the union $D_A \cup D_B$ as an approximation to the maximum independent set for the graph $G$. This algorithm is given informally in Figure 8.3, where $K$ is a constant to be determined later.

By the discussion above, the algorithm **PlanarIndSet** correctly returns an independent set for the graph $G$. Thus, it is an approximation algorithm for the PLANAR GRAPH INDEP-SET problem. We first study the properties of this algorithm.

The algorithm splits the graph $G$ into small pieces. If the size of a piece is larger than $K$, then the algorithm further splits, recursively, the piece into two smaller pieces in linear time according to Theorem 8.2.1. Otherwise, the algorithm finds a maximum independent set for the piece using brute force method. Let us consider the number of pieces whose size is within a certain region.

A piece is *at level* 0 if its size is not larger than $K$. For a general $i \geq 0$, a piece is *at level* $i$ if its size (i.e., the number of vertices in the piece) is in the region $((3/2)^{i-1}K, (3/2)^i K]$, i.e., if its size is larger than $(3/2)^{i-1}K$ but not larger than $(3/2)^i K$. Since the graph has $n$ vertices, the largest level number is bounded by $\log(n/K)/\log(3/2) = O(\log(n/K))$.

**Lemma 8.2.2** *For any fixed $i$, each vertex of the graph $G$ belongs to at most one piece at level $i$.*

PROOF. Fix a vertex $v$ of the graph $G$ and let $P$ be a piece containing the vertex $v$. Note that if $P$ is not the whole graph $G$, then $P$ must be obtained from splitting a larger piece.

Assume the contrary that the vertex $v$ is contained in two different pieces $P$ and $Q$ at level $i$. Then both $P$ and $Q$ have size larger than $(3/2)^{i-1}K$ but not larger than $(3/2)^i K$. Now consider the "splitting chains" for $P$ and $Q$:

$$P_1, P_2, \ldots, P_t, \qquad Q_1, Q_2, \ldots, Q_s$$

where $P_1 = P$, $Q_1 = Q$, $P_t = Q_s = G$, the piece $P_j$ is obtained from splitting the piece $P_{j+1}$ for $j = 1, \ldots, t-1$, the piece $Q_h$ is obtained from splitting the piece $Q_{h+1}$ for $h = 1, \ldots, s-1$, and all pieces $P_1$, …, $P_t$, $Q_1$, …, $Q_s$ contain the vertex $v$. Note that the piece $Q$ is not in the sequence $P_1$, $P_2$, …, $P_t$ since by Theorem 8.2.1, the piece $Q$ is split into two smaller pieces of size at most $(2/3)(3/2)^i K = (3/2)^{i-1}K$ while all pieces in the sequence $P_1$, $P_2$, …, $P_t$ have size at least as large as $|P|$, which is larger than $(3/2)^{i-1}K$. Similarly, the piece $P$ is not in the sequence $Q_1$, $Q_2$, …, $Q_s$. Let $j$ be the smallest index such that $P_j = Q_h$ for some $h$ (such an index $j$ must exist since $P_t = G = Q_s$). Then $P_j \neq P$ and $P_j \neq Q$, and by the assumption on the index $j$, we have $P_{j-1} \neq Q_{h-1}$. Therefore, the piece $P_j$ is split into two different pieces $P_{j-1}$ and $Q_{h-1}$. Now the fact that both pieces $P_{j-1}$ and $Q_{h-1}$ contain the vertex $v$ contradicts Theorem 8.2.1. $\square$

Therefore, for each fixed $i$, all pieces at level $i$ are disjoint. Since each piece at level $i$ consists of more than $(3/2)^{i-1}K$ vertices, there are no more than $(2/3)^{i-1}(n/K)$ pieces at level $i$, for all $i$. We summarize these discussions into the following facts.

> **Fact 1.** There are no more than $n$ pieces at level 0, each is of size at most $K$;
>
> **Fact 2.** For each fixed $i > 0$, there are no more than $(2/3)^{i-1}(n/K)$ pieces at level $i$, each is of size bounded by $(3/2)^i K$; and
>
> **Fact 3.** There are at most $O(\log n)$ levels.

Now we are ready to analyze the algorithm.

**Lemma 8.2.3** *The running time of the algorithm* **PlanarIndSet** *is bounded by* $O(n \log n + 2^K n)$.

PROOF. For each piece at level $i > 0$, we apply Theorem 8.2.1 to split it into two smaller pieces in time linear to the size of the piece. Since the total number of vertices belonging to pieces at level $i$ is bounded by $n$, we conclude that the total time spent by the algorithm **PlanarIndSet** on pieces at level $i$ is bounded by $O(n)$ for each $i > 0$. Since there are only $O(\log n)$ levels, the algorithm **PlanarIndSet** takes time $O(n \log n)$ on piece splitting.

For each piece $P$ at level 0, which has size bounded by $K$, the algorithm finds a maximum independent set by checking all subsets of vertices of the piece $P$. There are at most $2^K$ such subsets in $P$, and each such a subset can be checked in time linear to the size of the piece. Therefore, finding a maximum independent set in the piece $P$ takes time $O(2^K|P|)$. By Lemma 8.2.2, all pieces at level 0 are disjoint. We conclude that the algorithm **PlanarIndSet** spends time $O(2^K n)$ on pieces at level 0. In summary, the running time of the algorithm **PlanarIndSet** is bounded by $O(n \log n + 2^K n)$. □

Let us consider the approximation ratio for the algorithm **PlanarIndSet**.

Fix an $i > 0$. Suppose that we have $l$ pieces $P_1, P_2, \ldots, P_l$, of size $n_1, n_2, \ldots, n_l$, respectively, at level $i$. By Lemma 8.2.2 all these pieces are disjoint. Thus, $n_1 + n_2 + \cdots + n_l = n' \leq n$. For each piece $P_q$ of size $n_q$, a separator $S_q$ of size bounded by $\sqrt{8n_q} < 3\sqrt{n_q}$ is constructed to split the piece $P_q$ into two smaller pieces. The vertices in the separator $S_q$ will be ignored in the further consideration. Thus, the total number of vertices in the separators for the pieces $P_1, P_2, \ldots, P_l$ at level $i$, which will be ignored in the further consideration, is bounded by

$$3\sqrt{n_1} + 3\sqrt{n_2} + \cdots + 3\sqrt{n_l}$$

It is well-known that under the condition $n_1 + n_2 + \cdots + n_l = n'$ the above summation will be maximized when all $n_1, n_2, \ldots, n_l$ are equal. That is, $n_1 = n_2 = \cdots n_l = n'/l$. Hence, the above summation is bounded by

$$\underbrace{3\sqrt{n'/l} + 3\sqrt{n'/l} + \cdots + 3\sqrt{n'/l}}_{l \text{ terms}} = 3l\sqrt{n'/l} = 3\sqrt{n'l} \leq 3\sqrt{nl}$$

Now, since the number $l$ of pieces at level $i$ is bounded by $(2/3)^{i-1}(n/K)$ (see Fact 2), the total number of vertices belonging to separators for pieces at level $i$ is bounded by

$$3\sqrt{n \times \left(\frac{2}{3}\right)^{i-1} \frac{n}{K}} = \frac{3n}{\sqrt{K}} \left(\sqrt{\frac{2}{3}}\right)^{i-1}$$

Let $F$ denote the set of all vertices that belong to a separator at some level. Let $h$ be the largest level number. Then $h = O(\log n)$ (see Fact 3) and we derive

$$|F| \leq \sum_{i=1}^{h} \left(\frac{3n}{\sqrt{K}}\right) \left(\sqrt{\frac{2}{3}}\right)^{i-1} \leq \left(\frac{3n}{\sqrt{K}}\right) \sum_{i=1}^{\infty} \left(\sqrt{\frac{2}{3}}\right)^{i-1} = \frac{18n}{\sqrt{K}} \qquad (8.1)$$

where we have used the fact $\sum_{i=1}^{\infty}(\sqrt{2/3})^{i-1} = 1/(1-\sqrt{2/3}) \leq 6$.

Therefore, when the number $K$ is large enough, the total number of vertices contained in the separators is small compared with the total number $n$ of vertices in the graph $G$.

Now we derive an upper bound and a lower bound for the size of an optimal solution, i.e., a maximum independent set, to the planar graph $G$.

**Lemma 8.2.4** *Suppose that the planar graph $G$ has $n$ vertices. Let $D$ be the independent set constructed by the algorithm* **PlanarIndSet** *on input $G$ and let $F$ be the set of all vertices that are contained in any separators constructed by the algorithm* **PlanarIndSet**. *Then*

$$n/4 \leq Opt(G) \leq |D| + |F|$$

*where $Opt(G)$ is the size of a maximum independent set in the graph $G$.*

PROOF. Since the graph $G$ is planar, by the famous Four-Color Theorem [3, 4], $G$ can be colored with at most 4 colors such that no two adjacent vertices in $G$ are of the same color. It is easy to see that all vertices with the same color form an independent set for $G$, and that there are at least $n/4$ vertices in $G$ colored with the same color. In consequence, the size $Opt(G)$ of a maximum independent set in the graph $G$ is at least $n/4$.

Now we consider the upper bound for $Opt(G)$. Let $D_{\max}$ be a maximum independent set of the graph $G$ and let $P$ be a piece at level 0. It is easy to see that $D_{\max} \cap P$ is an independent set in the piece $P$, which cannot be larger than the maximum independent set $D^P_{\max}$ of $P$ constructed by the algorithm **PlanarIndSet**. Note that the independent set $D$ constructed by the algorithm **PlanarIndSet** is the union of $D^P_{\max}$ over all pieces at level 0. Let $\Gamma_0$ be the collection of all level 0 pieces. Note that

$$(\bigcup_{P\in\Gamma_0} P) \cup F$$

is the set of all vertices in the graph $G$ and the sets $\bigcup_{P\in\Gamma_0} P$ and $F$ are disjoint. Therefore,

$$D_{\max} = \bigcup_{P\in\Gamma_0} (D_{\max}\cap P) \cup (D_{\max}\cap F)$$

This gives (note that all level 0 pieces in $\Gamma_0$ are disjoint)

$$\begin{aligned} Opt(G) &= |D_{\max}| \le \sum_{P\in\Gamma_0} (|D_{\max}\cap P|) + |D_{\max}\cap F| \\ &\le \sum_{P\in\Gamma_0} |D^P_{\max}| + |F| \\ &= |D| + |F| \end{aligned}$$

The lemma is proved. $\Box$

Now we are ready to derive the approximation ratio for the algorithm **PlanarIndSet**. From Lemma 8.2.4, $Opt(G) \le |D| + |F|$. Thus,

$$\frac{Opt(G)}{|D|} \le 1 + \frac{|F|}{|D|} \le 1 + \frac{|F|}{Opt(G) - |F|}$$

Combining this with the inequalities $Opt(G) \ge n/4$ (see Lemma 8.2.4) and $|F| \le 18n/\sqrt{K}$ (see Inequality (8.1)), we obtain

$$\begin{aligned} \frac{Opt(G)}{|D|} &\le 1 + \frac{|F|}{Opt(G) - |F|} \le 1 + \frac{|F|}{(n/4) - |F|} \\ &\le 1 + \frac{18n/\sqrt{K}}{(n/4) - 18n/\sqrt{K}} = 1 + \frac{72}{\sqrt{K} - 72} \end{aligned}$$

Now for any fixed constant $\epsilon$, if we let

$$K \ge (72(1 + 1/\epsilon))^2 = 5184(1 + 1/\epsilon)^2$$

then the algorithm **PlanarIndSet**$(K)$ produces an independent set $D$ for the planar graph $G$ with approximation ratio

$$\frac{Opt(G)}{|D|} \le 1 + \epsilon$$

in time $O(n \log n + n2^{5184(1+1/\epsilon)^2})$ (see Lemma 8.2.3). For a fixed $\epsilon > 0$, this is a polynomial time algorithm. We conclude with the following theorem.

**Theorem 8.2.5** *The* PLANAR GRAPH INDEP-SET *problem has a polynomial time approximation scheme.*

Note that the algorithm **PlanarIndSet** is *not* a fully polynomial time approximation scheme for the PLANAR GRAPH INDEP-SET problem since its time complexity is not bounded by a polynomial of $n$ *and* $1/\epsilon$.

Other optimization problems on planar graphs that have polynomial time approximation schemes but have no fully polynomial time approximation schemes include the PLANAR GRAPH VERTEX-COVER problem, the PLANAR GRAPH H-MATCHING problem, the PLANAR GRAPH DOMINATING-SET problem, and some others (see [49] for precise definitions). Most of these polynomial time approximation scheme algorithms use the similar technique as the one we described for the PLANAR GRAPH INDEP-SET problem, i.e., using the divide-and-conquer method and the Planar Separator Theorem (Theorem 8.2.1) to separate a planar graph into small pieces by separators of small size, using brute force method to solve the problem for the small pieces, and combining the solutions to the small pieces into an approximation solution to the original planar graph.

Note that the algorithm **PlanarIndSet** of running time $O(n \log n + n2^{5184(1+1/\epsilon)^2})$ is hardly practical, even for a moderate value $\epsilon$. Research on improving the time complexity of polynomial time approximation schemes for optimization problems on planar graphs has performed. For example, a difference separating technique has been proposed by Baker [10]. We briefly describe the idea below based on the PLANAR GRAPH INDEP-SET problem. Let $G$ be a planar graph. Embed $G$ into the plane. Now the vertices on the unbounded face of the embedding give the first *layer* of the graph $G$. By peeling the first layer, i.e., deleting the vertices in the first layer, we obtain (maybe more than one) several separated pieces, each of which is a planar graph embedded in the plane. Now the first layers of these pieces form the second layer for the graph $G$. By peeling the second layer of $G$, we obtain the third layer, and so on. Define the *depth* of the planar graph $G$ to be the maximum number of layers of the graph. Baker observed that for a graph of constant depth, a maximum independent set can be constructed in polynomial time by dynamic programming techniques. Moreover, for any planar graph $G$ of arbitrary depth, if we remove one layer out of every $K$ consecutive layers, where $K$ is a constant, we obtain a set of separated planar graphs of constant depth. Now for each such graph of constant depth, we construct a maximum independent set. The union of these maximum independent sets forms an independent set for the original graph $G$. For sufficiently large $K$, the number of vertices belonging to the removed layers is very small and thus gives only a small error in the approximation. Baker [10] shows that this method produces a polynomial time approximation scheme for the PLANAR GRAPH INDEP-SET problem with running time bounded by $O(8^{1/\epsilon} n/\epsilon)$.

## 8.3 Optimization for geometric problems

The techniques described in the previous section for approximation of optimization problems on planar graphs are based on the well-known divide and conquer method that has been extensively applied in computer algorithm design. Based on this classical technique, systematic methods have been recently developed in designing polynomial time approximation schemes for a set of famous optimization problems on Euclidean space $\mathcal{E}^d$. Roughly speaking, the new methods work in a dynamic programming manner, which partition the Euclidean space into smaller areas, construct and store good approximations for all possible situations for each smaller area, and recursively construct a good approximation for each situation for a larger area based on the approximations for the smaller areas. The techniques are general for any Euclidean space of a fixed dimension $d$. We will discuss in detail the construction of a polynomial time approximation scheme for the TRAVELING SALESMAN problem on Euclidean plane $\mathcal{E}^2$. Explanation will be briefly provided on how the techniques are extended to other geometric problems and to general Euclidean space $\mathcal{E}^d$ for any $d$.

Each point $p$ in the Euclidean plane $\mathcal{E}^2$ is given by two real numbers that are the $x$- and $y$-coordinates of the point. The Euclidean distance between two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is given by the formula

$$dist(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Let $S$ be a set of $n$ points in $\mathcal{E}^2$, a *traveling salesman tour* on $S$ (or simply called a *salesman tour*) is a closed walk that visits all points in $S$. The EUCLIDEAN TRAVELING SALESMAN problem (or shortly EUCLIDEAN TSP) is to construct a salesman tour of minimum length for a given set of points in $\mathcal{E}^2$.

It is known that EUCLIDEAN TSP is NP-hard in the strong sense [47, 100]. Therefore by Theorem 6.4.8, EUCLIDEAN TSP has no fully polynomial time approximation scheme unless P = NP.

A salesman tour is *polygonal* if it consists of a finite number of line segments. Since the Euclidean distance satisfies the *triangle inequality rule* that the length of the straight line segment connecting two points is not larger than the length of any other path connecting the two points, it is clear that any minimum salesman tour on a set $S$ of $n$ points can be given by a cyclically ordered sequence of the $n$ points that specifies the polygonal salesman tour of $n$ segments on the $n$ points. We will concentrate on polygonal salesman tours. Sometimes in the discussion we may prefer to have the tours "bent" at points that are not in the original set $S$ in order to make the tours satisfy certain special properties. These bends can be easily removed at the end of our approximation: once such a bent polygonal salesman tour $\pi$ is constructed and suppose that it is a good approximation to the minimum salesman tour, we can simply delete points in $\pi$ that do not belong to $S$ to obtain a tour $\pi_c$ that contains only points in $S$. Note that deleting a point $p$ in the tour $\pi = \cdots p_1 p p_2 \cdots$ is equivalent to replacing the path $[p_1, p, p_2]$ of two line segments by a straight line segment $[p_1, p_2]$, which, by the triangle inequality rule, does not increase the length of the tour. Therefore, after deleting the points not in $S$ from the tour $\pi$, we get a polygonal salesman tour $\pi_c$, which only bends at points in the set $S$ and has performance at least as good as the original tour $\pi$.

### 8.3.1 Well-disciplined instances

We first show that when we study approximation algorithms for EUCLIDEAN TSP, we can perform a preprocessing to simplify the instance format and concentrate on only very well-behaved instances of EUCLIDEAN TSP.

**Definition 8.3.1** Let $\epsilon > 0$ be a fixed constant. An instance $S = \{p_1, p_2, \ldots, p_n\}$ of EUCLIDEAN TSP is *$\epsilon$-disciplined* if for each point $p_i = (x_i, y_i)$ in $S$, the coordinates $x_i$ and $y_i$ can be written as $x_i = a_i + 0.5$ and $y_i = b_i + 0.5$, where $a_i$ and $b_i$ are integers, and $0 < x_i, y_i < n/\epsilon$.

A direct consequence from the above definition is that the distance between any two different points in an $\epsilon$-disciplined instance of EUCLIDEAN TSP is at least 1. More importantly, the following lemma shows that it will suffice for us to concentrate on approximation schemes for $\epsilon$-disciplined instances. For a salesman tour $\pi$ of an instance $S$ of EUCLIDEAN TSP, we let $|\pi|$ be the length of the tour $\pi$, and let $Opt(S)$ be the length of a minimum salesman tour in $S$.

**Lemma 8.3.1** *Given any instance $S$ of* EUCLIDEAN TSP, *and any constant $0 < \epsilon < 1$, there is an $\epsilon$-disciplined instance $S_\epsilon$ constructible from $S$ in linear time, such that from any salesman tour $\pi_\epsilon$ for $S_\epsilon$ satisfying $|\pi_\epsilon|/Opt(S_\epsilon) \leq$*

*$1+\epsilon$, we can construct in linear time a salesman tour $\pi$ for $S$ satisfying $|\pi|/Opt(S) \le 1+7\epsilon$.*

PROOF. Let $Q_0$ be the smallest axis-aligned square that contains all the $n$ points in $S$. Since a translation of the Euclidean plane $\mathcal{E}^2$ (i.e., fix $a$ and $b$ and map each point $(x,y)$ in $\mathcal{E}^2$ to the point $(x+a, y+b)$) and a proportional expanding or shrinking of $\mathcal{E}^2$ (i.e., fix a $c$ and map each point $(x,y)$ to the point $(cx, cy)$) do not change the difficulty of approximation solutions to an instance of EUCLIDEAN TSP, we can assume without loss of generality that the lower-left corner of the square $Q_0$ is at the origin point $(0,0)$ and that the side length of the square $Q_0$ is $\lfloor n/\epsilon \rfloor$.

Place an $\lfloor n/\epsilon \rfloor \times \lfloor n/\epsilon \rfloor$ grid on the square $Q_0$ so that each cell in the grid is a $1\times 1$ square whose four corners are of integral coordinates. We construct a new instance $S_\epsilon$ as follows: for each point $p_i$ in $S$, we create a new point $p_i'$ that is at the center of the $1\times 1$ cell containing $p_i$ (if $p_i$ is on the boundary of more than one cell, then pick the center of any of these cells as $p_i'$). Note that the $x$- and $y$-coordinates of each point $p_i' = (x_i', y_i')$ are of the form $x_i' = a_i + 0.5$ and $y_i' = b_i + 0.5$, where $a_i$ and $b_i$ are integers. Moreover, the distance between the point $p_i'$ and the corresponding point $p_i$ in $S$ is bounded by $\sqrt{2}/2$. Let $S_\epsilon = \{p_1', p_2', \ldots, p_n'\}$. It is clear that the set $S_\epsilon$ is an $\epsilon$-disciplined instance of EUCLIDEAN TSP. Note that the $n$ points in the set $S_\epsilon$ may not be all different: a point may have more than one copy in the set $S_\epsilon$. Finally, we observe that the set $S_\epsilon$ can be constructed from the set $S$ in linear time. In fact, it is not necessary to construct the $\lfloor n/\epsilon \rfloor \times \lfloor n/\epsilon \rfloor$ grid: the point $p_i'$ in $S_\epsilon$ can be easily determined from the coordinates of the corresponding point $p_i$ in $S$.

Since $Q_0$ is the smallest square containing $S$, either both horizontal sides or both vertical sides of $Q_0$ contain points in $S$. In particular, there are two points in $S$ whose distance is at least $\lfloor n/\epsilon \rfloor > n/\epsilon - 1$. Therefore, the length of any salesman tour for $S$ is larger than $2n/\epsilon - 2$. Similarly, by the construction of the instance $S_\epsilon$, there are two points in $S_\epsilon$ whose distance is larger than $n/\epsilon - 2$, so the length of any salesman tour for $S_\epsilon$ is larger than $2n/\epsilon - 4$.

Now let $\pi_\epsilon$ be a salesman tour for the $\epsilon$-disciplined instance $S_\epsilon$. We construct a salesman tour $\pi$ for the instance $S$ as follows. We trace the salesman tour $\pi_\epsilon$ and at each point $p_i'$ we make a straight line round-trip from $p_i'$ to the corresponding point $p_i$ in $S$. Note that such a straight line round-trip from $p_i'$ to a corresponding point in $S$ increases the tour length by at most $\sqrt{2}$. Therefore, this process results in a salesman tour $\pi$ for $S$ whose length is bounded by $|\pi_\epsilon| + n\sqrt{2}$ (of course, we can further apply the triangle inequality rule on $\pi$ that may result in a further shorter salesman tour for $S$). In particular, we have shown

$$\frac{|\pi|}{|\pi_\epsilon|} \leq 1 + \frac{n\sqrt{2}}{|\pi_\epsilon|} < 1 + \epsilon \tag{8.2}$$

here we have used the fact $|\pi_\epsilon| > 2n/\epsilon - 4$, $\epsilon < 1$, and assumed $n \geq 8$.

The above method can also be used to estimate the value $Opt(S_\epsilon)$ in terms of $Opt(S)$: starting with an optimal salesman tour of $S$ and adding a straight line round-trip from each point $p_i$ in $S$ to the corresponding point $p'_i$ in $S_\epsilon$ result in a salesman tour of $S_\epsilon$ whose length is bounded by $Opt(S) + n\sqrt{2}$. Thus, the value $Opt(S_\epsilon)$ is bounded by $Opt(S) + n\sqrt{2}$. Combining this with the the lower bound $Opt(S) > 2n/\epsilon - 2$ gives us

$$\frac{Opt(S_\epsilon)}{Opt(S)} \leq 1 + \epsilon \tag{8.3}$$

Now if the salesman tour $\pi_\epsilon$ satisfies $|\pi_\epsilon|/Opt(S_\epsilon) \leq 1 + \epsilon$, then we have

$$\frac{|\pi|}{Opt(S)} = \frac{|\pi|}{|\pi_\epsilon|} \cdot \frac{|\pi_\epsilon|}{Opt(S_\epsilon)} \cdot \frac{Opt(S_\epsilon)}{Opt(S)} \leq (1+\epsilon)^3 \leq 1 + 7\epsilon$$

here we have used inequalities (8.2) and (8.3) and the assumption $\epsilon < 1$. □

### 8.3.2 The approximation scheme for EUCLIDEAN TSP

The polynomial time approximation scheme for the EUCLIDEAN TSP is based on an important Structure Theorem. In this subsection, we first state the Structure Theorem, assume its correctness, and present our algorithm. A proof for the Structure Theorem will be given in the next subsection.

According to Lemma 8.3.1, we only need to concentrate on $\epsilon$-disciplined instances for EUCLIDEAN TSP. Fix $0 < \epsilon < 1$. Let $S = \{p_1, \ldots, p_n\}$ be an $\epsilon$-disciplined instance for EUCLIDEAN TSP. Let $Q_0$ be the bounding square of $S$, where the lower-left corner of $Q_0$ is at the origin $(0,0)$, and each side of $Q_0$ is of length $2^{h_0}$, where $h_0 = \lceil \log(n/\epsilon) \rceil = O(\log n)$.

The bounding square $Q_0$ can be partitioned into four equal size smaller squares by a horizontal segment and a vertical segment. Recursively, suppose $Q$ is a $d \times d$ square that contains more than one point in $S$, then we partition $Q$ into four $(d/2) \times (d/2)$ squares using a horizontal segment and a vertical segment. The partition stops when a square contains no more than one point in $S$. The resulting structure will be called a *(regular) dissection* of the bounding square $Q_0$ (see Figure 8.4(A) for illustration). The squares

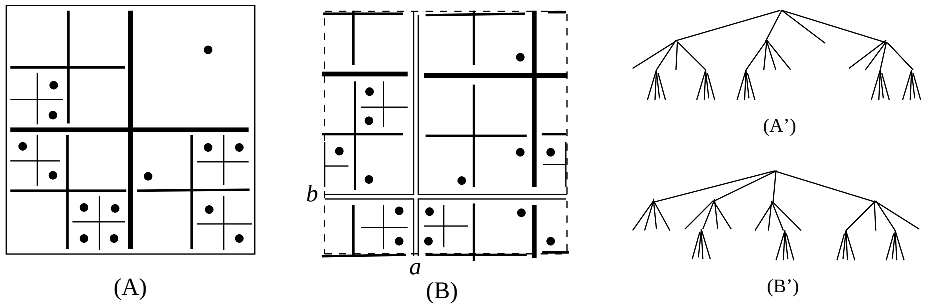



Figure 8.4: (A) a regular dissection; (B) a $(a, b)$-shifted dissection

constructed in the dissection, including those that are further partitioned into smaller squares, will all be called *squares* of the dissection. The sides of the squares will be called *square edges*. Since the instance $S$ is $\epsilon$-disciplined, the edge length of each square in the dissection of $Q_0$ is a positive integer. Moreover, no point in $S$ is on the boundary of any square.

The dissection of $Q_0$ will be represented as a 4-ary tree $T_0$ whose root corresponds to the bounding square $Q_0$. In general, each node $v$ in $T_0$ corresponds to a square $Q_v$ and the four children of $v$ correspond to the four smaller squares resulted from the partition of $Q_v$. Figure 8.4(A') shows the 4-ary tree for the dissection in Figure 8.4(A), where the children of a node are ordered from left to right in terms of the clockwise ordering of the four smaller squares, starting from the lower-left one.

The root of the 4-ary tree $T_0$ will be called the *level-0 node* in $T_0$. In general, a node in $T_0$ is a *level-$i$ node* if its parent is at level $i - 1$. A square corresponding to a level-$i$ node in $T_0$ is called a *level-$i$ square*.

Note that the depth of the tree $T_0$ is bounded by $h_0$. Moreover, since each node in $T_0$ either contains points in $S$ or has a brother containing points in $S$, and two squares at the same level contain no common points in $S$, the number of nodes at each level of $T_0$ is bounded by $O(n)$ (independent of $\epsilon$). In consequence, the total number of nodes in the tree $T_0$ is bounded by $O(h_0 n)$.

Given a square $Q$ and the set $S_Q$ of points in $S$ contained in $Q$, it is rather simple to go through the set $S_Q$ and distribute the points into the four smaller squares resulted from the partition of $Q$. Therefore, each level of the 4-ary tree $T_0$ can be constructed in time $O(n)$. In consequence, the 4-ary tree $T_0$ can be constructed from $S$ in time $O(h_0 n)$.

An important concept is the *shifted dissection structure*. Let $a$ and $b$

be two integers, $0 \leq a, b < 2^{h_0}$. We first identify the two vertical edges of the bounding square $Q_0$ then cut it along the vertical line $x = a$ (see Figure 8.5(A) and (B), which use the same point set $S$ as in Figure 8.4(A)). This is equivalent to cyclically rotating the square $Q_0$ to the left by $a$ units. Then similarly, we identify the two horizontal edges of the resulting square then cut it along the horizontal line $y = b$ (see Figure 8.5(C)). This is equivalent to cyclically rotating the square downwards by $b$ units. Now we put a regular dissection structure on the resulting square (see Figure 8.5(D)). This dissection is call the $(a, b)$*-shifted dissection* of the bounding square $Q_0$. The $(a, b)$-shifted dissection again partitions the bounding square $Q_0$ into "squares", with cuts along lines $x = a$ and $y = b$, and the two vertical edges and the two horizontal edges of $Q_0$ identified.

The $(a, b)$-shifted dissection can also be constructed directly on the original bounding square $Q_0$ with the $x$-coordinate shifted cyclically to the right by $a$ units and the $y$-coordinate shifted cyclically upwards by $b$ units. Regard $Q_0$ as the "square" by identifying the opposite edges of $Q_0$ and cutting $Q_0$ along the vertical line $x = a$ and the horizontal line $y = b$. Now the partition of $Q_0$ into four smaller squares is by the vertical line $x = (a+2^{h_0-1}) \bmod 2^{h_0}$ and the horizontal line $y = (b + 2^{h_0-1}) \bmod 2^{h_0}$. In general, if a square $Q$ is bounded by four lines $x = x_0$, $y = y_0$, $x = (x_0 + 2^i) \bmod 2^{h_0}$ and $y = (y_0 + 2^i) \bmod 2^{h_0}$, then the partition of the square $Q$ into four smaller squares is by the vertical line $x = (x_0 + 2^{i-1}) \bmod 2^{h_0}$ and the horizontal line $y = (y_0 + 2^{i-1}) \bmod 2^{h_0}$. This is illustrated in Figure 8.4(B), where the $(a, b)$-shifted dissection is given on the same bounding square $Q_0$ for the same set $S$ of points as in Figure 8.4(A). Note that the points in the set $S$ are not shifted with the dissection. Readers are advised to convince themselves that the figures in Figure 8.4(B) and Figure 8.5(D) give the same dissection structure.

As for regular dissections, the $(a, b)$-shifted dissection can also be represented by a 4-ary tree of depth $O(h_0)$ and $O(h_0 n)$ nodes, with all related terminologies transferred. Figure 8.4(B') gives the 4-ary tree for the $(a, b)$-shifted dissection in Figure 8.4(B) (again the children of each node are ordered from left to right in terms of the clockwise order of the four smaller squares starting from the lower-left one).

Let $D_{a,b}$ be the $(a, b)$-shifted dissection of the bounding square $Q_0$. Let $e$ be a square edge in the dissection $D_{a,b}$. The $m + 1$ points on $e$ that divide the edge $e$ into $m$ equal length segments will be called the $(1/m)$*-portals* of the square edge $e$.

**Definition 8.3.2** A salesman tour $\pi$ is $(r, m)$*-light* with respect to the $(a, b)$-

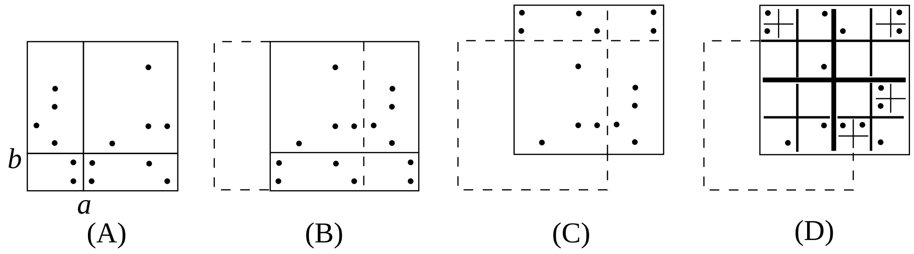



Figure 8.5: A shifted dissection structures

shifted dissection $D_{a,b}$ if for every square edge $e$ of $D_{a,b}$, $\pi$ crosses $e$ at most $r$ times, and each crossing of $\pi$ on $e$ is at a $(1/m)$-portal of $e$.

The polynomial time approximation algorithm for the EUCLIDEAN TSP is heavily based on the following Structure Theorem. We will assume the correctness of the theorem and use it directly in our development of the algorithm. A proof for the Structure Theorem will be given in the next subsection.

**Theorem 8.3.2** (The Structure Theorem) *Let $S$ be an $\epsilon$-disciplined instance for* EUCLIDEAN TSP *and let $Q_0$ be the $2^{h_0} \times 2^{h_0}$ bounding square of $S$, with lower-left corner at the origin $(0,0)$ and $h_0 = \lceil \log(n/\epsilon) \rceil$. Then there is a constant $c_0$ such that for at least half of the pairs $(a,b)$ of integers, $0 \le a, b < 2^{h_0}$, there exists a $(c_0, c_0 h_0)$-light salesman tour $\pi_{a,b}$ with respect to the $(a,b)$-shifted dissection of $Q_0$ satisfying $|\pi_{a,b}| \le (1+\epsilon)Opt(S)$.*

We remark that the constant $c_0$ in Theorem 8.3.2 is independent of the number $n$ of points in the set $S$, but dependent of the given constant $\epsilon$.

Based on Theorem 8.3.2, our algorithm proceeds as follows. For each pair $(a,b)$ of integers, $0 \le a, b < 2^{h_0}$, we apply a dynamic programming algorithm to construct an optimal $(c_0, c_0 h_0)$-light salesman tour $\pi_{a,b}$ with respect to the $(a,b)$-shifted dissection of $Q_0$. According to Theorem 8.3.2, the shortest salesman tour $\pi_o$ among all the $(c_0, c_0 h_0)$-light salesman tours we construct over all $(a,b)$-shifted dissections of $Q_0$, $0 \le a, b \le 2^{h_0}$, will satisfies the condition $|\pi_o| \le (1+\epsilon)Opt(S)$. Note that there are only $2^{h_0} \times 2^{h_0} = O(n^2)$ such pairs $(a,b)$.

For notational simplicity, we let $m_0 = c_0 h_0$.

Consider an $(a,b)$-shifted dissection $D_{a,b}$ of $Q_0$. Let $\pi$ be a $(c_0, m_0)$-light salesman tour with respect to $D_{a,b}$. For each square $Q$ in $D_{a,b}$, the salesman

**Algorithm.** **ETSP**$(S, a, b)$

Input: an $\epsilon$-disciplined instance $S$ and integers $a$ and $b$
Output: an optimal $(c_0, m_0)$-light salesman tour $\pi_{a,b}$ on the $(a, b)$-shifted dissection

1. construct the 4-ary tree $T_{a,b}$ for the $(a, b)$-shifted dissection $D_{a,b}$;
2. **for** each node $v$ in the tree $T_{a,b}$
   {starting from the leaves of $T_{a,b}$ in a bottom-up manner}
   **for** each crossing sequence $\sigma$ of the square $Q_v$ for the node $v$
   **if** $v$ is a leaf
   **then** construct the shortest partial salesman tour in $Q$ consistent with $\sigma$
   **else** { the children of $v$ are 4 smaller squares }
   construct the shortest partial salesman tour in $Q$ consistent with $\sigma$, based on the partial salesman tours constructed for the four smaller squares in $Q_v$.

Figure 8.6: Constructing the $(c_0, m_0)$-light salesman tour for $D_{a,b}$.

tour $\pi$ passes through all points in $S$ contained in $Q$, and the crossings of $\pi$ over the boundaries of $Q$ form a sequence of $(1/m_0)$-portals on the square edges of $Q$ (note that a $(1/m_0)$-portal may appear more than once in the sequence). We will call this sequence a *crossing sequence* of the square $Q$. The line segments of the salesman tour $\pi$ that are contained in the square $Q$ (and pass through all points of $S$ contained in the square $Q$) will be called the *partial salesman tour* (of $\pi$) in the square $Q$. Note that each sequence of even number of $(1/m_0)$-portals on the square edges of $Q$ can be interpreted as a crossing sequence of $Q$ for some salesman tour. We say that a partial salesman tour in a square $Q$ is *consistent* with a crossing sequence $\sigma$ if the partial salesman tour crosses the $(1/m_0)$-portals of the square edges of $Q$ in exactly the same order given in the crossing sequence $\sigma$. Note that there may be more than one partial salesman tour consistent with the same crossing sequence.

Our algorithm works as follows. For each square $Q$ in the dissection $D_{a,b}$, we construct for each possible crossing sequence $\sigma$ of $Q$ the shortest partial salesman tour in $Q$ consistent with $\sigma$. The algorithm runs in a dynamic programming manner, starting from the leaves of the 4-ary tree $T_{a,b}$ for the dissection $D_{a,b}$. The algorithm is given in Figure 8.6.

We give more detailed explanation for the algorithm $\mathbf{ETSP}(S,a,b)$. Suppose the crossing sequence $\sigma$ of the square $Q_v$ corresponding to the node $v$ in the 4-ary tree $T_{a,b}$ is given:

$$\sigma = [I_1, O_1, I_2, O_2, \ldots, I_r, O_r]$$

where $I_i$ and $O_i$ are the $(1/m_0)$-portals for the salesman tour to enter and leave the square $Q_v$, respectively, and $r \leq 4c_0$. In case $v$ is a leaf in the 4-ary tree $T_{a,b}$, the corresponding square $Q_v$ contains at most one point in the set $S$. Therefore, the shortest partial salesman tour in $Q_v$ consistent with the crossing sequence $\sigma$ can be constructed easily: if $Q_v$ contains no point in $S$, then the shortest partial salesman tour in $Q_v$ consistent with $\sigma$ should consist of the $r$ line segments $[I_1, O_1]$, $\ldots$, $[I_r, O_r]$; while if $Q_v$ contains a single point $p$ in $S$, then the shortest partial salesman tour in $Q_v$ should consist of one "bent" line segment $[I_i, p, O_i]$ plus $r-1$ straight line segments $[I_j, O_j]$, $j \neq i$. Since $c_0$ is a constant, the shortest salesman tour in $Q_v$ consistent with the crossing sequence $\sigma$ can be constructed in time $O(1)$.

Now consider the case where the node $v$ is not a leaf. Then the square $Q_v$ is partitioned into four smaller squares $Q_1'$, $Q_2'$, $Q_3'$, and $Q_4'$. Note that there are four edges of the smaller squares that are not on the edges of $Q_v$ but are shared by the smaller squares. We will call these edges the "inner edges" of the smaller squares.

Let $\pi$ be a $(c_0, m_0)$-light salesman tour with the crossing sequence $\sigma$ on the square $Q_v$. If we trace $\pi$ on its crossings on the edges of the smaller squares $Q_1'$, $Q_2'$, $Q_3'$, and $Q_4'$, we obtain a sequence $\tau_0$ of $(1/m_0)$-portals on the edges of the smaller squares. It is easy to see that this sequence $\sigma_0$ can be obtained by merging the crossing sequence $\sigma$ and a sequence $\tau_0'$ of $(1/m_0)$-portals on the inner edges of the smaller squares, with the restriction that at most $c_0$ portal appearances from each inner edge may appear in the sequence $\tau_0'$ (note that a portal on an inner edge may appear more than once in the sequence $\tau_0'$). The four corresponding crossing sequences $\sigma_1'$, $\sigma_2'$, $\sigma_3'$, and $\sigma_4'$, where $\sigma_i'$ is for the smaller square $Q_i'$, $1 \leq i \leq 4$, can be uniquely determined from the sequence $\tau_0$. Moreover, if the partial salesman tour of $\pi$ in the square $Q_v$ is the shortest over all partial salesman tours in $Q_v$ consistent with the crossing sequence $\sigma$, then the partial salesman tour of $\pi$ in each $Q_i'$ of the smaller squares, $1 \leq i \leq 4$, must be the shortest over all partial salesman tours in $Q_i'$ consistent with the crossing sequence $\sigma_i'$.

Therefore, to construct the shortest partial salesman tour in $Q_v$ consistent with the crossing sequence $\sigma$, we examine all sequences $\tau_0$ that can be obtained by merging the crossing sequence $\sigma$ and a sequence $\tau_0'$ of portals on the inner edges of the smaller squares, with the restriction that at most $c_0$ portal appearances from each inner edge may appear in the sequence $\tau_0'$. We consider the complexity of systematically enumerating all these sequences.

Each sequence $\tau_0$ can be obtained as follows. Pick at most $c_0$ portal appearances from each inner edge of the smaller squares. Let $P_0$ be the set of all these portals selected from the inner edges. Now we properly insert each of the portals in $P_0$ in the crossing sequence $\sigma$. Of course, many sequences constructed this way do not give valid crossing sequences on the smaller squares. But this can be checked easily from the sequences themselves.

Since each inner edge $e$ of the smaller squares has $m_0+1$ $(1/m_0)$-portals, there are

$$(m_0+1)^{c_0} + (m_0+1)^{c_0-1} + \cdots + (m_0+1) + 1 \le 2(m_0+1)^{c_0}$$

ways to pick at most $c_0$ portal appearances from $e$. Therefore, totally there are at most $2^4(m_0+1)^{4c_0}$ ways to construct a set $P_0$ of portal appearances, in which each inner edge has at most $c_0$ portal appearances. Once the set $P_0$ is decided, the number of ways to insert the portal appearances in $P_0$ into the crossing sequence $\sigma$ is bounded by (note that both the set $P_0$ and the crossing sequence $\sigma$ have at most $4c_0$ portal appearances):

$$(4c_0+1)(4c_0+2)\cdots(4c_0+4c_0) = \frac{(8c_0)!}{(4c_0)!}$$

Therefore, the number of sequences $\tau_0$ that may represent valid crossing sequences for the four smaller squares consistent with the crossing sequence $\sigma$ of $Q_v$ is bounded by

$$2^4(m_0+1)^{4c_0} \cdot \frac{(8c_0)!}{(4c_0)!} = O((\log n)^{O(1)})$$

and these sequences can be enumerated systematically in time $O((\log n)^{O(1)})$ (note that each sequence is of length $O(c_0) = O(1)$ so operations on each sequence can always be done in constant time).

By our algorithm, the shortest partial salesman tour consistent with each crossing sequence for each smaller square has been constructed and stored. Therefore, for each valid set $\{\sigma_1', \sigma_2', \sigma_3', \sigma_4'\}$ of crossing sequences for the smaller squares $Q_1'$, $Q_2'$, $Q_3'$, and $Q_4'$, consistent with the crossing sequence $\sigma$ of $Q_v$, we can easily construct the corresponding partial salesman tour consistent with $\sigma$ in the square $Q_v$. Examining all valid sets of crossing sequences for the smaller squares will result in the shortest partial salesman tour consistent with the crossing sequence $\sigma$ in the square $Q_v$.

Summarizing the above discussion, we conclude that for each node $v$ in the 4-ary tree $T_{a,b}$ and for each crossing sequence $\sigma$ of the square $Q_v$ for $v$, the shortest partial salesman tour in $Q_v$ consistent with $\sigma$ can be constructed in time $O((\log n)^{O(1)})$.

Now we count the number of different crossing sequences for a given square $Q_v$. For each edge $e$ of $Q_v$, a crossing sequence may cross the portals of $e$ at most $c_0$ times. There are at most $2(m_0+1)^{c_0}$ ways to pick no more than $c_0$ portal appearances on the edge $e$, and there are totally at most $2^4(m_0+1)^{4c_0}$ ways to pick no more than $c_0$ portal appearances from each of the four edges of $Q_v$. For each such selection of portal appearances, a permutation of these selected portal appearances gives a crossing sequence of $Q_v$. Since each such selection contains at most $4c_0$ portal appearances, the total number of possible crossing sequences of $Q_v$ is bounded by

$$2^4(m_0+1)^{4c_0}(4c_0)! = O((\log n)^{O(1)})$$

Therefore, the algorithm $\mathbf{ETSP}(S, a, b)$ spends at most time

$$O((\log n)^{O(1)}) \cdot O((\log n)^{O(1)}) = O((\log n)^{O(1)})$$

on each node in the 4-ary tree $T_{a,b}$. Since the number of nodes in the 4-ary tree $T_{a,b}$ is bounded by $O(n \log n)$, we conclude that to construct an optimal $(c_0, m_0)$-light salesman tour for the dissection $D_{a,b}$ takes time $O(n(\log n)^{O(1)})$.

**Remark.** There are also some "obvious" restrictions we should keep in mind when we are constructing crossing sequences for the squares in a dissection. For example, suppose that a square edge $e$ is on the boundary of the original bounding square $Q_0$, then no portals of $e$ should be picked in any crossing sequence of the square since an optimal $(c_0, m_0)$-light salesman tour will definitely not cross the edge $e$. Moreover, in the crossing sequences of the level-0 square $Q$ for the $(a,b)$-shifted dissection, if a portal in an "out-portal" on a edge of $Q$, then the same position on the opposite edge of $Q$ should be an "in-portal" since the opposite edges of $Q$ are actually the same line in the original bounding square $Q_0$. These obvious restrictions can be all easily checked.

**Theorem 8.3.3** *For any fixed $\epsilon > 0$, there is an $O(n^3(\log n)^{O(1)})$ time approximation algorithm for the* EUCLIDEAN TSP *problem that for any instance $S$ constructs a salesman tour $\pi$ satisfying $|\pi|/Opt(S) \leq 1+\epsilon$.*

PROOF. Let $\delta = \epsilon/7$. According to Lemma 8.3.1, we can construct a

$\delta$-disciplined instance $S_\delta$ in linear time such that for any salesman tour $\pi_\delta$ for $S_\delta$ satisfying $|\pi_\delta|/Opt(S_\delta) \le 1 + \delta$, we can construct in linear time a salesman tour $\pi$ for $S$ satisfying $|\pi|/Opt(S) \le 1 + 7\delta = 1 + \epsilon$.

The theorem is concluded since according to the above analysis: for each $(a, b)$-shifted dissection $D_{a,b}$, we can construct in time $O(n(\log n)^{O(1)})$ the optimal $(c_0, c_0 h_0)$-light salesman tour with respect to $D_{a,b}$. According to Theorem 8.3.2, the salesman tour $\pi_\delta$ that is the shortest over all $(c_0, c_0 h_0)$-light salesman tours constructed for all shifted dissections must satisfy $|\pi_\delta|/Opt(S_\delta) \le 1 + \delta$. Moreover, the total number of shifted dissections is bounded by $n^2$. $\square$

We remark that in the time complexity $O(n^3(\log n)^{O(1)})$ of the algorithm in Theorem 8.3.3, both the constant coefficient and the constant exponent of the logorithmic function depend on the given $\epsilon$. In particular, the algorithm is not a fully polynomial time approximation scheme.

The time complexity of the algorithm in Theorem 8.3.3 can be improved if we are allowed to use randomization in our computation. According to Theorem 8.3.2, for at least half of the pairs $(a, b)$, the optimal $(c_0, c_0 h_0)$-light salesman tour $\pi_{a,b}$ with respect to the $(a, b)$-shifted dissection $D_{a,b}$ satisfies $|\pi_{a,b}| \le (1 + \epsilon)Opt(S)$. Therefore, if we randomly pick, say, 10 pairs of $(a, b)$ and construct the optimal $(c_0, c_0 h_0)$-light salesman tour for each of the corresponding shifted dissections, then the probability that the shortest $\pi_{a,b}$ of these ten $(c_0, c_0 h_0)$-light salesman tours satisfies the condition $|\pi_{a,b}| \le (1 + \epsilon)Opt(S)$ is as large as $1 - 1/2^{10} > 0.999$. Therefore, using randomization, the time consuming enumeration of all the $O(n^2)$ pairs of $(a, b)$ can be avoided. This gives us the following theorem.

**Theorem 8.3.4** *For any fixed $\epsilon > 0$ and any fixed $\delta > 0$, there is an $O(n(\log n)^{O(1)})$ time randomized approximation algorithm for the* EUCLIDEAN TSP *problem that for any instance $S$ constructs a salesman tour $\pi$ satisfying $|\pi|/Opt(S) \le 1 + \epsilon$ with probability at least $1 - \delta$.*

### 8.3.3 Proof for the Structure Theorem

For completeness, we present a detailed proof for the Structure Theorem (Theorem 8.3.2) in this subsection. Readers may skip this subsection on their first reading. This will not affect continuous understanding of the rest of the book.

Let $S = \{p_1, \ldots, p_n\}$ be an $\epsilon$-disciplined instance for EUCLIDEAN TSP. Let $Q_0$ be the bounding square of $S$, where the lower-left corner of $Q_0$ is at the origin $(0,0)$, and each side of $Q_0$ is of length $2^{h_0}$, where $h_0 = \lceil \log(n/\epsilon) \rceil = O(\log n)$. For each pair $(a,b)$ of integers, $0 \le a, b < 2^{h_0}$, denote by $D_{a,b}$ the $(a,b)$-shifted dissection of $Q_0$.

The Structure Theorem claims that for at least half of the pairs $(a,b)$ of integers, $0 \le a, b < 2^{h_0}$, there is a $(c_0, c_0 h_0)$-light salesman tour $\pi_{a,b}$ with respect to $D_{a,b}$ satisfying $|\pi_{a,b}| \le (1+\epsilon)Opt(S)$, where $c_0$ is a constant.

To prove the Structure Theorem, we start with a shortest salesman tour $\pi_o$ for the instance $S$ and show how the salesman tour $\pi_o$ can be modified into a $(c_0, c_0 h_0)$-light salesman tour $\pi_{a,b}$ without much increase in tour length. For this, we need to show that how the shortest salesman tour $\pi_o$ is modified so that the number of crossings at each square edge of $D_{a,b}$ is bounded by the constant $c_0$ and that all crossings occur only at the $(1/(c_0 h_0))$-portals of the square edge.

Intuitively, the total number of crossings of the shortest salesman tour $\pi_o$ over the square edges in $D_{a,b}$ should not be very large since a large number of crossings over a line segment should be very costful. However, it is still possible that the number of crossings of $\pi_o$ over a particular square edge exceeds the constant $c_0$. Therefore, we first need to discuss how to reduce the number of crossings of a salesman tour over a particular square edge. The second requirement, that crossings only occur at portals of the square edges, is relatively easier to achieve since moving a crossing to its nearest portal is not very expensive. In the following, we formally implement these ideas.

First we consider the number of crossings over a particular square edge. Note by a "crossing" we mean the salesman tour hits the square edge from one side of the edge then, maybe after some "zigzag" moves along the edge, leaves the edge to the other side. In particular, it will not count as a crossing if the tour hits the edge then leaves the edge back to the same side. The reduction of the number of crossings on a square edge is based on the following lemma, which we will call the "Patching Lemma".

**Lemma 8.3.5** (Patching Lemma) *Let $s$ be a line segment of length $|s|$ and $\pi$ be a salesman tour for an instance $S$ of* EUCLIDEAN TSP. *Then there is a salesman tour $\pi'$ for $S$ such that $|\pi'| \le |\pi| + 3|s|$ and $\pi'$ crosses $s$ at most twice.*

PROOF. Without loss of generality, assume that $s$ is a vertical line.

Let $p_1, p_2, \ldots, p_t$ be the points on $s$ at which $\pi$ crosses, sorted by their $y$-coordinates in nonincreasing order (see Figure 8.7(A)), with $t \ge 3$. Here we do not exclude the possibility that some of these points are identical.

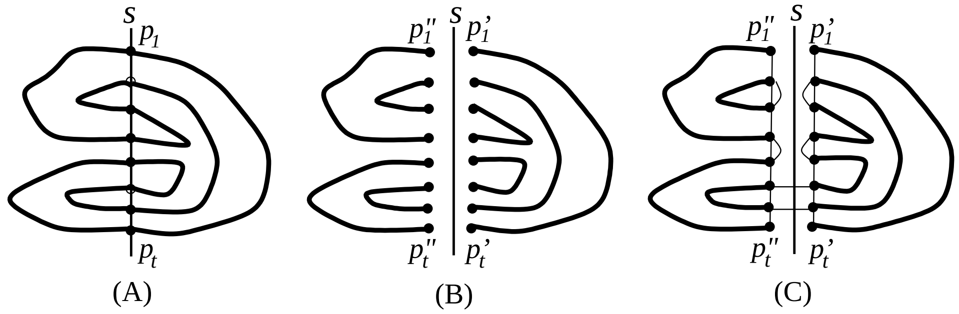



Figure 8.7: Reducing the number of crossings by patching

Duplicate each point $p_i$ into two copies $p_i'$ and $p_i''$ and imagine that the points $p_1'$, ..., $p_t'$ are connected to the "right part" of the tour $\pi$ while the points $p_1''$, ..., $p_t''$ are connected to the "left part" of the tour $\pi$ (see Figure 8.7(B)). Now add edges

$$(p_1', p_2'), (p_2', p_3'), \ldots, (p_{t-1}', p_t'), (p_1'', p_2''), (p_2'', p_3''), \ldots, (p_{t-1}'', p_t'') \tag{8.4}$$

The total length of the added edges in (8.4) is bounded by $2|s|$ (the length of an edge is defined to be the Euclidean distance between its two endpoints). In the resulting graph, the vertices $p_1'$, $p_1''$, $p_t'$, and $p_t''$ have degree 2, and all other vertices have degree 3. Based on the parity of $t$, we add another set of multiple edges to the graph.

In case $t$ is even, we add to the graph one of the following two sets of multiple edges, with smaller total edge length

$$(p_2', p_3'), (p_4', p_5'), \ldots, (p_{t-4}', p_{t-3}'), (p_2'', p_3''), (p_4'', p_5''), \ldots, (p_{t-4}'', p_{t-3}''),$$
$$\text{and } (p_{t-2}', p_{t-2}''), (p_{t-1}', p_{t-1}'')$$

or (8.5)

$$(p_3', p_4'), (p_5', p_6'), \ldots, (p_{t-3}', p_{t-2}'), (p_3'', p_4''), (p_5'', p_6''), \ldots, (p_{t-3}'', p_{t-2}''),$$
$$\text{and } (p_2', p_2''), (p_{t-1}', p_{t-1}'')$$

In case $t$ is odd, we add to the graph one of the following two sets of multiple edges, with smaller total edge length

$$(p_2', p_3'), (p_4', p_5'), \ldots, (p_{t-3}', p_{t-2}'), (p_2'', p_3''), (p_4'', p_5''), \ldots, (p_{t-3}'', p_{t-2}''),$$
$$\text{and } (p_{t-1}', p_{t-1}'')$$

or (8.6)

$$(p_3', p_4'), (p_5', p_6'), \ldots, (p_{t-2}', p_{t-1}'), (p_3'', p_4''), (p_5'', p_6''), \ldots, (p_{t-2}'', p_{t-1}''),$$
$$\text{and} \;\; (p_2', p_2'')$$

(see Figure 8.7(C) for illustration, where the thinner edges are the added edges). The idea here is that the total length of the added multiple edges in (8.5) or (8.6) is bounded by $|s|$ (note that the length of the edges $(p_i', p_i'')$ is 0). Moreover, the salesman tour $\pi$ and all the added edges form a graph $G$ in which every vertex has an even degree. The sum of the edge lengths of the graph $G$ is bounded by $|\pi| + 3|s|$.

It is well-known in graph theory (see Appendix A, Theorem **??**) that in a graph in which all vertices have even degree, there is a *Eulerian tour* (i.e., a tour that passes each edge in the graph exactly once). Therefore, the Eulerian tour $\pi'$ in the graph $G$ forms a salesman tour for the instance $S$, which crosses the line segment $s$ at most twice and has length bounded by $|\pi| + 3|s|$. $\square$

Now we are ready to prove the Structure Theorem. Let $c_0 = \lceil 64/\epsilon + 3 \rceil$.

Put a $2^{h_0} \times 2^{h_0}$ uniform grid structure on the bounding square $Q_0$ by $2^{h_0}$ equally spaced vertical lines and $2^{h_0}$ equally spaced horizontal lines (note we identify the opposite sides of the square $Q_0$). These lines will be called *grid lines*. Note that every square edge in the dissection $D_{a,b}$ is on a grid line. Recall that a square is a *level-$i$ square* if its corresponding node is at level $i$ in the 4-ary tree $T_{a,b}$ for the dissection $D_{a,b}$. A level-$i$ square is a $2^{h_0-i} \times 2^{h_0-i}$ square, and the maximum level number is $h_0$. The square edges of a level-$i$ square will be called *level-$i$ square edges*. We say that a grid line $\ell$ is *at level-$i$* if $i$ is the smallest integer such that a level-$i$ square edge is on $\ell$. Note that a level-$i$ grid line may also contain square edges of level number larger than $i$.

Let $\pi_o$ be a polygonal salesman tour for the instance $S$ and $|\pi_o| = Opt(S)$. Let $\ell$ be a level-$i$ grid line (either vertical or horizontal). We discuss how to reduce the number of crossings of $\pi_o$ on the square edges of $D_{a,b}$ on $\ell$, using the Patching Lemma. The simplest way is to apply the Patching Lemma to each level-$i$ square edge on $\ell$ so that the number of crossings on each of these edges is bounded by 2. Note that this also automatically ensures that the number of crossings on each square edge of level larger than $i$ on $\ell$ is also bounded by 2, since each level-$j$ square edge on $\ell$, where $j > i$, must be entirely contained in a level-$i$ square edge on $\ell$. Unfortunately, this simple method may be expensive due to the following observation: suppose that more than $c_0$ crossings occur on a level-$j$ square edge $e_j$ on $\ell$, where $j > i$, then, to replace these crossings by at most 2 crossings, applying the Patching

**Algorithm. Modify**($\ell$)
{ Assume that $\ell$ is a level-$i$ grid line }.

**for** $j = h_0$ **downto** $i$ **do**
  **for** each level-$j$ square edge $e$ on $\ell$ **do**
    **if** $\pi$ crosses $e$ more than $c_0$ times
    **then** apply Patching Lemma to $\pi$ and $e$.

Figure 8.8: Reducing the number of crossings on a grid line

Lemma directly to the level-$i$ square edge $e_i$ containing $e_j$ would possibly increase the tour length by $3|e_i|$, while applying the Patching Lemma to the square edge $e_j$ has tour length increase bounded by $3|e_j|$. The edge length $|e_i|$ can be much larger than the edge length $|e_j|$.

Based on this observation, we apply the Patching Lemma in a "bottom up" manner starting from the shortest square edges, i.e., the square edges of the highest level number, on the grid line $\ell$. The patching procedure, which is called **Modify**($\ell$), is given in Figure 8.8.

To analyze the algorithm, we introduce two new notations. Let $\omega(\pi_o, \ell)$ be the total number of crossings of the shortest salesman tour $\pi_o$ on the grid line $\ell$, and let $\rho(\ell, j)$ be the number of times the algorithm **Modify**($\ell$) applies the Patching Lemma to a level-$j$ square edge on the grid line $\ell$.

Here we have to be a little more careful about the square edges in the shifted dissection $D_{a,b}$. Recall that a "square" in $D_{a,b}$ may be formed by several non-connected pieces in the original bounding square $Q_0$ (see Figure 8.4(B)). If a square edge $e$ crosses a boundary side of $Q_0$, then the square edge is actually formed by two non-connected segments $e'$ and $e''$ in $Q_0$. Therefore, in case there are more than $c_0$ crossings of $\pi_o$ over $e$, the Patching lemma should be applied to the two segments $e'$ and $e''$ separately since formally applying the Patching Lemma to the square edge $e$ would introduce a partial tour that crosses a boundary side of $Q_0$ and continues on the opposite side of $Q_0$. The two separated applications of the Patching Lemma on $e'$ and $e''$ may leave up to 4 crossings (instead of 2) on the square edge $e$ in $D_{a,b}$. Therefore, we can ensure that each application of the Patching Lemma to a square edge in $D_{a,b}$ replaces a set of more than $c_0$ crossings by a set of at most 4 crossings, thus reducing the number of crossings by at least $c_0 - 3$. This observation gives the following relation for the values

$\rho(\ell, j)$ and $\omega(\pi_o, \ell)$:

$$\sum_{j=i}^{h_0} \rho(\ell, j) = \sum_{j=0}^{h_0} \rho(\ell, j) \leq \frac{\omega(\pi_o, \ell)}{c_0 - 3} \tag{8.7}$$

The first equality in (8.7) is because this relation is independent of the level number of the grid line $\ell$.

Since the length of a level-$j$ square edge is $2^{h_0 - j}$, each application of the Patching Lemma to a level-$j$ square edge increases the tour length by at most $3 \cdot 2^{h_0 - j}$. Therefore, the total increase in tour length by the algorithm **Modify**($\ell$) is bounded by

$$\sum_{j=i}^{h_0} 3 \cdot 2^{h_0 - j} \rho(\ell, j) = 3 \sum_{j=i}^{h_0} \cdot 2^{h_0 - j} \rho(\ell, j) \tag{8.8}$$

The algorithm **Modify**($\ell$) modifies the salesman tour and ensures that the number of crossings over each square edge on $\ell$ is bounded by $c_0$. However, here is a potential problem we need to take care of. Without loss of generality, suppose that $\ell$ is a vertical grid line. Patching on $\ell$ may introduce many "zigzag" moves along the line $\ell$, which may cause new crossings over horizontal grid lines. Let $\ell'$ be such a horizontal grid line and let $\omega'$ be the set of new crossings over $\ell'$ caused by patching the grid line $\ell$. Note that all these new crossings over $\ell'$ are along the line $\ell$ so the segment $s'$ on $\ell'$ holding these crossings has length 0. Therefore, applying the Patching Lemma to $s'$ and $\omega'$ will reduce the number of crossings to at most 2 *without increasing the tour length*! In order to avoid introducing new crossings on the grid line $\ell$ by the patching on $s'$ and $\omega'$, we can actually apply the Patching Lemma twice, first to the segment $s'$ and the crossings in $\omega'$ that occur on the left side of $\ell$, then to the segment $s'$ and the crossings in $\omega'$ that occur on the right side of $\ell$. This will reduce the number of crossings on the segment $s'$ to at most 4, without increasing the tour length and the number of crossings over the grid line $\ell$.

After the application of the algorithm **Modify**($\ell$), each square edge on the grid line $\ell$ contains at most $c_0$ crossings. Now we move each crossing point $p$ to its nearest $(1/(c_0 h_0))$-portal on the level-$i$ square edge $e_i$ on $\ell$ in an obvious way: instead of crossing at $p$, we let the salesman tour first go along the grid line $\ell$ (without crossing $\ell$) to the nearest $(1/(c_0 h_0))$-portal $p'$ of $e_i$, then cross $e_i$ at the portal $p'$ and go along $\ell$ (now on the other side of $\ell$) to come back to the old crossing point $p$ and continue the tour. Note that this will also move the crossing to a $(1/(c_0 h_0))$-portal on the square edge containing $p'$ at any level on $\ell$ since a $(1/(c_0 h_0))$-portal on a level-$i$ square edge $e_i$ is also a $(1/(c_0 h_0))$-portal on any level-$j$ square edge contained in $e_i$, where $j \geq i$. Since the distance between two neighbor $(1/(c_0 h_0))$-portals on a level-$i$ square edge is $2^{h_0 - i}/(c_0 h_0)$, the above modification on the tour increases the tour length by at most $2^{h_0-i}/(c_0 h_0)$. Since there are no more than $\omega(\pi_o, \ell)$ crossings over the grid line $\ell$, the total increase in tour length to move the crossings to portals is bounded by $2^{h_0-i}\omega(\pi_o, \ell)/(c_0 h_0)$. Combining this with (8.8), we conclude that we can modify the salesman tour $\pi_o$ so that the number of crossings on each square edge on the grid line $\ell$ is bounded by $c_0$ and all crossings are only at $(1/(c_0 h_0))$-portals of the square edges, with the tour length increase $\tau(\ell, i)$ bounded by

$$\tau(\ell, i) = 3\sum_{j=i}^{h_0} \cdot 2^{h_0-j}\rho(\ell, j) + \frac{2^{h_0-i}\omega(\pi_o, \ell)}{c_0 h_0} \tag{8.9}$$

Now instead of computing directly the tour length increase due to the above modification, we use a probabilistic argument. Look at a given $(a, b)$-shifted dissection $D_{a,b}$. With respect to the dissection $D_{a,b}$, there is 1 vertical grid line and 1 horizontal grid line of level 0, and for $i > 0$, there are $2^{i-1}$ vertical grid lines and $2^{i-1}$ horizontal grid lines of level $i$. Therefore, if the integers $a$ and $b$ are picked randomly (with a uniform distribution) from the set $\{0, 1, \cdots, 2^{h_0} - 1\}$, then for a fixed grid line $\ell$ (either vertical or horizontal), the probability that $\ell$ is a level-0 grid line is $1/2^{h_0}$, and for $i > 0$, the probability that $\ell$ is a level-$i$ grid line is $2^{i-1}/2^{h_0}$. Therefore, the expected value of the tour length increase on the grid line $\ell$ is bounded by

$$\begin{aligned}
& \sum_{i=0}^{h_0} \tau(\ell, i) \cdot \text{Prob}[\ell \text{ is a level-}i \text{ grid line}] \\
= \ & \frac{1}{2^{h_0}}\tau(\ell, 0) + \sum_{i=1}^{h_0} \frac{2^{i-1}}{2^{h_0}}\tau(\ell, i) \\
\leq \ & \sum_{i=0}^{h_0} \frac{2^i}{2^{h_0}}\tau(\ell, i) \\
= \ & \sum_{i=0}^{h_0} \frac{2^i}{2^{h_0}} \left( 3\sum_{j=i}^{h_0} \cdot 2^{h_0-j}\rho(\ell, j) + \frac{2^{h_0-i}\omega(\pi_o, \ell)}{c_0 h_0} \right) \\
= \ & 3\sum_{i=0}^{h_0}\sum_{j=i}^{h_0} 2^{i-j}\rho(\ell, j) + \sum_{i=0}^{h_0} \frac{\omega(\pi_o, \ell)}{c_0 h_0}
\end{aligned}$$

$$\begin{aligned}
&= 3\sum_{j=0}^{h_0}\left(\sum_{i=0}^{j}\frac{1}{2^i}\right)\rho(\ell,j) + \frac{2\omega(\pi_o,\ell)}{c_0} \\
&\le 6\sum_{j=0}^{h_0}\rho(\ell,j) + \frac{2\omega(\pi_o,\ell)}{c_0} \\
&\le \frac{6\omega(\pi_o,\ell)}{c_0-3} + \frac{2\omega(\pi_o,\ell)}{c_0} \\
&\le \frac{8\omega(\pi_o,\ell)}{c_0-3}
\end{aligned}$$

Here we have used the inequality (8.7).

Thus, if the integers $a$ and $b$ are picked randomly, then the expected value of the total tour length increase to modify the shortest salesman tour $\pi_o$ into a $(c_0, c_0h_0)$-light salesman tour with respect to $D_{a,b}$ is bounded by

$$\sum_{\text{grid line } \ell}\frac{8\omega(\pi_o,\ell)}{c_0-3} = \frac{8}{c_0-3}\sum_{\text{grid line } \ell}\omega(\pi_o,\ell) = \frac{\epsilon}{8}\sum_{\text{grid line } \ell}\omega(\pi_o,\ell) \tag{8.10}$$

To complete the proof, we show

$$\sum_{\text{grid line } \ell}\omega(\pi_o,\ell) \le 4|\pi_o| = 4\cdot Opt(S)$$

Recall that $\pi_o$ is a polygonal salesman tour consisting of $n$ segments connecting the points in $S$. For each segment $s$ in $\pi_o$ with length $|s|>0$, let $|x_s|$ and $|y_s|$ be the length of the horizontal and vertical projections of $s$. Then the segment $s$ can cross at most $|x_s|+1$ vertical grid lines and at most $|y_s|+1$ horizontal grid lines. We have

$$|x_s|+|y_s|+2 \le \sqrt{2(|x_s|^2+|y_s|^2)}+2 = |s|\sqrt{2}+2$$

where we have used the facts that the length $|s|$ of the segment $s$ satisfies $|s|^2 = |x_s|^2+|y_s|^2$ and $|x_s|^2+|y_s|^2 \ge 2|x_s|\cdot|y_s|$. Therefore, the total number of crossings of the salesman tour $\pi_o$ over all grid lines can be bounded by

$$\begin{aligned}
\sum_{\text{grid line } \ell}\omega(\pi_o,\ell) &\le \sum_{\text{segment } s}(|x_s|+|y_s|+2) \le \sum_{\text{segment } s}(|s|\sqrt{2}+2) \\
&\le \sum_{\text{segment } s}(|s|\sqrt{2}+2|s|) \le 4\sum_{\text{segment } s}|s| \le 4|\pi_o| = 4\cdot Opt(S)
\end{aligned}$$

Note here we have used the fact $|s|\ge 1$, which is true because the instance $S$ is $\epsilon$-disciplined. Combining this with (8.10), we conclude that for random integers $a$ and $b$, $0 \leq a, b < 2^{h_0}$, the expected value of the total tour length increase to modify the shortest salesman tour $\pi_o$ into a $(c_0, c_0 h_0)$-light salesman tour, with respect to $D_{a,b}$, is bounded by $\epsilon \cdot Opt(S)/2$. This implies immediately that for at least half of the pairs $(a, b)$, the total tour length increase to modify the shortest salesman tour $\pi_o$ into a $(c_0, c_0 h_0)$-light salesman tour $\pi_{a,b}$, with respect to the $(a, b)$-shifted dissection $D_{a,b}$, is bounded by $\epsilon \cdot Opt(S)$. That is, the $(c_0, c_0 h_0)$-light salesman tour $\pi_{a,b}$ with respect to $D_{a,b}$ satisfies $|\pi_{a,b}| \leq (1 + \epsilon) Opt(S)$.

This completes the proof for the Structure Theorem.

**Remark.** The probabilistic argument used above is not necessary. In fact, direct counting, using the idea adopted in the probabilistic argment, would also derive the same result. For this, we first count the number of level-$i$ grid lines with respect to each dissection $D_{a,b}$, then compute the tour length increase on this particular dissection $D_{a,b}$. Finally, we add the tour length increases over all dissections $D_{a,b}$, and will find out that the "average" tour length increase on each dissection is bounded by $\epsilon \cdot Opt(S)/2$. Now the same conclusion should be derived.

### 8.3.4 Generalization to other geometric problems

A number of important techniques have been described in the discussion of our polynomial time approximation scheme for the EUCLIDEAN TSP. The concepts of $\epsilon$-disciplined instances and $(c_0, m_0)$-light salesman tours enable us to concentrate on very well-behaved instances and solutions. The Patching Lemma introduces an effective method to convert a solution to a well-behaved solution, and the Structure Theorem makes it possible to apply dynamic programming to search for an optimal well-behaved solution efficiently. This systematic echnique turns out to be very effective and powerful in development of approximation algorithms for geometric problems. In the following, we briefly describe the extensions of this technique to solve other geometric problems.

The extension of Theorem 8.3.3 to EUCLIDEAN TSP in higher dimensional Euclidean space $\mathcal{E}^d$ is natural, when $d$ is a fixed constant. As before, we first make an instance $S$ $\epsilon$-disciplined by rescaling and perturbation, as we did in Lemma 8.3.1. Now the Patching Lemma is applied to a $(d-1)$-dimensional hypercube (instead of to a line segment as we did in Lemma 8.3.5) to reduce the number of crossings to the $(d-1)$-dimensional hypercube to at most 2. A similar Structure Theorem can be proved based on these modifications for EUCLIDEAN TSP in $\mathcal{E}^d$ which again makes the dynamic programming possible to search for a well-behaved salesman tour for $S$ efficiently. We omit all details and refer the readers to Arora's original paper [5]. Here we only state the final result for this extension.

**Theorem 8.3.6** *For any fixed $\epsilon > 0$ and any fixed integer $d$, there is a polynomial time approximation algorithm for the* EUCLIDEAN TSP *problem in the $d$-dimensional Euclidean space $\mathcal{E}^d$ that for any instance $S$ constructs a salesman tour $\pi$ satisfying $|\pi|/Opt(S) \leq 1 + \epsilon$.*

The technique can also be applied to develop polynomial time approximation schemes for the geometric problems listed below, where $d$ is a fixed integer. For each of these problems, we need to modify our concepts of the $\epsilon$-disciplined instances, the well-behaved solutions, the Patching Lemma, and the Structure Theorem accordingly. We also refer our readers to the original paper [5] for details.

EUCLIDEAN STEINER TREE: Given a set $S$ of $n$ points in the Euclidean space $\mathcal{E}^d$, find a minimum cost tree connecting all points in $S$ (the tree does not have to use only the given points in $S$ as its nodes).

PARTIAL TSP: Given a set $S$ of $n$ points in the Euclidean space $\mathcal{E}^d$ and an integer $k > 1$, find the shortest tour that visits at least $k$ points in $S$.

PARTIAL MST: Given a set $S$ of $n$ points in the Euclidean space $\mathcal{E}^d$ and an integer $k > 2$, find $k$ points in $S$ such that the minimum spanning tree on the $k$ points is the shortest (over minimum spanning trees on all subsets of $k$ points in $S$).

**Theorem 8.3.7** *Each of the following geometric problems has a polynomial time approximation scheme:* EUCLIDEAN STEINER TREE, PARTIAL TSP, *and* PARTIAL MST.

## 8.4 Which problems have no PTAS?

Polynomial time approximation schemes offer an efficient method to construct solutions very close to the optimal solutions for optimization problems whose optimal solutions otherwise would be hard to construct. Therefore, it is desirable to know whether a given NP-hard optimization problem has a polynomial time approximation scheme. In Section 6.4, we have developed effective and powerful methods (Theorem 6.4.1 and Theorem 6.4.8) to identify NP-hard optimization problems that have no *fully* polynomial time approximation scheme. One would expect that a similar approach could offer equally effective and powerful methods for identifying NP-hard optimization problems with no (non-fully) polynomial time approximation scheme. However, the solution to this task turns out to require deeper understanding of the complexity of NP-hard optimization problems.

It is interesting and enlightening to have a brief historical review on the development of polynomial time approximation schemes for certain NP-hard optimization problems.

Consider the MAKESPAN problem (see Section 8.1 for a precise definition for the problem). Graham initialized the study of approximation algorithms for this important optimization problem in 1966 [56] and showed that there is a polynomial time approximation algorithm for the problem with approximation ratio 2 (see Algorithm **Graham-Schedule** and Theorem 5.3.1). The algorithm is based on a very simple greedy method that assigns each job to the earliest available processor. Three years later he further showed that if a preprocessing is performed that sorts the jobs by their processings times in non-increasing order before the algorithm **Graham-Schedule** is applied, then the approximation ratio of the algorithm can be improved to 4/3 (see Theorem 5.3.3). The approximation ratio for the MAKESPAN problem then was continuously improved. In 1978, it was improved to 1.22, then to 1.20 and then to $72/61 = 1.18\cdots$ (see the introduction section in [63] for more detailed review and references of this line of research). This line of research was eventually closed by Hochbaum and Shmoys' polynomial time approximation scheme for the problem [63], which concludes that for any $\epsilon > 0$, there is a polynomial time approximation algorithm for the MAKESPAN problem with approximation ratio bounded by $1 + \epsilon$. This approximation scheme has been described in detail in Section 8.1.

Another similar story has happened for EUCLIDEAN TSP (see Section 8.3 for a precise definition for the problem). A very neat approximation algorithm based on minimum spanning trees for the problem has an approximation ratio 2. Christofides' remarkable work, based on the approach of minimum spanning trees incorporated with observations in minimum weight complete matchings and Euler tours, improved this ratio to 1.5. Christofides' ratio for EUCLIDEAN TSP stood as the best result for two decades (see the introduction section in [5] for more detailed review and references for this line of research, also see Section 9.1 for detailed descriptions for these algorithms). In fact, the difficulty for improving Christofides' ratio had made people attempt to believe that EUCLIDEAN TSP has no polynomial time approximation scheme. A surprising breakthrough by Arora [5] was made twenty year after Christofides's algorithm, which presented a polynomial time approximation scheme for EUCLIDEAN TSP, as we described in Section 8.3. It is also interesting to point out that Arora's result was initiated from his attempt at proving the nonexistence of polynomial time approximation schemes for EUCLIDEAN TSP.

The efforts are not always as successful as this for some other NP-hard optimization problems. We give an example below. A *Boolean variable* $x$ is a variable that can take values TRUE or FALSE. A *literal* is either a Boolean variable or a negation of a Boolean variable. A *clause* is a disjunction (i.e., OR) of literals. We say that an assignment makes a clause *satisfied* if the assignment makes at least one literal in the clause have value TRUE. Consider the following problem:

> MAX-2SAT
>
> Given a set $F$ of clauses, each containing at most 2 literals, find an assignment of the boolean variables in $F$ that maximizes the number of satisfied clauses.

In 1974, Johnson presented an approximation algorithm of ratio 2 for the MAX-2SAT problem. This ratio was then improved to $1.33\cdots$ in 1991, then to $1.138\cdots$ in 1994 and to $1.074\cdots$ in 1995 (see [38] for detailed review and references for this line of research, also see Section 9.3 for detailed discussions on some of these algorithms). One might expect that this line of research would eventually lead to a polynomial time approximation scheme for the MAX-2SAT problem. This, actually, is not possible since recent research has shown that unless P = NP, there is no polynomial time approximation algorithm of ratio 1.0476 for the MAX-2SAT problem [61].

Characterization of optimization problems that have no polynomial time approximation scheme has been a very active research area in the last three decades. Recently, deep and exciting advances have been made in this direction, which provide effective and powerful methods for identification of optimization problems with no polynomial time approximation scheme. We will describe these results systematically in the next part of the book (Chapters 9-11). In the following, we mention some simple techniques, which can be used to prove the nonexistence of polynomial time approximation schemes for certain optimization problems.

If an optimization problem remains NP-hard even when the optimal value for its objective function is very small, then we can derive immediately that the problem has no polynomial time approximation scheme (based on the assumption P $\neq$ NP). For example, observing for the BIN

PACKING problem that deciding whether the minimum number of bins for a given set of items is 2 is NP-hard, we derived immediately that there is no polynomial time approximation algorithm of ratio less than $3/2$ for the BIN PACKING problem (Theorem 7.1.2). In Section 7.2, based on the fact that deciding whether the edges of a graph can be colored with at most 3 colors is NP-hard, we derived that the GRAPH EDGE COLORING problem has no polynomial time approximation algorithm for ratio less than $4/3$ (Theorem 7.2.3). These, of course, exclude immediately the possibility of existence of polynomial time approximation schemes for the problems.

For certain optimization problems, it is possible to change the approximation ratio dramatically by trivial modifications in input instances. For this kind of problems, one may prove the nonexistence of polynomial time approximation schemes. Consider the general TRAVELING SALESMAN problem:

> TRAVELING SALESMAN
>
> Given a weighted complete graph $G$, construct a traveling salesman tour in $G$ with the minimum weight.

**Theorem 8.4.1** *If $P \neq NP$, then for any function $f(n) = O(c^n)$, where $c$ is a constant, the* TRAVELING SALESMAN *problem has no polynomial time approximation algorithm of ratio bounded by $f(n)$.*

PROOF. We first reduce the NP-hard problem HAMILTONIAN CIRCUIT to the TRAVELING SALESMAN problem. Recall that the HAMILTONIAN CIRCUIT problem is, for each given graph $G$, decides if there is a simple cycle in $G$ containing all the vertices in $G$ (such a cycle is called a *Hamiltonian circuit*). For an instance $G$ of $n$ vertices for the HAMILTONIAN CIRCUIT problem, we construct an instance $G'$ for the TRAVELING SALESMAN problem as follows. The graph $G'$ has the same set of vertices as $G$. For each pair of vertices $u$ and $v$, if $(u, v)$ is an edge in $G$, then the edge $(u, v)$ in $G'$ has weight 1, and if $(u, v)$ is not an edge in $G$, then the edge $(u, v)$ in $G'$ has weight $nf(n)$. It is easy to see that if the graph $G$ has a Hamiltonian circuit then the minimum traveling salesman tour in $G'$ has weight $n$, while if the graph $G$ has no Hamiltonian circuit then the minimum traveling salesman tour in $G'$ has weight at least $nf(n) + n - 1$. Also note that the condition $f(n) = O(c^n)$ ensures that the transformation from $G$ to $G'$ can be done in polynomial time.

If the TRAVELING SALESMAN problem had a polynomial time approximation algorithm $A$ of ratio bounded by $f(n)$, then we would be able to use this algorithm $A$ to solve the HAMILTONIAN CIRCUIT problem, as follows. Applying the approximation algorithm $A$ to the instance $G'$. Since the approximation ratio of $A$ is bounded by $f(n)$, in case the graph $G$ has a Hamiltonian circuit (i.e., the minimum salesman tour in $G'$ has weight $n$), the algorithm $A$ returns a salesman tour of weight at most $nf(n)$, while in case the graph $G$ has no Hamiltonian circuit (so the minimum salesman tour in $G'$ has weight at least $nf(n) + n - 1 > nf(n)$), the algorithm $A$ returns a salesman tour of weight larger than $nf(n)$ (we assume $n > 1$). Therefore, based on the weight of the salesman tour returned by the algorithm $A$, we can directly decide if the graph $G$ has a Hamiltonian circuit. This would solve the NP-hard problem HAMILTONIAN CIRCUIT in polynomial time, which in consequence would imply that P = NP. □

Note that Theorem 8.4.1 is actually much stronger than saying the the TRAVELING SALESMAN problem has no polynomial time approximation scheme.

# Part III

# Constant Ratio Approximable Problems

This part of the book concentrates on the study of optimization problems that have polynomial time approximation algorithms with approximation ratio bounded by a constant. These problems will be called *approximable optimization problems*.

For a given approximable optimization problem $Q$, development of an approximation algorithm in general involves four steps:

1. design a polynomial time approximation algorithm $A$ for $Q$;
2. analyze the algorithm $A$ and derive an upper bound $c_A$ on the approximation ratio for $A$;
3. study the optimility of the value $c_A$, i.e. is there another $c'_A < c_A$ such that $c'_A$ is also an upper bound for the approximation ratio for the algorithm $A$?
4. study the optimality of the algorithm $A$, i.e., is there another approximation algorithm $A'$ for the problem $Q$ such that the approximation ratio of $A'$ is smaller than that of $A$?

Step 1 may involve a wide range of techniques in general algorithm design. Many approximation algorithms are based intuition, experience, or deeper insight on the given problems. Popular techniques include the greedy method, branch and bound, and other combinatorical methods. Probabilistic method has also turned out to be very powerful. Step 2 is special in particular for the study of approximation algorithms. One challenging task in this step is the estimation of the value of an optimal solution, which is necessary in comparison with the value of the approximation solution given by the algorithm $A$ to derive the ratio $c_A$. To prove that the value $c_A$ is the best possible for the algorithm $A$ in Step 3, it suffices to construct a single instance $\alpha$ and show that the algorithm $A$ on the instance $\alpha$ gives a solution whose value is equal to $Opt(\alpha)$. In some cases, the algorithm designer through his development of the algorithm may have got some ideas about what are the "obstacles" for his algorithm. In this case, Step 3 may become pretty easy. However, there are also other examples of approximation algorithms, for which Step 3 have turned out to be extremely difficult. In most cases, Step 4 is the most challenging task in the study of approximation algorithms for optimization problems, which involves the study of "precise" and "intrinsic" polynomial time approximability for optimization problems.

Also note that if an optimization problem has polynomial time approximation scheme, then the answers to the questions in Step 3 and Step 4 become trivial. Therefore, powerful techniques for identifying optimization problems that have no polynomial time approximation scheme is also central in the study of approximable optimization problems.

The discussion in this part will be centered around the above issues. In Chapter 9, we present constant ratio approximation algorithms for a number of well-known approximable optimization problems, based on the polular greedy method. Chapter 10 introduces a more recently developed probabilistic method that turns out to be very powerful in developing approximation algorithms for optimization problems. In Chapter 11, we introduce the theory of Apx-completeness, which provides a powerful and systematical methodology for identifying approximable optimization problems with no polynomial time approximation scheme.

# Chapter 9

# Combinatorical Methods

Popular combinatorical methods in approximation algorithms include the greedy method, dynamic programming, branch and bound, local search, and combinatorical transformations.

In this chapter, we present approximation algorithms, based on these combinatorical methods, for a number of well-known NP-hard optimization problems, including the metric traveling salesman problem, the maximum satisfiability problem, the maximum 3-dimensional matching problem, and the minimum vertex cover problem. Note that these four problems are, respectively, the optimization versions of four of the six "basic" NP-complete problems according to Garey and Johnson [49]: the Hamiltonian circuit problem, the satisfiability problem, the 3-dimensional matching problem, and the vertex cover problem,

For each of the problems, we start with a simple approximation algorithm and analyze its approximation ratio. We then discuss how to derive improved the approximation ratio using more sophisticated techniques or more thorough analysis, or both.

## 9.1 Metric TSP

In Section 8.3, we have discussed in detail the traveling salesman problem in Euclidean space, and shown that the problem has a polynomial time approximation scheme. Euclidean spaces are special cases of a *metric space*, in which a non-negative function $\omega$ (the *metric*) is defined on pairs of points such that for any points $p_1$, $p_2$, $p_3$ in the space

(1) $\omega(p_1, p_2) = 0$ if and only if $p_1 = p_2$,

(2) $\omega(p_1, p_2) = \omega(p_2, p_1)$, and

(3) $\omega(p_1, p_2) \leq \omega(p_1, p_3) + \omega(p_3, p_2)$.

The third condition $\omega(p_1, p_2) \leq \omega(p_1, p_3) + \omega(p_3, p_2)$ is called the *triangle inequality*. In an Euclidean space, the metric between two points is the distance between the two points. Many non-Euclidean spaces are also metric spaces. An example is a traveling cost map in which points are cities while the metric between two cities is the cost for traveling between the two cities.

In this section, we consider the traveling salesman problem on a general metric space. Since the metric between two points $p_1$ and $p_2$ in a metric space can be represented by an edge of weight $\omega(p_1, p_2)$ between the two points, we can formulate the problem in terms of weighted graphs.

**Definition 9.1.1** A graph $G$ is a *metric graph* if $G$ is a weighted, undirected, and complete graph, in which edge weights are all positive and satisfy the triangle inequality.

A *salesman tour* $\pi$ in a metric graph $G$ is a simple cycle in $G$ that contains all vertices of $G$. The *weight* $wt(\pi)$ of the salesman tour $\pi$ is the sum of weights of the edges in the tour. The traveling salesman problem on metric graphs is formally defined as follows.

METRIC TSP

$I_Q$: the set of all metric graphs

$S_Q$: $S_Q(G)$ is the set of all salesman tours in $G$

$f_Q$: $f_Q(G, \pi)$ is the weight of the salesman tour $\pi$ in $G$

$opt_Q$: min

Since EUCLIDEAN TSP is NP-hard in the strong sense [47, 100], and EUCLIDEAN TSP is a subproblem of METRIC TSP, we derive that METRIC TSP is also NP-hard in the strong sense and, by Theorem 6.4.8, METRIC TSP has no fully polynomial time approximation scheme unless P = NP.

We will show in Chapter 11 that METRIC TSP is actually "harder" than EUCLIDEAN TSP in the sense that METRIC TSP has no polynomial time approximation scheme unless P = NP. In this section, we present approximation algorithms with approximation ratio bounded by a constant for the problem METRIC TSP.

### 9.1.1 Approximation based on a minimum spanning tree

Our first approximation algorithm for METRIC TSP is based on minimum spanning trees. See the algorithm presented in Figure 9.1, here the con-

> **Algorithm. MTSP-Apx-I**
>
> Input: a metric graph $G$
> Output: a salesman tour $\pi$ in $G$, given in an array $V[1..n]$
>
> 1. construct a minimum spanning tree $T$ for $G$;
> 2. let $r$ be the root of $T$; $i = 0$;
> 3. Seq($r$)
>
> **Seq**($v$)
>
> 1. $i = i + 1$;
> 2. $V[i] = v$;
> 3. **for** each child $w$ of $v$ **do**
>    Seq($w$);

Figure 9.1: Approximating METRIC TSP

structed salesman tour is given in the array $V[1..n]$ as a (cyclic) sequence of the vertices in $G$, in the order the vertices appear in the tour.

The minimum spanning tree $T$ can be constructed in time $O(nm)$ (see Section 1.3.1 for detailed discussion). Therefore, step 1 of the algorithm **MTSP-Apx-I** takes time $O(nm)$. Step 3 calls a recursive subroutine **Seq**($r$), which is essentially a depth first search procedure on the minimum spanning tree $T$ to order the vertices of $T$ in terms of their depth first search numbers (see Appendix A). Since the depth first search process takes time $O(m + n)$ on a graph of $n$ vertices and $m$ edges, step 3 of the algorithm **MTSP-Apx-I** takes time $O(n)$. In conclusion, the time complexity of the algorithm **MTSP-Apx-I** is $O(nm)$.

The depth first search process **Seq**($r$) on the tree $T$ can be regarded as a closed walk $\pi_0$ in the tree (a *closed walk* is a cycle in $T$ in which each vertex may appear more than once). Each edge $[u, v]$, where $u$ is the father of $v$ in $T$, is traversed exactly twice in the walk $\pi_0$: the first time when **Seq**($u$) calls **Seq**($v$) we traverse the edge from $u$ to $v$, and the second time when **Seq**($v$) is finished and returns back to **Seq**($u$) we traverse the edge from $v$ to $u$. Therefore, the walk $\pi_0$ has weight exactly twice the weight of the tree $T$. It is also easy to see that the list $V[1..n]$ produced by the algorithm **MTSP-Apx-I** can be obtained from the walk $\pi_0$ by deleting for each vertex $v$ all but the first occurrence of $v$ in the list $\pi_0$. Since each vertex appears exactly once in the list $V[1..n]$, $V[1..n]$ corresponds to a salesman tour $\pi$ in the metric graph $G$.

**Example.** Consider the tree $T$ in Figure 9.2, where $r$ is the root of the

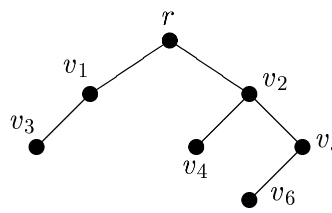



Figure 9.2: The minimum spanning tree $T$

tree $T$. The depth first search process (i.e., the subroutine **Seq**) traverses the tree $T$ in the order

$$\pi_0 : \quad r, v_1, v_3, v_1, r, v_2, v_4, v_2, v_5, v_6, v_5, v_2, r$$

By deleting for each vertex $v$ all but the first vertex occurrence for $v$, we obtain the list of vertices of the tree $T$ sorted by their depth first search numbers

$$\pi : \quad r, v_1, v_3, v_2, v_4, v_5, v_6,$$

Deleting a vertex occurrence of $v$ in the list $\{\cdots uvw \cdots\}$ is equivalent to replacing the path $\{u, v, w\}$ of two edges by a single edge $[u, w]$. Since the metric graph $G$ satisfies the triangle inequality, deleting vertex occurrences from the walk $\pi_0$ does not increase the weight of the walk. Consequently, the weight of the salesman tour $\pi$ given in the array $V[1..n]$ is not larger than the weight of the closed walk $\pi_0$, which is bounded by 2 times the weight of the minimum spanning tree $T$.

Since removing any edge (of positive weight) from any minimum salesman tour results in a spanning tree of the metric graph $G$, the weight of a minimum salesman tour in $G$ is at least as large as the weight of the minimum spanning tree $T$. In conclusion, the salesman tour $\pi$ given in the array $V[1..n]$ by the algorithm **MTSP-Apx-I** has its weight bounded by 2 times the weight of a minimum salesman tour. This gives the following theorem.

**Theorem 9.1.1** *The approximation ratio of the algorithm* **MTSP-Apx-I** *is bounded by* 2.

Two natural questions follow from Theorem 9.1.1. First, we have shown that the ratio of the weight $wt(\pi)$ of the salesman tour $\pi$ constructed by the algorithm **MTSP-Apx-I** and the weight $wt(\pi_o)$ of a minimum salesman tour $\pi_o$ is bounded by 2. Is it possible, by a more careful analysis, to show that $wt(\pi)/wt(\pi_o) \leq c$ for a smaller constant $c < 2$? Second, is there a

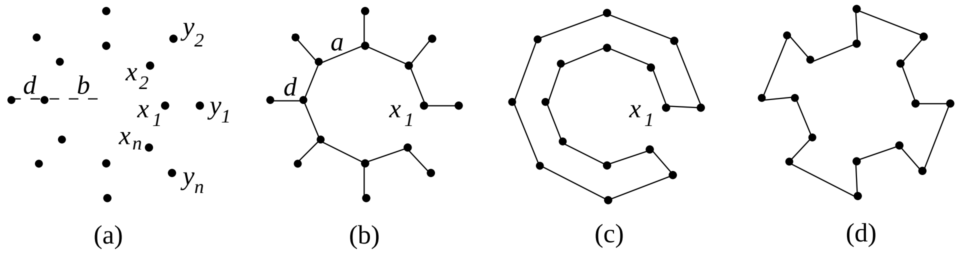



Figure 9.3: METRIC TSP instance for **MTSP-Apx-I**.

polynomial time approximation algorithm for METRIC TSP whose approximation ratio is better than that of the approximation algorithm **MTSP-Apx-I**?

These two questions constitute two important and in general highly non-trivial topics in the study of approximation algorithms. Essentially, the first question asks whether our analysis is the best possible *for the algorithm*, while the second question asks whether our algorithm is the best possible *for the problem*.

The answer to the first question some times is easy if we can find an instance for the given problem on which the solution constructed by our algorithm reaches the specified approximation ratio. In some cases, such instances can be realized during our analysis on the algorithm: these instances are the obstacles preventing us from further lowering down the approximation ratio in our analysis. However, there are also situations in which finding such instances is highly non-trivial.

The algorithm **MTSP-Apx-I** for the METRIC TSP problem belongs to the first category. We give below simple instances for METRIC TSP to show that the ratio 2 is tight for the algorithm in the sense that there are instances for METRIC TSP for which the algorithm **MTSP-Apx-I** produces solutions with approximation ratio arbitrarily close to 2.

Consider the figures in Figure 9.3, where our metric space is the Euclidean plane and the metric between two points is the Euclidean distance between the two points. Suppose we are given $2n$ points on the Euclidean plane with polar coordinates $x_k = (b, 360k/n)$ and $y_k = (b+d, 360k/n)$, $k = 1, \ldots, n$, where $d$ is much smaller than $b$. See Figure 9.3(a). It is not hard (for example, by Kruskal's algorithm for minimum spanning trees [28]) to see that the edges $[x_k, x_{k+1}]$, $k = 1, \ldots, n-1$ and $[x_j, y_j]$, $j = 1, \ldots, n$ form a minimum spanning tree $T$ for the set of points. See Figure 9.3(b). Now if we perform a depth first search on $T$ starting from the vertex $x_1$ and construct a salesman tour, we will get a salesman tour $\pi_c$ that is shown in Figure 9.3(c), while an optimal salesman tour $\pi_d$ is shown in Figure 9.3(d).

The weight of the salesman tour $\pi_c$ is about $2a(n-1)+2d$, where $a$ is the distance between two adjacent points $x_k$ and $x_{k+1}$ (note that when $d$ is sufficiently small compared with $a$, the distance between two adjacent points $y_k$ and $y_{k+1}$ is roughly equal to the distance between the two corresponding points $x_k$ and $x_{k+1}$), while the optimal salesman tour $\pi_d$ has weight roughly $nd + na$. When $d$ is sufficiently small compared with $a$ and when $n$ is sufficiently large, the ratio of the weight of the tour $\pi_c$ and the weight of the tour $\pi_d$ can be arbitrarily close to 2.

### 9.1.2 Christofides' algorithm

Now we turn our attention to the section question. Is the approximation algorithm **MTSP-Apx-I** the best possible for the problem METRIC TSP?

Let us look at the algorithm **MTSP-Apx-I** in Figure 9.1 in detail. After the minimum spanning tree $T$ is constructed, we traverse the tree $T$ by a depth first search process (the subroutine **Seq**) in which each edge of $T$ is traversed exactly twice. This process can be re-interpreted as follows:

1. construct a minimum spanning tree;

2. double each edge of $T$ into two edges, each of which has the same weight as the original edge. Let the resulting graph be $D$;

3. make a closed walk $W$ in the graph $D$ such that each edge of $D$ is traversed exactly once in $W$;

4. use "shortcuts", i.e., delete all but the first occurrence for each vertex in the walk $W$ to make a salesman tour $\pi$.

There are three crucial facts that make the above algorithm correctly produce a salesman tour with approximation ratio 2: (1) the graph $D$ gives a closed walk $W$ in the graph $G$ that contains all vertices of $G$; (2) the total weight of the closed walk $W$ is bounded by 2 times the weight of an optimal salesman tour; and (3) the shortcuts do not increase the weight of the closed walk $W$ so that we can derive a salesman tour $\pi$ from $W$ without increasing the weight of the walk.

If we can construct a graph $D'$ that gives a closed walk $W'$ with weight smaller than that of $W$ constructed by the algorithm **MTSP-Apx-I** such that $D'$ contains all vertices of $G$, then using the shortcuts on $W'$ should derive a better approximation to the optimal salesman tour.

Graphs whose edges constitute a single closed walk have been studied based on the following concept.

**Definition 9.1.2** An *Eulerian tour* in a graph $G$ is a closed walk in $G$ that traverses each edge of $G$ exactly once. An undirected connected graph $G$ is an *Eulerian graph* if it contains an Eulerian tour.

Eulerian graphs have been extensively studied in graph theory literature (see for example, [59]). Recent research has shown that Eulerian graphs play an important role in designing efficient parallel graph algorithms [79]. A proof of the following theorem can be found in Appendix A (see Theorems A.1 and A.2).

**Theorem 9.1.2** *An undirected connected graph $G$ is an Eulerian graph if and only if every vertex of $G$ has an even degree. Moreover, there is a linear time algorithm that, given an Eulerian graph $G$, constructs an Eulerian tour in $G$.*

Thus, the graph $D$ described above for the algorithm **MTSP-Apx-I** is actually an Eulerian graph and the closed walk $W$ is an Eulerian tour in $D$. Now we consider how a better Eulerian graph $D'$ can be constructed based on the minimum spanning tree $T$, which leads to a better approximation to the minimum salesman tour.

Let $G$ be a metric graph, an input instance of the METRIC TSP problem and let $T$ be a minimum spanning tree in $G$. We have

**Lemma 9.1.3** *The number of vertices of the tree $T$ that has an odd degree in $T$ is even.*

PROOF. Let $v_1, \ldots, v_n$ be the vertices of the tree $T$. Since each edge $e = [v_i, v_j]$ of $T$ contributes one degree to $v_i$ and one degree to $v_j$, and $T$ has exactly $n - 1$ edges, we must have

$$\sum_{i=1}^{n} deg_T(v_i) = 2(n-1)$$

where $deg_T(v_i)$ is the degree of the vertex $v_i$ in the tree $T$. We partition the set of vertices of $T$ into odd degree vertices and even degree vertices. Then we have

$$\sum_{v_i:\ \text{even degree}} deg_T(v_i) + \sum_{v_j:\ \text{odd degree}} deg_T(v_j) = 2(n-1)$$

**Algorithm. Christofides**

Input: a metric graph $G$
Output: a salesman tour $\pi'$ in $G$

1. construct a minimum spanning tree $T$ for $G$;
2. let $v_1, \ldots, v_{2h}$ be the odd degree vertices in $T$;
   construct a minimum weighted perfect matching $M_h$ in the complete graph $H$ induced by the vertices $v_1, \ldots, v_{2h}$;
3. construct an Eulerian tour $W'$ in the Eulerian graph $D' = T + M_h$;
4. use shortcuts to derive a salesman tour $\pi'$ from $W'$;
5. return $\pi'$.

Figure 9.4: Christofides' Algorithm for Metric TSP

Since both $\sum_{v_i:\text{ even degree}} deg_T(v_i)$ and $2(n-1)$ are even numbers, the value $\sum_{v_j:\text{ odd degree}} deg_T(v_j)$ is also an even number. Consequently, the number of vertices that have odd degree in $T$ must be even. $\square$

By Lemma 9.1.3, we can suppose, without loss of generality, that $v_1$, $v_2$, $\ldots$, $v_{2h}$ are the odd degree vertices in the tree $T$. The vertices $v_1$, $v_2$, $\ldots$, $v_{2h}$ induce a complete subgraph $H$ in the original metric graph $G$ (recall that a metric graph is a complete graph). Now construct a minimum weighted perfect matching $M_h$ in $H$ (a *perfect matching* in a complete graph of $2h$ vertices is a matching of $h$ edges. See Section 3.5 for more detailed discussion). Since each of the vertices $v_1$, $v_2$, $\ldots$, $v_{2h}$ has degree 1 in the graph $M_h$, adding the edges in $M_h$ to the tree $T$ results in a graph $D' = T + M_h$ in which all vertices have an even degree. By Theorem 9.1.2, the graph $D'$ is an Eulerian graph. Moreover, the graph $D'$ contains all vertices of the original metric graph $G$. We are now able to derive a salesman tour $\pi'$ from $D'$ by using shortcuts.

We formally present this in the algorithm given in Figure 9.4. The algorithm is due to Christofides [24].

According to Theorem 3.5.5, the minimum weighted perfect matching $M_h$ in the complete graph $H$ induced by the vertices $v_1, \ldots, v_{2h}$ can be constructed in time $O(h^3) = O(n^3)$. By Theorem 9.1.2, step 3 of the algorithm **Christofides** takes linear time. Thus, the algorithm **Christofides** runs in time $O(n^3)$.

Now we study the approximation ratio for the algorithm **Christofides**.

**Lemma 9.1.4** *The weight of the minimum weighted perfect matching $M_h$*

*in the complete graph $H$ induced by the vertices $v_1, \ldots, v_{2h}$, $\sum_{e \in M_h} wt(e)$, is at most $1/2$ of the weight of a minimum salesman tour in the graph $G$.*

PROOF. Let $\pi_o$ be an optimal salesman tour in the metric graph $G$. By using shortcuts, i.e., by removing the vertices that are not in $\{v_1, v_2, \ldots, v_{2h}\}$ from the tour $\pi_o$, we obtain a simple cycle $\pi$ that contains exactly the vertices $v_1, \ldots, v_{2h}$. Since the metric graph $G$ satisfies the triangle inequality, the weight of $\pi$ is not larger than the weight of $\pi_o$.

The simple cycle $\pi$ can be decomposed into two disjoint perfect matchings in the complete graph $H$ induced by the vertices $v_1, \ldots, v_{2h}$: one matching is obtained by taking every other edge in the cycle $\pi$, and the other matching is formed by the rest of the edges. Of course, both of these two perfect matchings in $H$ have weight at least as large as the minimum weighted perfect matching $M_h$ in $H$. This gives

$$wt(\pi_o) \geq wt(\pi) \geq 2 \cdot wt(M_h)$$

This completes the proof. $\Box$

Now the analysis is clear. We have $D' = T + M_h$. Thus

$$wt(D') = wt(T) + wt(M_h)$$

As we discussed in the analysis for the algorithm **MTSP-Apr-I**, the weight of the minimum spanning tree $T$ of the metric graph $G$ is not larger than the weight of a minimum salesman tour for $G$. Combining this with Lemma 9.1.4, we conclude that the weight of the Eulerian graph $D'$ is bounded by 1.5 times the weight of a minimum salesman tour in $G$. Thus, the Eulerian tour $W'$ constructed in step 3 of the algorithm **Christofides** has weight bounded by 1.5 times the weight of a minimum salesman tour in $G$. Finally, the salesman tour $\pi'$ constructed by the algorithm **Christofides** is obtained by using shortcuts on the Eulerian tour $W'$ and the metric graph $G$ satisfies the triangle inequality. Thus, the weight of the salesman tour $\pi'$ constructed by the algorithm **Christofides** is bounded by 1.5 times the weight of a minimum salesman tour in $G$. This is concluded in the following theorem.

**Theorem 9.1.5** *The algorithm* **Christofides** *for the* METRIC TSP *problem runs in time $O(n^3)$ and has approximation ratio* 1.5.

As for the algorithm **MTSP-Apx-I**, one can show that the ratio 1.5 is tight for the algorithm **Christofides**, in the sense that there are input instances for METRIC TSP for which the algorithm **Christofides** produces salesman tours whose weights are arbitrarily close to 1.5 times the weights of minimum salesman tours. The readers are encouraged to construct these instances for a deeper understanding of the algorithm.

It has been a well-known open problem whether the ratio 1.5 can be further improved for approximation algorithms for the METRIC TSP problem. In Chapter 11, we will show that the METRIC TSP problem has no polynomial time approximation scheme unless P = NP. This implies that there is a constant $c > 1$ such that no polynomial time approximation algorithm for METRIC TSP can have approximation ratio smaller than $c$ (under the assumption P $\neq$ NP). However, little has been known for this constant $c$.

## 9.2 Maximum satisfiability

Let $X = \{x_1, \ldots, x_n\}$ be a set of boolean variables. A *literal* in $X$ is either a boolean variable $x_i$ or its negation $\overline{x_i}$, for some $1 \leq i \leq n$. A *clause* on $X$ is a disjunction, i.e., an OR, of a set of literals in $X$. We say that a truth assignment to $\{x_1, \ldots, x_n\}$ *satisfies* a clause if the assignment makes at least one literal in the clause TRUE, and we say that a set of clauses is *satisfiable* if there is an assignment that satisfies all clauses in the set.

SATISFIABILITY (SAT)

INPUT: a set $F = \{C_1, C_2, \ldots, C_m\}$ of clauses on $\{x_1, \ldots, x_n\}$

QUESTION: is $F$ satisfiable?

The SAT problem is the "first" NP-complete problem, according to the famous Cook's Theorem (see Theorem 1.4.2 in Chapter 1).

If we have further restrictions on the number of literals in each clause, we obtain an interesting subproblem for SAT.

$k$-SATISFIABILITY ($k$-SAT)

INPUT: a set $F = \{C_1, C_2, \ldots, C_m\}$ of clauses on $\{x_1, \ldots, x_n\}$ such that each clause has at most $k$ literals

QUESTION: is $F$ satisfiable?

It is well-known that the $k$-SAT problem remains NP-complete for $k \geq 3$, while the 2-SAT problem can be solved in polynomial time (in fact, in linear time). Interested readers are referred to [28] for details.

As the SAT problem plays a fundamental role in the study of NP-completeness theory, an optimization version of the SAT problem, the MAX-SAT problem, plays a similar role in the study of approximation algorithms.

> MAXIMUM SATISFIABILITY (MAX-SAT)
>
> INPUT: a set $F = \{C_1, C_2, \dots, C_m\}$ of clauses on $\{x_1, \dots, x_n\}$
>
> OUTPUT: a truth assignment on $\{x_1, \dots, x_n\}$ that satisfies the maximum number of the clauses in $F$

The optimization version for the $k$-SAT problem is defined similarly.

> MAXIMUM $k$-SATISFIABILITY (MAX-$k$SAT)
>
> INPUT: a set $F = \{C_1, C_2, \dots, C_m\}$ of clauses on $\{x_1, \dots, x_n\}$ such that each clause has at most $k$ literals
>
> OUTPUT: a truth assignment on $\{x_1, \dots, x_n\}$ that satisfies the maximum number of the clauses in $F$

It is easy to see that the SAT problem can be reduced in polynomial time to the MAX-SAT problem: a set $\{C_1, \dots, C_m\}$ of clauses is a yes-instance for the SAT problem if and only if when it is regarded as an instance of the MAX-SAT problem, its optimal value is $m$. Therefore, the MAX-SAT problem is NP-hard. Similarly, the $k$-SAT problem for $k \geq 3$ can be reduced in polynomial time to the MAX-$k$SAT problem so the MAX-$k$SAT problem is NP-hard for $k \geq 3$.

Since the 2-SAT problem can be solved in linear time, one may expect that the corresponding optimization problem MAX-2SAT is also easy. However, the following theorem gives a bit surprising result.

**Theorem 9.2.1** *The* MAX-2SAT *problem is NP-hard.*

PROOF. We show that the NP-complete problem 3-SAT can be reduced in polynomial time to the MAX-2SAT problem.

Let $F = \{C_1, \dots, C_m\}$ be an instance for the 3-SAT problem, where each $C_i$ is a clause of at most three literals in $\{x_1, \dots, x_n\}$. The set $F$ may contain clauses with fewer than three literals. We first show how to convert $F$ into an instance for 3-SAT in which all clauses have exactly three literals.

If a clause $C_i$ in $F$ has exactly two literals: $C_i = (l_1 \vee l_2)$, then we replace $C_i$ by two clauses of three literals $(l_1 \vee l_2 \vee y_1)$ and $(l_1 \vee l_2 \vee \overline{y_1})$, where $y_1$ is a new boolean variable; if a clause $C_j$ in $F$ has exactly one literal: $C_j = (l_3)$, then we replace $C_j$ by four clauses of three literals $(l_3 \vee y_2 \vee y_3)$, $(l_3 \vee y_2 \vee \overline{y_3})$,

$(l_3 \vee \overline{y_2} \vee y_3)$, and $(l_3 \vee \overline{y_2} \vee \overline{y_3})$, where $y_2$ and $y_3$ are new variables. The resulting set $F'$ of clauses is still an instance for 3-SAT in which each clause has exactly three literals. It is straightforward to see that the instance $F$ is satisfiable if and only if the instance $F'$ is satisfiable.

Thus, we can assume, without loss of generality, that each clause in the given instance $F$ for the 3-SAT problem has exactly three literals.

Consider a clause $C_i = (a_i \vee b_i \vee c_i)$ in $F$, where $a_i$, $b_i$, and $c_i$ are literals in $\{x_1, \ldots, x_n\}$. We construct a set of ten clauses:

$$\begin{aligned} F_i \;=\; & \{(a_i),\;\; (b_i),\;\; (c_i),\;\; (y_i),\;\; (\overline{a_i} \vee \overline{b_i}),\;\; (\overline{a_i} \vee \overline{c_i}), \\ & (\overline{b_i} \vee \overline{c_i}),\;\; (a_i \vee \overline{y_i}),\;\; (b_i \vee \overline{y_i}),\;\; (c_i \vee \overline{y_i})\} \end{aligned} \tag{9.1}$$

where $y_i$ is a new variable. It is easy to verify the following facts.

- if all $a_i$, $b_i$, $c_i$ are set FALSE, then any assignment to $y_i$ can satisfy at most 6 clauses in $F_i$;
- if at least one of $a_i$, $b_i$, $c_i$ is set TRUE, then there is an assignment to $y_i$ that satisfies 7 clauses in $F_i$, and no assignment to $y_i$ can satisfy more than 7 clauses in $F_i$.

Let $F'' = F_1 \cup F_2 \cup \cdots \cup F_m$ be the set of the $10m$ clauses constructed from the $m$ clauses in $F$ using the formula given in (9.1). The set $F''$ is an instance for the MAX-2SAT problem. It is easy to see that the set $F''$ can be constructed in polynomial time from the set $F$.

Suppose that $F$ is a yes-instance for the 3-SAT problem. Then there is an assignment $S_x$ to $\{x_1, \ldots, x_n\}$ that satisfies at least one literal in each $C_i$ of the clauses in $F$. According to the analysis given above, this assignment $S_x$ plus a proper assignment $S_y$ to the new variable set $\{y_1, \ldots, y_m\}$ will satisfy 7 clauses in the set $F_i$, for each $i = 1, \ldots, m$. Thus, the assignment $S_x + S_y$ to the boolean variables $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ satisfies $7m$ clauses in $F''$. Since no assignment can satisfy more than 7 clauses in each set $F_i$, we conclude that in this case the optimal value for the instance $F''$ of MAX-2SAT is $7m$.

Now suppose that $F$ is a no-instance for the 3-SAT problem. Let $S'$ be any assignment to $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$. The assignment $S'$ can be decomposed into an assignment $S'_x$ to $\{x_1, \ldots, x_n\}$ and an assignment $S'_y$ to $\{y_1, \ldots, y_m\}$. Since $F$ is a no-instance for the 3-SAT problem, for at least one clause $C_i$ in $F$, the assignment $S'_x$ makes all literals false. According to our previous analysis, any assignment to $y_i$ plus the assignment $S'_x$ can satisfy at most 6 clauses in the corresponding set $F_i$. Moreover, since no assignment

**Algorithm. Johnson**

Input: a set of clauses $F = \{C_1, \ldots, C_m\}$ on $\{x_1, \ldots, x_n\}$
Output: a truth assignment $\tau$ to $\{x_1, \ldots, x_n\}$

1. **for** each clause $C_j$ **do** $w(C_j) = 1/2^{|C_j|}$
2. $L = \{C_1, \ldots, C_m\}$;
3. **for** $t = 1$ **to** $n$ **do**
    find all clauses $C_1^T, \ldots, C_q^T$ in $L$ that contain $x_t$;
    find all clauses $C_1^F, \ldots, C_s^F$ in $L$ that contain $\overline{x_t}$;
    **if** $\sum_{i=1}^{q} w(C_i^T) \geq \sum_{i=1}^{s} w(C_i^F)$
    **then** $\tau(x_t) = \text{TRUE}$; delete $C_1^T, \ldots, C_q^T$ from $L$;
        **for** $i = 1$ **to** $s$ **do** $w(C_i^F) = 2w(C_i^F)$
    **else** $\tau(x_t) = \text{FALSE}$; delete $C_1^F, \ldots, C_s^F$ from $L$;
        **for** $i = 1$ **to** $q$ **do** $w(C_i^T) = 2w(C_i^T)$

Figure 9.5: Johnson's Algorithm

to $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ can satisfy more than 7 clauses in each set $F_j$, for $j = 1, \ldots, m$, we conclude that the assignment $S'$ can satisfy at most $7(m-1)+6 = 7m-1$ clauses in $F''$. Since $S'$ is arbitrary, we conclude that in this case, no assignment to $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ can satisfy more than $7m-1$ clauses in $F''$. Thus, in this case the optimal value for the instance $F''$ for MAX-2SAT is at most $7m-1$.

Summarizing the discussion above, we conclude that the set $F$ of $m$ clauses of three literals is a yes-instance for the 3-SAT problem if and only if the optimal value for the instance $F''$ of MAX-2SAT is $7m$. Consequently, the 3-SAT problem is polynomial time reducible to the MAX-2SAT problem. We conclude that the MAX-2SAT problem is NP-hard. □

### 9.2.1 Johnson's algorithm

Now we present an approximation algorithm for the MAX-SAT problem, due to David Johnson [71]. Consider the algorithm given in Figure 9.5, where for a clause $C_i$, we use $|C_i|$ to denote the number of literals in $C_i$.

The algorithm **Johnson** obviously runs in polynomial time. We analyze the approximation ratio for the algorithm.

**Lemma 9.2.2** *If each clause in the input instance $F$ contains at least $k$ literals, then the algorithm* **Johnson** *constructs an assignment that satisfies at least $m(1-1/2^k)$ clauses in $F$, where $m$ is the number of clauses in $F$.*

PROOF. In the algorithm **Johnson**, once a literal in a clause is set to TRUE, i.e., the clause is satisfied, the clause is removed from the set $L$. Therefore, the number of clauses that are not satisfied by the constructed assignment $\tau$ is equal to the number of clauses left in the set $L$ at the end of the algorithm.

Each clause $C_i$ is associated with a weight value $w(C_i)$. Initially, we have $w(C_i) = 1/2^{|C_i|}$ for all $C_i$. By our assumption, each clause $C_i$ contains at least $k$ literals. So initially we have

$$\sum_{C_i \in L} w(C_i) = \sum_{i=1}^{m} w(C_i) = \sum_{i=1}^{m} 1/2^{|C_i|} \leq \sum_{i=1}^{m} 1/2^k = m/2^k$$

In step 3, we update the set $L$ and the weight for the clauses in $L$. It can be easily seen that we never increase the value $\sum_{C_i \in L} w(C_i)$: each time we update the set $L$, we remove a heavier set of clauses from $L$ and double the weight for a lighter set of clauses remaining in $L$. Therefore, at end of the algorithm we should still have

$$\sum_{C_i \in L} w(C_i) \leq m/2^k \tag{9.2}$$

At the end of the algorithm, all boolean variables $\{x_1, \ldots, x_n\}$ have been assigned a value. A clause $C_i$ left in the set $L$ has been considered by the algorithm exactly $|C_i|$ times and each time the corresponding literal in $C_i$ was assigned FALSE. Therefore, for each literal in $C_i$, the weight of the clause $C_i$ is doubled once. Since initially the clause $C_i$ has weight $1/2^{|C_i|}$ and its weight is doubled exactly $|C_i|$ times in the algorithm, we conclude that at the end of the algorithm, the clause $C_i$ left in $L$ has weight 1. Combining this with the inequality (9.2), we conclude that at the end of the algorithm, the number of clauses in the set $L$ is bounded by $m/2^k$. In other words, the number of clauses satisfied by the constructed assignment $\tau$ is at least $m - m/2^k = m(1 - 1/2^k)$. The lemma is proved. □

The observation given in Lemma 9.2.2 derives the following bound on the approximation ratio for the algorithm **Johnson** immediately.

**Theorem 9.2.3** *The algorithm* **Johnson** *for the* MAX-SAT *problem has its approximation ratio bounded by* 2.

PROOF. According to Lemma 9.2.2, on an input $F$ of $m$ clauses, each containing at least $k$ literals, the algorithm **Johnson** constructs an assignment that satisfies at least $m(1 - 1/2^k)$ clauses in $F$. Since a clause in the input $F$ contains at least one literal, i.e., $k \geq 1$, we derive that for any instance $F$ for MAX-SAT, the assignment constructed by the algorithm **Johnson** satisfies at least $m(1 - 1/2) = m/2$ clauses in $F$. Since the optimal value for the instance $F$ is obviously bounded by $m$, the approximation ratio for the algorithm must be bounded by $\frac{m}{m/2} = 2$. $\square$

The algorithm **Johnson** has played an important role in the study of approximation algorithms for the MAX-SAT problem. In particular, the algorithm is an excellent illustration for the probabilistic method, which has been playing a more and more important role in the design and analysis of approximation algorithms for NP-hard optimization problems. We will reconsider the algorithm **Johnson** in the next chapter from a different point of view.

### 9.2.2 Revised analysis on Johnson's algorithm

Theorem 9.2.3 claims that the algorithm **Johnson** has approximation ratio bounded by 2. Is the bound 2 tight for the algorithm? In this subsection, we provide a more careful analysis on the algorithm and show that the approximation ratio of the algorithm is actually 1.5. Readers may skip this subsection in their first reading.

In order to analyze the algorithm **Johnson**, we may need to "flip" a boolean variable $x_t$, i.e., interchange $x_t$ and $\overline{x_t}$, in an instance for MAX-SAT. This may change the set of clauses satisfied by the assignment $\tau$ constructed by the algorithm. In order to take care of this abnormality, we will augment the algorithm **Johnson** by a boolean array $b[1..n]$. The augmented boolean array $b[1..n]$ will be part of the input to the algorithm. We call such an algorithm the *augmented Johnson's algorithm*. Our first analysis will be performed on the augmented Johnson's algorithm with an *arbitrarily* augmented boolean array. The bound on the approximation ratio for the augmented Johnson's algorithm will imply the same bound for the original algorithm **Johnson**.

The augmented Johnson's algorithm is given in Figure 9.6.

The only difference between the original algorithm **Johnson** and the augmented Johnson's algorithm is that in case $\sum_{i=1}^{q} w(C_i^T) = \sum_{i=1}^{s} w(C_i^F)$, the original algorithm **Johnson** assigns $\tau(x_t) = \text{TRUE}$ while the augmented Johnson's algorithm assigns $\tau(x_t) = b[t]$.

In the following, we prove a lemma for the augmented Johnson's algorithm. To do this, we need to introduce some terminologies and notations.

**Augmented Johnson's Algorithm.**

Input: a set $F$ of clauses on $\{x_1, \ldots, x_n\}$, and a boolean array $b[1..n]$
Output: a truth assignment $\tau$ to $\{x_1, \ldots, x_n\}$

1. **for** each clause $C_j$ in $F$ **do** $w(C_j) = 1/2^{|C_j|}$
2. $L = F$;
3. **for** $t = 1$ **to** $n$ **do**
   - find all clauses $C_1^T, \ldots, C_q^T$ in $L$ that contain $x_t$;
   - find all clauses $C_1^F, \ldots, C_s^F$ in $L$ that contain $\overline{x_t}$;
   - **case 1.** $\left(\sum_{i=1}^q w(C_i^T) > \sum_{i=1}^s w(C_i^F)\right)$ **or** $\left(\sum_{i=1}^q w(C_i^T) = \sum_{i=1}^s w(C_i^F) \text{ and } b[t] = \text{TRUE}\right)$
     - $\tau(x_t) = \text{TRUE}$; delete $C_1^T, \ldots, C_q^T$ from $L$;
     - **for** $i = 1$ **to** $s$ **do** $w(C_i^F) = 2w(C_i^F)$
   - **case 2.** $\left(\sum_{i=1}^q w(C_i^T) < \sum_{i=1}^s w(C_i^F)\right)$ **or** $\left(\sum_{i=1}^q w(C_i^T) = \sum_{i=1}^s w(C_i^F) \text{ and } b[t] = \text{FALSE}\right)$
     - $\tau(x_t) = \text{FALSE}$; delete $C_1^F, \ldots, C_s^F$ from $L$;
     - **for** $i = 1$ **to** $q$ **do** $w(C_i^T) = 2w(C_i^T)$

Figure 9.6: the augmented Johnson's algorithm

A literal is a *positive literal* if it is a boolean variable $x_i$ for some $i$, and a *negative literal* if it is the negation $\overline{x_i}$ of a boolean variable.

Fix an instance $F = \{C_1, \ldots, C_m\}$ for MAX-SAT and let $b[1..n]$ be any fixed boolean array. Let $r$ be the maximum number of literals in a clause in $F$. Apply the augmented Johnson's algorithm on $F$ and $b[1..n]$. Consider a fixed moment in the execution of the augmented Johnson's algorithm. We say that a literal is still *active* if it has not been assigned a truth value yet. A clause $C_j$ in $F$ is *satisfied* if at least one literal in $C_j$ has been assigned value TRUE. A clause $C_j$ is *killed* if all literals in $C_j$ are assigned value FALSE. A clause $C_j$ is *negative* if it is neither satisfied nor killed, and all active literals in $C_j$ are negative literals.

**Definition 9.2.1** Fix a $t$, $0 \leq t \leq n$, and suppose that we are at the end of the $t^{th}$ iteration of the **for** loop in step 3 of the augmented Johnson's algorithm. Let $S^{(t)}$ be the set of satisfied clauses, $K^{(t)}$ be the set of killed clauses, and $N_i^{(t)}$ be the set of negative clauses with exactly $i$ active literals.

For a set $S$ of clauses, denote by $|S|$ the number of clauses in $S$, and let $w(S) = \sum_{C_j \in S} w(C_j)$.

**Lemma 9.2.4** *For all $t$, $0 \le t \le n$, the sets $S^{(t)}$, $K^{(t)}$, and $N_i^{(t)}$ satisfy the following condition:*

$$|S^{(t)}| \ge 2|K^{(t)}| + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} - A_0$$

*where $A_0 = \sum_{i=1}^{r} |N_i^{(0)}|/2^{i-1}$.*

PROOF. The proof proceeds by induction on $t$. For $t = 0$, since $S^{(0)} = K^{(0)} = \emptyset$, and $\sum_{i=1}^{r} |N_i^{(t)}|/2^{i-1} = A_0$, the lemma is true.

Suppose $t > 0$. We need to introduce two more notations. At the end of the $t^{th}$ iteration for the **for** loop in step 3 of the augmented Johnson's algorithm, let $P_{i,j}$ be the set of clauses that contain the positive literal $x_{t+1}$ such that each clause in $P_{i,j}$ contains exactly $i$ active literals, of which exactly $j$ are positive, and let $N_{i,j}$ be the set of clauses that contain the negative literal $\overline{x_{t+1}}$ such that each clause in $N_{i,j}$ contains exactly $i$ active literals, of which exactly $j$ are positive. Note that according to the augmented Johnson's algorithm, if at this moment a clause $C_h$ has exactly $i$ active literals, then the weight value $w(C_h)$ equals exactly $1/2^i$.

**Case 1.** Suppose that the augmented Johnson's algorithm assigns $\tau(x_{t+1}) =$ TRUE. Then according to the algorithm, regardless of the value $b[t]$ we must have

$$\sum_{i=1}^{r} \sum_{j=1}^{i} w(P_{i,j}) \ge \sum_{i=1}^{r} \sum_{j=0}^{i-1} w(N_{i,j})$$

This is equivalent to

$$\sum_{i=1}^{r} \frac{\sum_{j=1}^{i} |P_{i,j}|}{2^i} \ge \sum_{i=1}^{r} \frac{\sum_{j=0}^{i-1} |N_{i,j}|}{2^i} \tag{9.3}$$

Now we have

$$\begin{aligned}
N_1^{(t+1)} &= (N_1^{(t)} - N_{1,0}) \cup N_{2,0} \\
N_2^{(t+1)} &= (N_2^{(t)} - N_{2,0}) \cup N_{3,0} \\
&\cdots \\
N_{r-1}^{(t+1)} &= (N_{r-1}^{(t)} - N_{r-1,0}) \cup N_{r,0} \\
N_r^{(t+1)} &= (N_r^{(t)} - N_{r,0})
\end{aligned}$$

This gives us

$$|N_1^{(t+1)}| + \frac{1}{2}|N_2^{(t+1)}| + \cdots + \frac{1}{2^{r-1}}|N_r^{(t+1)}|$$

$$
\begin{aligned}
&= |N_1^{(t)}| + \frac{1}{2}|N_2^{(t)}| + \cdots + \frac{1}{2^{r-1}}|N_r^{(t)}| \qquad (9.4)\\
&\qquad -|N_{1,0}| + \frac{1}{2}|N_{2,0}| + \frac{1}{2^2}|N_{3,0}| + \cdots + \frac{1}{2^{r-1}}|N_{r,0}|\\
&= \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} + \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} - 2|N_{1,0}|
\end{aligned}
$$

On the other hand, we have

$$
S^{(t+1)} = S^{(t)} \cup \bigcup_{i=1}^{r} \bigcup_{j=1}^{i} P_{i,j} \qquad \text{and} \qquad K^{(t+1)} = K^{(t)} \cup N_{1,0} \qquad (9.5)
$$

Combining relations (9.3)-(9.5), and using the inductive hypothesis, we get

$$
\begin{aligned}
|S^{(t+1)}| &= |S^{(t)}| + \sum_{i=1}^{r} \sum_{j=1}^{i} |P_{i,j}|\\
&\geq 2|K^{(t)}| + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} - A_0 + \sum_{i=1}^{r} \frac{\sum_{j=1}^{i} |P_{i,j}|}{2^{i-1}}\\
&\geq 2|K^{(t)}| + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} - A_0 + \sum_{i=1}^{r} \frac{\sum_{j=0}^{i-1} |N_{i,j}|}{2^{i-1}}\\
&\geq 2(|K^{(t)}| + |N_{1,0}|) + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} + \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} - 2|N_{1,0}| - A_0\\
&= 2|K^{(t+1)}| + \sum_{i=1}^{r} \frac{|N_i^{(t+1)}|}{2^{i-1}} - A_0
\end{aligned}
$$

Therefore, the induction goes through in this case.

**Case 2.** Suppose that the augmented Johnson's algorithm assigns $\tau(x_{t+1}) = \text{FALSE}$. The proof for this case is similar but slightly more complicated. We will concentrate on describing the differences.

According to the augmented Johnson's algorithm, we have

$$
\sum_{i=1}^{r} \frac{\sum_{j=1}^{i} |P_{i,j}|}{2^i} \leq \sum_{i=1}^{r} \frac{\sum_{j=0}^{i-1} |N_{i,j}|}{2^i} \qquad (9.6)
$$

Based on the relations

$$
\begin{aligned}
N_1^{(t+1)} &= (N_1^{(t)} - N_{1,0}) \cup P_{2,1}\\
N_2^{(t+1)} &= (N_2^{(t)} - N_{2,0}) \cup P_{3,1}
\end{aligned}
$$

$$\begin{array}{rcl} & \cdots & \\ N_{r-1}^{(t+1)} & = & (N_{r-1}^{(t)} - N_{r-1,0}) \cup P_{r,1} \\ N_r^{(t+1)} & = & (N_r^{(t)} - N_{r,0}) \end{array}$$

we get

$$\begin{array}{rl} & |N_1^{(t+1)}| + \frac{1}{2}|N_2^{(t+1)}| + \cdots + \frac{1}{2^{r-1}}|N_r^{(t+1)}| \\ = & \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} + \sum_{i=2}^{r} \frac{|P_{i,1}|}{2^{i-2}} - \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} \end{array} \tag{9.7}$$

Moreover, we have

$$S^{(t+1)} = S^{(t)} \cup \bigcup_{i=1}^{r} \bigcup_{j=0}^{i-1} N_{i,j} \qquad \text{and} \qquad K^{(t+1)} = K^{(t)} \cup P_{1,1} \tag{9.8}$$

Combining relations (9.7) and (9.8) and using the inductive hypothesis,

$$\begin{array}{rl} & 2|K^{(t+1)}| + \sum_{i=1}^{r} \frac{|N_i^{(t+1)}|}{2^{i-1}} - A_0 \\ = & 2|K^{(t)}| + 2|P_{1,1}| + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} + \sum_{i=2}^{r} \frac{|P_{i,1}|}{2^{i-2}} - \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} - A_0 \\ \leq & |S^{(t)}| + \sum_{i=1}^{r} \frac{|P_{i,1}|}{2^{i-2}} - \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} \\ = & |S^{(t)}| + \sum_{i=1}^{r} \sum_{j=0}^{i-1} |N_{i,j}| + \sum_{i=1}^{r} \frac{|P_{i,1}|}{2^{i-2}} - \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} - \sum_{i=1}^{r} \sum_{j=0}^{i-1} |N_{i,j}| \end{array}$$

Now according to equation (9.8),

$$|S^{(t+1)}| = |S^{(t)}| + \sum_{i=1}^{r} \sum_{j=0}^{i-1} |N_{i,j}|$$

Moreover, since

$$\begin{array}{rl} & \sum_{i=1}^{r} \frac{|N_{i,0}|}{2^{i-1}} + \sum_{i=1}^{r} \sum_{j=0}^{i-1} |N_{i,j}| \geq |N_{1,0}| + |N_{1,0}| + \sum_{i=2}^{r} \sum_{j=0}^{i-1} |N_{i,j}| \\ \geq & 2|N_{1,0}| + \sum_{i=2}^{r} \frac{\sum_{j=0}^{i-1} |N_{i,j}|}{2^{i-2}} = \sum_{i=1}^{r} \frac{\sum_{j=0}^{i-1} |N_{i,j}|}{2^{i-2}} \geq \sum_{i=1}^{r} \frac{\sum_{j=1}^{i} |P_{i,j}|}{2^{i-2}} \\ \geq & \sum_{i=1}^{r} \frac{|P_{i,1}|}{2^{i-2}} \end{array}$$

the third inequality above follows from relation (9.6), we conclude

$$2|K^{(t+1)}| + \sum_{i=1}^{r} \frac{|N_i^{(t+1)}|}{2^{i-1}} - A_0 \leq |S^{(t+1)}|$$

Thus, the induction also goes through in this case.

The lemma now follows directly from the inductive proof. $\square$

Now we are ready to prove our main theorem. Let us come back to the original algorithm **Johnson**.

**Theorem 9.2.5** *The approximation ratio for the algorithm* **Johnson** *given in Figure 9.5 for the* MAX-SAT *problem is* 1.5. *This bound is tight.*

PROOF. Let $F$ be an instance to the MAX-SAT problem. Let $\tau_o$ be an arbitrary optimal assignment to $F$. Now we construct another instance $F'$ for MAX-SAT, as follows. Starting with $F$, if for a boolean variable $x_t$, we have $\tau_o(x_t) = \text{FALSE}$, then we "flip" $x_t$ (i.e., interchange $x_t$ and $\overline{x_t}$) in $F$. Thus, there is a one-to-one correspondence between the set of clauses in $F$ and the set of clauses in $F'$. It is easy to see that the sets $F$ and $F'$, as instances for MAX-SAT, have the same optimal value. In particular, the assignment $\tau_o'$ on $F'$ such that $\tau_o'(x_t) = \text{TRUE}$ for all $t$ is an optimal assignment for the instance $F'$.

We let a boolean array $b[1..n]$ be such that $b[t] = \tau_o(x_t)$ for all $t$.

We show that the assignment constructed by the original algorithm **Johnson** on the instance $F$ and the assignment constructed by the augmented Johnson's algorithm on the instance $F'$ augmented by the boolean array $b[1..n]$ satisfy exactly the same set of clauses.

Inductively, suppose that for the first $(t-1)$st iterations of the **for** loop in step 3, both algorithms satisfy exactly the same set of clauses. Now consider the $t^{th}$ iteration of the algorithms.

If $x_t$ in $F$ is not flipped in $F'$, then $b[t] = \text{TRUE}$. Thus, the augmented Johnson's algorithm assigns $\tau(x_t) = \text{TRUE}$ and makes the clauses $C_1^T, \ldots, C_q^T$ satisfied during the $t^{th}$ iteration if and only if $\sum_{i=1}^q w(C_i^T) \geq \sum_{i=1}^s w(C_i^F)$, where $C_1^T$, ..., $C_q^T$ are the clauses in $L$ containing $x_t$ and $C_1^F$, ..., $C_s^F$ are the clauses in $L$ containing $\overline{x_t}$. On the other hand, if $x_t$ in $F$ is flipped in $F'$, then $b[t] = \text{FALSE}$, and the augmented Johnson's algorithm assigns $\tau(x_t) = \text{FALSE}$ and makes the clauses $C_1^F$, ..., $C_s^F$ satisfied if and only if $\sum_{i=1}^q w(C_i^T) \leq \sum_{i=1}^s w(C_i^F)$. Note that if $x_t$ is not flipped, then $\{C_1^T, \ldots, C_q^T\}$ is exactly the set of clauses containing $x_t$ in the $t^{th}$ iteration of the original algorithm **Johnson** for the instance $F$, while if $x_t$ is flipped, then $\{C_1^F, \ldots, C_s^F\}$ is exactly the set of clauses containing $x_t$ in the $t^{th}$ iteration of the original algorithm **Johnson** for the instance $F$. Therefore, in the $t^{th}$ iteration, the set of the clauses satisfied by the augmented Johnson's algorithm on $F'$ and $b[1..n]$ corresponds exactly to the set of clauses satisfied by the original algorithm **Johnson** on $F$. In conclusion, the assignment constructed by the original algorithm **Johnson** on the instance $F$ and the assignment constructed by the augmented Johnson's algorithm on the instance $F'$ and the boolean array $b[1..n]$ satisfy exactly the same set of clauses.

Therefore, we only need to analyze the approximation ratio of the augmented Johnson's algorithm on the instance $F'$ and the boolean array $b[1..n]$.

Let $K^{(t)}$, $S^{(t)}$, and $N_i^{(t)}$ be the sets defined before for the augmented Johnson's algorithm on the instance $F'$ and the boolean array $b[1..n]$. According to Lemma 9.2.4, we have

$$|S^{(t)}| \geq 2|K^{(t)}| + \sum_{i=1}^{r} \frac{|N_i^{(t)}|}{2^{i-1}} - A_0 \tag{9.9}$$

for all $0 \leq t \leq n$, where $A_0 = \sum_{i=1}^{r} |N_i^{(0)}|/2^{i-1}$.

At the end of the augmented Johnson's algorithm, i.e., $t = n$, $S^{(n)}$ is exactly the set of clauses satisfied by the assignment constructed by the algorithm, and $K^{(n)}$ is exactly the set of clauses not satisfied by the assignment. Moreover, $N_i^{(n)} = \emptyset$ for all $i \geq 1$.

According to (9.9), we have

$$|S^{(n)}| \geq 2|K^{(n)}| - A_0 \tag{9.10}$$

Note that

$$A_0 = \sum_{i=1}^{r} \frac{|N_i^{(0)}|}{2^{i-1}} \leq \sum_{i=1}^{r} |N_i^{(0)}| \tag{9.11}$$

Combining relations (9.10) and (9.11), we get

$$3|S^{(n)}| \geq 2(|S^{(n)}| + |K^{(n)}|) - \sum_{i=1}^{r} |N_i^{(0)}| \tag{9.12}$$

Since $S^{(n)} \cup K^{(n)}$ is the whole set $\{C_1, \ldots, C_m\}$ of clauses in $F'$, we have $|S^{(n)}| + |K^{(n)}| = m$. Moreover, the assignment $\tau_o'(x_t) = \text{TRUE}$ for all $1 \leq t \leq n$ is an optimal assignment to the instance $F'$, which satisfies all clauses in $F'$ except those in $N_i^{(0)}$, for $1 \leq i \leq r$. Thus, the optimal value of the instance $F'$, i.e., the number of clauses satisfied by an optimal assignment to $F'$ is equal to

$$Opt(F') = m - \sum_{i=1}^{r} |N_i^{(0)}| \tag{9.13}$$

Now combining the relations (9.13) and (9.12), we get

$$3|S^{(n)}| \geq m + Opt(F') \geq 2 \cdot Opt(F')$$

The set $S^{(n)}$ is the set of clauses satisfied by the assignment constructed by the augmented Johnson's algorithm. Since the original algorithm **Johnson** and the augmented Johnson's algorithm satisfy the same set of clauses and since $Opt(F) = Opt(F')$, we conclude that the approximation ratio of the algorithm **Johnson** for the MAX-SAT problem is $Opt(F')/|S^{(n)}| \leq 1.5$.

To see that the bound 1.5 is tight for the algorithm **Johnson**, consider the following instance $F_h$ of $3h$ clauses for MAX-SAT, where $h$ is any integer larger than 0.

$$F_h = \{(x_{3k+1} \vee x_{3k+2}), (x_{3k+1} \vee x_{3k+3}), (\overline{x_{3k+1}}) \mid 0 \leq k \leq h-1\}$$

It is easy to verify that the algorithm **Johnson** assigns $x_t$ = TRUE for all $1 \leq t \leq 3h$, and this assignment satisfies exactly $2h$ clauses in $F_h$. On the other hand, the assignment $x_{3k+1}$ = FALSE, $x_{3k+2} = x_{3k+3}$ = TRUE for all $0 \leq k \leq h-1$ obviously satisfies all $3h$ clauses in $F_h$. □

Theorem 9.2.5 shows an example in which the precise approximation ratio for a simple algorithm is difficult to derive. The next question is whether the approximation ratio 1.5 on the MAX-SAT problem can be further improved by a "better" algorithm for the problem. This has been a very active research topic in recent years. In the next chapter, we will develop new techniques that give better approximation algorithms for the MAX-SAT problem.

## 9.3 Maximum 3-dimensional matching

Let $X$, $Y$, and $Z$ be three disjoint finite sets. Given a set $S \subseteq X \times Y \times Z$ of triples, a *matching* $M$ in $S$ is a subset of $S$ such that no two triples in $M$ have the same coordinate at any dimension. The 3-DIMENSIONAL MATCHING problem is defined as follows.

3-DIMENSIONAL MATCHING (3D-MATCHING)

INPUT: a set $S \subseteq X \times Y \times Z$ of triples

OUTPUT: a matching $M$ in $S$ with the maximum number of triples

The 3D-MATCHING problem is a generalization of the classical "marriage problem": Given $n$ unmarried men and $m$ unmarried women, along with a list of all male-female pairs who would be willing to marry one another, find the largest number of pairs so that polygamy is avoided and every paired person receives an acceptable spouse. Analogously, in the 3D-MATCHING problem, the sets $X$, $Y$, and $Z$ correspond to three sexes, and each triple in $S$ corresponds to a 3-way marriage that would be acceptable to all three participants.

Note that the marriage problem is simply the 2D-MATCHING problem: given a set $S \subseteq X \times Y$ of pairs, find a maximum subset $M$ of $S$ such that no two pairs in $M$ agree in any coordinate. The 2D-MATCHING problem is actually the standard bipartite graph matching problem. In fact, the disjoint sets $X$ and $Y$ can be regarded as the vertices of a bipartite graph $G$, and each pair in the set $S$ corresponds to an edge in the graph $G$. Now a matching $M$ in $S$ is simply a subset of edges in which no two edges share a common end. That is, a matching in $S$ is a graph matching in the corresponding bipartite graph $G$. As we have studied in Chapter 3, the bipartite graph matching problem, i.e., the 2D-MATCHING problem can be solved in time $O(m\sqrt{n})$.

The decision version of the 3D-MATCHING problem is described as follows: given a set $S \subseteq X \times Y \times Z$ of triples, where each of the sets $X$, $Y$, and $Z$ has exactly $n$ elements, is there a matching in $S$ that contains $n$ triples? The decision version of the 3D-MATCHING problem is one of the six "basic" NP-complete problems [49]. It is easy to see that the decision version of the 3D-MATCHING problem can be reduced in polynomial time to the 3D-MATCHING problem: a set $S \subseteq X \times Y \times Z$ of triples, where each of the sets $X$, $Y$, and $Z$ has exactly $n$ elements, is a yes-instance for the decision version of the 3D-MATCHING problem if and only if when $S$ is regarded as an instance for the 3D-MATCHING problem, $S$ has an optimal value $n$. In consequence, the 3D-MATCHING problem is NP-hard.

We consider polynomial time approximation algorithms for the 3D-MATCHING problem.

Let $S \subseteq X \times Y \times Z$ be a set of triples and let $M$ be a matching in $S$. We say that a triple $(x, y, z)$ in $S - M$ *does not contradict the matching* $M$ if no triple in $M$ has a coordinate agreeing with $(x, y, z)$. In other words,

**Algorithm. Apx3DM-First**

Input: a set $S \subseteq X \times Y \times Z$ of triples
Output: a matching $M$ in $S$

1. $M = \phi$;
2. **for** each triple $(x, y, z)$ in $S$ **do**
   **if** $(x, y, z)$ does not contradict $M$
   **then** $M = M \cup \{(x, y, z)\}$.

Figure 9.7: First algorithm for 3D-MATCHING

$(x, y, z)$ does not contradict the matching $M$ if the set $M \cup \{(x, y, z)\}$ still forms a matching in $S$.

Our first approximation algorithm for 3D-MATCHING is given in Figure 9.7.

It is easy to verify that the algorithm **Apx3DM-First** runs in polynomial time. In fact, if we use three arrays for the elements in $X$, $Y$, and $Z$, and mark the elements as "in $M$" or "not in $M$", then in constant time we can decide whether a triple $(x, y, z)$ contradicts the matching $M$. With these data structures, the algorithm **Apx3DM-First** runs in linear time.

**Theorem 9.3.1** *The algorithm* **Apx3DM-First** *for the* 3D-MATCHING *problem has its approximation ratio bounded by* 3.

PROOF. From the algorithm **Apx3DM-First**, it is clear that the set $M$ constructed is a matching in the given set $S$.

Let $M_{\max}$ be a maximum matching in $S$. Let $(x, y, z)$ be any triple in $M_{\max}$. Either the triple $(x, y, z)$ is also in the matching $M$, or, according to the algorithm **Apx3DM-First**, the triple $(x, y, z)$ contradicts the matching $M$ (otherwise, the triple $(x, y, z)$ would have been added to $M$ by the algorithm). Therefore, in any case, at least one of the three elements $x$, $y$, and $z$ in the triple $(x, y, z)$ is in a triple in $M$. Therefore, there are at least $|M_{\max}|$ different elements in the triples in the matching $M$. Since each triple in $M$ has exactly three different elements, we conclude that the number of triples in $M$ is at least $|M_{\max}|/3$. This gives

$$Opt(S)/|M| = |M_{\max}|/|M| \leq 3$$

The theorem is proved. $\square$

It is easy to see that the bound 3 is tight for the algorithm **Apx3DM-First**. Consider the following set of $4h$ triples, where $h$ is any positive integer.

$$S_h = \{(a_i, b_i, c_i), (a_i, d_i, e_i), (g_i, b_i, h_i), (p_i, q_i, c_i) \mid i = 1, \ldots, h\}$$

If the algorithm **Apx3DM-First** first picks the triples $(a_i, b_i, c_i)$, $i = 1, \ldots, h$, then picks the other triples, then the matching $M$ constructed by the algorithm contains $h$ triples:

$$M_h = \{(a_i, b_i, c_i) \mid i = 1, \ldots, h\} \tag{9.14}$$

On the other hand, the maximum matching $M_{\max}$ in the set $S_h$ contains $3h$ triples:

$$M_{\max} = \{(a_i, d_i, e_i), (g_i, b_i, h_i), (p_i, q_i, c_i) \mid i = 1, \ldots, h\}$$

Now we describe an improved approximation algorithm for the 3D-Matching problem.

A matching $M$ in the set $S$ is *maximal* if every triple in $S - M$ contradicts $M$. In particular, the matching constructed by the algorithm **Apx3DM-First** is a maximal matching. The proof for Theorem 9.3.1 shows that the size of a maximal matching is at least $1/3$ of the size of a maximum matching in $S$.

Let $M$ be a maximal matching. By the definition, no triple can be added directly to $M$ to make a larger matching in $S$. However, it is still possible that if we remove one triple from $M$, then we are able to add more than one triple from $S - M$ to $M$ to make a larger matching. For example, the matching $M_h$ of $h$ triples in (9.14) is a maximal matching in the set $S_h$. By removing the triple $(a_1, b_1, c_1)$ from the set $M_h$, we will be able to add three triples $(a_1, d_1, e_1)$, $(g_1, b_1, h_1)$, $(p_1, q_1, c_1)$ to make a matching of $h+2$ triples in $S_h$.

We say that the matching $M$ in $S$ is 1-optimal if no such a triple in $M$ exists. More formally, we say that a matching $M$ in $S$ is *1-optimal* if $M$ is maximal and it is impossible to find a triple $(a_1, b_1, c_1)$ in $M$ and two triples $(a_2, b_2, c_2)$, and $(a_3, b_3, c_3)$ in $S - M$ such that

$$M - \{(a_1, b_1, c_1)\} \cup \{(a_2, b_2, c_2), (a_3, b_3, c_3)\}$$

is a matching in $S$.

Our second approximation algorithm for the 3D-Matching problem is based on 1-optimal matchings. See Figure 9.8.

We analyze the algorithm **Apx3DM-Second**.

**Algorithm. Apx3DM-Second**

Input: a set $S \subseteq X \times Y \times Z$ of triples
Output: a matching $M$ in $S$

1. construct a maximal matching $M$ in $S$;
2. change = TRUE;
3. **while** change **do**
   change = FALSE;
   **for** each triple $(a, b, c)$ in $M$ **do**
     $M = M - \{(a, b, c)\}$;
     let $S_r$ be the set of triples in $S$ not contradicting $M$;
     construct a maximum matching $M_r$ in $S_r$;
     **if** $M_r$ contains more than one triple
     **then** $M = M \cup M_r$; change = TRUE;
     **else** $M = M \cup \{(a, b, c)\}$.

Figure 9.8: Second algorithm for 3D-MATCHING

**Lemma 9.3.2** *After each execution of the* **for** *loop in step 3 of the algorithm* **Apx3DM-Second**, *the matching $M$ is a maximal matching.*

PROOF. Before the algorithm enters step 3, the matching $M$ is maximal.

Since the set $S_r$ has no common element with the matching $M$ after the triple $(a, b, c)$ is removed from $M$, for any matching $M'$ in $S_r$, $M \cup M'$ is a matching in $S$. Moreover, since all triples in $S - S_r$ contradict $M$, and all triples in $S_r - M_r$ contradict $M_r$, we conclude that all triples in $S - (M \cup M_r)$ contradict $M \cup M_r$. That is, the matching $M \cup M_r$ is a maximal matching in $S$, which is assigned to $M$ if $M_r$ has more than one triple. In case $M_r$ has only one triple, the triple $(a, b, c)$ is put back to $M$, which by induction is also maximal. $\square$

**Lemma 9.3.3** *The matching constructed by the algorithm* **Apx3DM-Second** *is 1-optimal.*

PROOF. It is easy to see that there are a triple $(a_1, b_1, c_1)$ in $M$ and two triples $(a_2, b_2, c_2)$ and $(a_3, b_3, c_3)$ in $S - M$ such that

$$M - \{(a_1, b_1, c_1)\} \cup \{(a_2, b_2, c_2), (a_3, b_3, c_3)\}$$

is a matching in $S$ if and only if the matching $M_r$ in $S_r$ contains more than one triple. Therefore, the algorithm **Apx3DM-Second** actually goes through all triples in $M$ and checks whether each of them can be traded for more than one triple in $S - M$. The algorithm stops when it finds out no such trading is possible. In other words, the algorithm **Apx3DM-Second** ends up with a 1-optimal matching $M$. $\square$

**Theorem 9.3.4** *The algorithm* **Apx3DM-Second** *runs in polynomial time.*

PROOF. Suppose that the input instance $S$ contains $n$ triples.

As explained before, the algorithm **Apx3DM-First** constructs a maximal matching. Therefore, the maximal matching in step 1 of the algorithm **Apx3DM-Second** can be constructed in linear time.

We first show that in each execution of the **for** loop in step 3 of the algorithm **Apx3DM-Second**, the maximum matching $M_r$ in the set $S_r$ contains at most 3 triples. Assume the contrary and let $(a_1, b_1, c_1)$, $(a_2, b_2, c_2)$, $(a_3, b_3, c_3)$, and $(a_4, b_4, c_4)$ be four triples in the maximum matching $M_r$ in $S_r$. Then at least one of them, say $(a_1, b_1, c_1)$, contains no element in the triple $(a, b, c)$ removed from $M$. Since $(a_1, b_1, c_1)$ does not contradict $M - \{(a, b, c)\}$, $(a_1, b_1, c_1)$ does not contradict $M$ even before the triple $(a, b, c)$ is removed from $M$. Therefore, before the triple $(a, b, c)$ is removed, the matching $M$ is not maximal. This contradicts Lemma 9.3.2.

Therefore, a maximum matching in $S_r$ contains at most 3 triples. Thus, to construct the maximum matching $M_r$ in $S_r$, we can try all groups of three triples and all groups of two triples in $S_r$. There are only $O(n^3)$ such groups in the set $S_r$ (note $S_r$ is a subset of $S$ so it contains at most $n$ triples). Therefore, in time $O(n^3)$, we can construct the maximum matching $M_r$ for the set $S_r$. In consequence, the **for** loop in step 3 takes time $O(n^4)$.

Since each execution of the **while** loop body increases the number of triples in the matching $M$ by at least 1, and a maximum matching in the set $S$ has at most $n$ triples, we conclude that the algorithm **Apx3DM-Second** has its running time bounded by $O(n^5)$. $\square$

**Remark.** In fact, the maximum matching $M_r$ in the set $S_r$ can be constructed in linear time. This will reduce the total running time of the algorithm **Apx3DM-Second** to $O(n^3)$. We leave this improvement to the reader.

We finally consider the approximation ratio for the algorithm **Apx3DM-Second**.

**Theorem 9.3.5** *The algorithm* **Apx3DM-Second** *has its approximation ratio bounded by* 2.

PROOF. We denote by $M$ the matching in $S$ constructed by the algorithm **Apx3DM-Second** and let $M_{\max}$ be a maximum matching in $S$. We say that an element $a \in X \cup Y \cup Z$ is in a matching if $a$ is in a triple in the matching. Since the sets $X$, $Y$, and $Z$ are disjoint, this introduces no ambiguity.

Based on the matchings $M$ and $M_{\max}$, we introduce a weight function $w(\cdot)$ on elements in $X \cup Y \cup Z$ as follows.

- if an element $a$ is not in both $M$ and $M_{\max}$, then $w(a) = 0$;
- if an element $a$ is in both $M$ and $M_{\max}$, and $a$ is in a triple of $M_{\max}$ that contains only one element in $M$, then $w(a) = 1$;
- if an element $a$ is in both $M$ and $M_{\max}$, and $a$ is in a triple of $M_{\max}$ that contains exactly two elements in $M$, then $w(a) = 1/2$;
- if an element $a$ is in both $M$ and $M_{\max}$, and $a$ is in a triple of $M_{\max}$ that contains three elements in $M$, then $w(a) = 1/3$;

The *weight* $w(t)$ of a triple $t = (a, b, c)$ is the sum of the weights of its elements: $w(t) = w(a) + w(b) + w(c)$. According to the definition of the weight function, each triple in the matching $M_{\max}$ has weight exactly 1.

Let $t = (a, b, c)$ be a triple in $M$. If $w(t) > 2$, then at least two elements in $t$ have weight 1. Without loss of generality, suppose that $w(a) = w(b) = 1$. By the definition, there are two triples $t_1 = (a, b_1, c_1)$ and $t_2 = (a_2, b, c_2)$ in the matching $M_{\max}$ such that the elements $b_1$, $c_1$, $a_2$, $c_2$ are not in the matching $M$. However, this would imply that

$$M - \{(a, b, c)\} \cup \{(a, b_1, c_1), (a_2, b, c_2)\}$$

is a matching in $S$, so that the matching $M$ constructed by the algorithm **Apx3DM-Second** would not be 1-optimal. This contradicts Lemma 9.3.3.

Thus, each triple in the matching $M$ has weight at most 2. Since only elements in both matchings $M$ and $M_{\max}$ have nonzero weight, we have

$$\sum_{t \in M_{\max}} w(t) = \sum_{t \in M} w(t)$$

Since each triple in $M_{\max}$ has weight 1, we have $\sum_{t \in M_{\max}} w(t) = |M_{\max}|$. Moreover, since each triple in $M$ has weight at most 2, we have

$$\sum_{t \in M} w(t) \leq 2|M|$$

This gives us

$$|M_{\max}| \leq 2|M|$$

or $|M_{\max}|/|M| \leq 2$. This completes the proof. $\square$

To see that the bound 2 is tight for the algorithm **Apx3DM-Second**, consider the following instance

$$\begin{aligned} S &= \{(a_1, b_1, c_1), (a_2, b_2, c_2), (a_1, b_3, c_3), (a_2, b_1, c_4), (a_5, b_2, c_5), \\ &\quad (a_6, b_2, c_1), (a_1, b_7, c_2), (a_2, b_8, c_1), (a_9, b_1, c_2)\} \end{aligned}$$

If the triples are given in this order, then step 1 of the algorithm **Apx3DM-Second**, if it calls **Apx3DM-First** as a subroutine, will return the maximal matching

$$M = \{(a_1, b_1, c_1), (a_2, b_2, c_2)\} \tag{9.15}$$

It is not hard to see that the matching $M$ given in (9.15) is already 1-optimal. Thus, the algorithm **Apx3DM-Second** returns $M$ of 2 triples as the approximation matching. On the other hand, the maximum matching

$$M_{\max} = \{(a_1, b_3, c_3), (a_5, b_2, c_5), (a_2, b_8, c_1), (a_9, b_1, c_2)\}$$

in the set $S$ contains 4 triples.

A natural extension of the algorithm **Apx3DM-Second** is to consider 2-optimal, or in general $k$-optimal matchings. That is, we construct a maximal matching $M$ in $S$ such that no $k$ triples in $M$ can be traded for $k+1$ triples in $S-M$. It is not very hard to see that for any fixed integer $k \geq 1$, a $k$-optimal matching in $S$ can be constructed in polynomial time. We can show that a $k$-optimal matching gives an approximation ratio smaller than 2 for $k > 1$. For example, a 2-optimal matching has approximation ratio 9/5 while a 3-optimal matching has approximation ratio 5/3. In general, for any given constant $\epsilon > 0$, we can choose a proper integer $k \geq 1$ such that the ratio of a maximum matching and a $k$-optimal matching is bounded by $1.5 + \epsilon$. Therefore, for any $\epsilon > 0$, there is a polynomial time approximation algorithm for the 3D-MATCHING problem whose approximation ratio is bounded by $1.5 + \epsilon$. Interested readers are referred to [18, 69]. It is unknown whether there is a polynomial time approximation algorithm for the 3D-MATCHING problem whose approximation ratio is smaller than 1.5.

Two generalizations of the 3D-MATCHING problems are the $k$-DIMENSIONAL MATCHING problem and the $k$-SET PACKING problem:

k-DIMENSIONAL MATCHING

INPUT: a set $S \subseteq X_1 \times X_2 \times \cdots \times X_k$ of $k$-tuples

OUTPUT: a maximum subset $M$ of $S$ in which no two $k$-tuples agree on any coordinate.

k-SET PACKING

INPUT: a collection $T$ of sets $S_1$, $S_2$, ..., $S_n$, where each set $S_i$ contains at most $k$ elements

OUTPUT: a maximum subcollection $P$ of disjoint sets in $T$

Approximation algorithms for the $k$-DIMENSIONAL MATCHING problem and the $k$-SET PACKING problem have been studied based on the techniques presented in this section [69].

## 9.4 Minimum vertex cover

The decision version of the vertex cover problem is one of the six "basic" NP-complete problems [49]. The optimization version of the vertex cover problem has been a central problem in the study of approximation algorithms.

Let $G$ be an undirected graph. A *vertex cover* of $G$ is a set $C$ of vertices in $G$ such that every edge in $G$ has at least one endpoint in $C$ (thus, the set $C$ "covers" the edges of $G$). Vertex covers of a graph are related to independent sets of the graph by the following lemma.

**Lemma 9.4.1** *A set $C$ of vertices in a graph $G = (V, E)$ is a vertex cover of $G$ if and only if the set $V - C$ is an independent set in $G$.*

PROOF. Suppose $C$ is a vertex cover. Since every edge in $G$ has at least one endpoint in $C$, no two vertices in $V - C$ are adjacent. That is, $V - C$ is an independent set.

Conversely, if $V - C$ is an independent set, then every edge in $G$ has at least one endpoint not in $V - C$. Therefore, every edge in $G$ has at least one endpoint in $C$ and $C$ forms a vertex cover. □

The minimum vertex cover problem is formally defined as follows.

MIN VERTEX COVER

$I_Q$: the set of all undirected graphs

**Algorithm. VC-Apx-I**

Input: a graph $G$
Output: a vertex cover $C$ of $G$

1. $C = \emptyset$;
2. **for** each edge $e$ in $G$ **do**
   **if** no endpoint of $e$ is in $C$
   **then** add both endpoints of $e$ to $C$;
3. **return** $C$.

Figure 9.9: Approximating vertex cover I

$S_Q$: $S_Q(G)$ is the set of all vertex covers of the graph $G$

$f_Q$: $f_Q(G, C)$ is the size of the vertex cover $C$ of $G$.

$opt_Q$: min

### 9.4.1 Vertex cover and matching

Recall that a *matching* in a graph $G$ is a set $M$ of edges such that no two edges in $M$ share a common endpoint. A vertex is *matched* if it is an endpoint of an edge in $M$ and *unmatched* otherwise.

The problems Graph Matching and Min Vertex Cover are closely related. We first present a simple approximation algorithm for Min Vertex Cover based on matching.

**Lemma 9.4.2** *Let $M$ be a matching in a graph $G$ and let $C$ be a vertex cover of $G$, then $|M| \leq |C|$. In particular, the size of a minimum vertex cover of $G$ is at least as large as the size of a maximum matching in $G$.*

Proof. Since the vertex cover $C$ must cover all edges in $G$, each edge in the matching $M$ has at least one endpoint in $C$. Since no two edges in $M$ share a common endpoint, we conclude that the number $|C|$ of vertices in the vertex cover $C$ is at least as large as the number $|M|$ of edges in the matching $M$. □

A matching $M$ in a graph $G$ is *maximal* if there is no edge $e$ in $G$ such that $e \notin M$ and $M \cup \{e\}$ still forms a matching in $G$. Our first approximation algorithm for Min Vertex Cover, based on maximal matchings, is given in Figure 9.9.

**Theorem 9.4.3** *The algorithm* **VC-Apx-I** *is a linear time approximation algorithm with approximation ratio* 2 *for* MIN VERTEX COVER.

PROOF. The algorithm obviously runs in linear time.

Because of the **for** loop in step 2 of the algorithm, every edge in $G$ has at least one endpoint in the set $C$. Therefore, $C$ is a vertex cover of the graph $G$.

The **for** loop in step 2 of the algorithm implicitly constructs a maximal matching $M$, as follows. Suppose we also initialize $M = \emptyset$ in step 1, and in step 2 whenever we encounter an edge $e$ that has no endpoint in $C$, we, in addition to adding both endpoints of $e$ to $C$, also add the edge $e$ to $M$. It is straightforward to see that the set $M$ constructed this way will be a maximal matching and $C$ is the set of endpoints of the edges in $M$. Thus, $2|M| = |C|$. Now by Lemma 9.4.2, we have (where $Opt(G)$ is the size of a minimum vertex cover of $G$)

$$\frac{|C|}{Opt(G)} = \frac{2|M|}{Opt(G)} \leq \frac{2 \cdot Opt(G)}{Opt(G)} = 2$$

Thus, the approximation ratio of the algorithm is bounded by 2. □

GRAPH MATCHING and MIN VERTEX COVER are actually dual problems in their formulations by integer linear programming. To see this, let $G$ be a graph of $n$ vertices $v_1, \ldots, v_n$ and $m$ edges $e_1, \ldots, e_m$. Introduce $n$ integral variables $x_1, \ldots, x_n$ to record the membership of the vertices in $G$ in a vertex cover such that $x_i > 0$ if and only if the vertex $v_i$ is in the vertex cover. Then the instance $G$ of MIN VERTEX COVER can be formulated as an instance $Q_G$ of the INTEGER LP problem:

<u>Primal Instance $Q_G$</u>

minimize $\quad x_1 + \cdots + x_n$

subject to

$$x_{i_1} + x_{i_2} \geq 1 \qquad \text{for } i = 1, 2, \ldots, m$$

$\{$suppose the two endpoints of the edge $e_i$ are $v_{i_1}$ and $v_{i_2}\}$

$$x_j \text{ are integers and } x_j \geq 0 \qquad \text{for } j = 1, 2, \ldots, n$$

The formal dual problem of this instance for the INTEGER LP problem is (see Section 4.3 for more details):

<u>Dual Instance $Q'_G$</u>

maximize $\qquad y_1 + \cdots + y_m$

subject to

$$y_{j_1} + y_{j_2} + \cdots + y_{j_{h_j}} \leq 1 \qquad \text{for } j = 1, 2, \ldots, n$$

$$\{\text{suppose the vertex } v_j \text{ is incident on the edges } e_{j_1}, e_{j_2}, \ldots, e_{j_{h_j}}\}$$

$$y_j \text{ are integers and } y_i \geq 0 \qquad \text{for } i = 1, 2, \ldots, m$$

If we define a set $M$ of edges in $G$ based on the dual instance $G'_Q$ such that $y_i > 0$ if and only if the edge $e_i$ in the graph $G$ is in $M$, then the condition $y_{j_1} + \cdots + y_{j_{h_j}} \leq 1$ for $j = 1, \ldots, n$ requires that each vertex $v_j$ in $G$ be incident to at most one edge in $M$, or equivalently, that the set $M$ forms a matching. Therefore, the dual instance $Q'_G$ in the Integer LP problem exactly characterizes the instance $G$ for Graph Matching.

Combining this observation with Lemma 4.3.1 gives us an alternative proof for Lemma 9.4.2.

### 9.4.2 Vertex cover on bipartite graphs

Lemma 9.4.2 indicates that the size of a maximum matching of a graph $G$ is not larger than the size of a minimum vertex cover of the graph. This provides an effective lower bound for the minimum vertex cover of a graph. Since Graph Matching can be solved in polynomial time while Min Vertex Cover is NP-hard, one should not expect that in general these two values are equal. However, for certain important graph classes, the equality does hold, which induces polynomial time (precise) algorithms for Min Vertex Cover on the graph classes. In this subsection, we use this idea to develop a polynomial time (precise) algorithm for Min Vertex Cover on the class of bipartite graphs. The algorithm will turn out to be very useful in the study of approximation algorithms for Min Vertex Cover on general graphs.

Let $M$ be a matching in a graph $G$. We say that a path $p = \{u_0, u_1, \ldots\}$ is an *alternating path* (with respect to $M$) if vertex $u_0$ is unmatched, the edges $[u_{2i-1}, u_{2i}]$ are in $M$, and the edges $[u_{2i}, u_{2i+1}]$ are not in $M$, for $i = 1, 2, \ldots$. Note that an alternating path of odd length in which the last vertex is also unmatched is an *augmenting path* defined in Section 3.1. By Theorem 3.1.1, the matching $M$ is maximum if and only if there is no augmenting path with respect to $M$.

We say that a vertex $u$ is *$M$-reachable* from an unmatched vertex $u_0$ if there is an alternating path starting at $u_0$ and ending at $u$. For a set $U$ of unmatched vertices, we say that a vertex $u$ is $M$-reachable from $U$ if $u$ is

**Algorithm. VC-BipGraph**$(G, M)$

Input: bipartite graph $G = (V_1 \cup V_2, E)$ and maximum matching $M$ in $G$
Output: a minimum vertex cover $C$ of $G$

1. let $U_1$ be the set of unmatched vertices in $V_1$;
2. let $N_1$ be the set of vertices in $V_1$ that are not $M$-reachable from $U_1$;
3. let $R_2$ be the set of vertices in $V_2$ that are $M$-reachable from $U_1$;
4. output $C = N_1 \cup R_2$.

Figure 9.10: Constructing a minimum vertex cover in a bipartite graph

$M$-reachable from a vertex in $U$.

Let $G = (V_1 \cup V_2, E)$ be a bipartite graph, where every edge in $G$ has one endpoint in $V_1$ and one endpoint in $V_2$. Let $M$ be a maximum matching in $G$. Consider the algorithm given in Figure 9.10. The algorithm **VC-BipGraph** produces a set of vertices for the bipartite graph $G$, which we will prove is a minimum vertex cover of the graph $G$.

**Lemma 9.4.4** *The algorithm* **VC-BipGraph** *runs in linear time and constructs a minimum vertex cover $C$ for the bipartite graph $G$. In particular, we have $|C| = |M|$.*

PROOF. The set $R$ of $M$-reachable vertices from $U_1$ can be constructed in linear time using the algorithm **Bipartite Augment** given in Section 3.1 (see Figure 3.3). Basically, we perform a searching procedure similar to Breadth First Search, starting from each vertex in the set $U_1$. Note that in this situation, the algorithm **Bipartite Augment** never stops at step 3 since according to Theorem 3.1.1, there is no augmenting path with respect to the maximum matching $M$. Once the set $R$ is available, the set $C$ is easily obtained.

Every vertex in the set $N_1$ is matched because every unmatched vertex in $V_1$ is in the set $U_1$, which is obviously $M$-reachable from $U_1$.

Now consider the set $R_2$ of vertices in $V_2$ that are $M$-reachable from $U_1$. We claim that all vertices in $R_2$ are matched. In fact, if $v_2 \in R_2$ is unmatched and $p$ is an alternating path starting from an unmatched vertex $v_1$ in $U_1$ and ending at $v_2$, then, since the graph $G$ is bipartite, the path $p$ is of odd length. Therefore, the path $p$ would be an augmenting path with respect to the maximum matching $M$. This contradicts Theorem 3.1.1.

Let $v_1 \in N_1$ and $[v_1, v_2]$ is an edge in the matching $M$. We claim $v_2 \notin R_2$. In fact, if $v_2$ is in $R_2$ then the alternating path from a vertex $u_1$ in $U_1$ to

$v_2$ plus the edge $[v_2, v_1]$ in $M$ would form an alternating path from $u_1$ to $v_1$. This would imply that $v_1$ is $M$-reachable from $U_1$, contradicting the definition of the set $N_1$.

Therefore, each edge in the matching $M$ has at most one endpoint in the set $C = N_1 \cup R_2$ and all vertices in $C$ are matched. Consequently, $|C| \leq |M|$.

Now we prove that $C$ is a vertex cover of the graph $G$. According to the above discussion, the set $V_1$ of vertices can be partitioned into three disjoint parts: the set $U_1$ of unmatched vertices, the set $R_1$ of matched vertices $M$-reachable from $U_1$, and the set $N_1$ of matched vertices not $M$-reachable from $U_1$. Let $e = [v_1, v_2]$ be any edge in $G$, where $v_1 \in V_1$ and $v_2 \in V_2$.

If $v_1 \notin N_1$, then $v_1 \in U_1$ or $v_1 \in R_1$. In case $v_1 \in U_1$ then the edge $e$ is not in $M$. Thus, $[v_1, v_2]$ is an alternating path and $v_2 \in R_2$. On the other hand, suppose $v_1 \in R_1$. Let $p = \{u_0, \ldots, v_1\}$ be an alternating path from $u_0 \in U_1$ to $v_1$. Since $v_1$ is in the set $V_1$, by the bipartiteness of the graph $G$, $p$ is of even length. Therefore, either the vertex $v_2$ is contained in the path $p$ or the path $p$ plus the edge $[v_1, v_2]$ forms an alternating path from $u_0$ to $v_2$. In either case, $v_2 \in R_2$. This proves that for any edge $e = [v_1, v_2]$ in the graph $G$, either $v_1 \in N_1$ or $v_2 \in R_2$. In conclusion, $C = N_1 \cup R_2$ is a vertex cover of $G$.

Combining the inequality $|C| \leq |M|$ and Lemma 9.4.2, we conclude that $|C| = |M|$ and $C$ is a minimum vertex cover of $G$. $\square$

**Theorem 9.4.5** Min Vertex Cover *on bipartite graphs can be solved in time $O(\sqrt{n}m)$.*

Proof. By Corollary 3.3.8, a maximum matching of a bipartite graph $G$ can be constructed in time $O(\sqrt{n}m)$. The theorem follows from Lemma 9.4.4. $\square$

### 9.4.3 Local approximation and local optimization

By Theorem 9.4.3, the simple approximation algorithm **VC-Apx-I** given in Figure 9.9 for Min Vertex Cover has approximation ratio 2. One may expect that the ratio can be easily improved using more sophisticated techniques. However, despite long time efforts, no significant progress has been made and asymptotically, the ratio 2 still stands as the best approximation ratio for polynomial time approximation algorithms for the problem. In this subsection, we introduce several techniques that lead to slight improvements on the approximation ratio for Min Vertex Cover. The techniques can also be extended to approximation algorithms with the same ratio for the weighted version of Min Vertex Cover, in which each vertex has an assigned weight and we are looking for a vertex cover of the minimum weight.

The first technique has been called the "local optimization" in the literature, developed by Nemhauser and Trotter [99], which turns out to be very useful in the study of approximation algorithms for Min Vertex Cover, both for weighted and unweighted versions.

For a subset $V'$ of vertices in a graph $G$, denote by $G(V')$ the subgraph of $G$ induced by the vertex set $V'$, that is, $G(V')$ has $V'$ as its vertex set and contains all edges in $G$ that have their both endpoints in $V'$.

**Theorem 9.4.6** *There is an $O(\sqrt{n}m)$ time algorithm that, given a graph $G$, constructs two disjoint subsets $C_0$ and $V_0$ of the vertices in $G$ such that*

(1) *the set $C_0$ plus any vertex cover of $G(V_0)$ forms a vertex cover of $G$;*
(2) *there is a minimum vertex cover $C_{\min}$ of $G$ such that $C_0 \subseteq C_{\min}$;*
(3) *$Opt(G(V_0)) \geq |V_0|/2$.*

Proof. Let $\{v_1, \ldots, v_n\}$ be the set of vertices in the graph $G$. Construct a bipartite graph $B$ of $2n$ vertices: $v_1^L$, $v_1^R$, …, $v_n^L$, $v_n^R$ such that there is an edge $[v_i^L, v_j^R]$ in $B$ if and only if $[v_i, v_j]$ is an edge in $G$.

Let $C_B$ be a minimum vertex cover of the bipartite graph $B$. Define two disjoint subsets of vertices in the graph $G$:

$$C_0 = \{v_i \mid \text{both } v_i^L \text{ and } v_i^R \text{ are in } C_B\}$$
$$V_0 = \{v_j \mid \text{exactly one of } v_j^L \text{ and } v_j^R \text{ is in } C_B\}$$

According to Theorem 9.4.5, the minimum vertex cover $C_B$ of the bipartite graph $B$ can be constructed in time $O(\sqrt{n}m)$. Therefore, in order to prove the theorem, it suffices to prove that the constructed subsets $C_0$ and $V_0$ satisfy the conclusions in the theorem.

Let $I_0 = \{v_1, \ldots, v_n\} - (C_0 \cup V_0)$, then $I_0$ is the set of vertices $v_i$ in $G$ such that both $v_i^L$ and $v_i^R$ are not in $C_B$. For each edge $[v_i, v_j]$ in $G$, by the definition, $[v_i^L, v_j^R]$ and $[v_j^L, v_i^R]$ are edges in the bipartite graph $B$. Therefore, $v_i \in I_0$ implies $v_j \in C_0$, and $v_i \in V_0$ implies $v_j \notin I_0$.

**Proof for (1).** Let $C_V$ be a vertex cover of the induced graph $G(V_0)$. For any edge $[v_i, v_j]$ in $G$, if neither of $v_i$ and $v_j$ is in $C_V$, then one of them must be in $C_0 \cup I_0$ (otherwise, $[v_i, v_j]$ is an edge in $G(V_0)$ that should be covered by $C_V$). Without loss of generality, let $v_i \in C_0 \cup I_0$. If $v_i \in I_0$ then $v_j \in C_0$. Therefore, if the edge $[v_i, v_j]$ is not covered by $C_V$, then it must be covered by $C_0$. This proves that $C_0 \cup C_V$ is a vertex cover of $G$.

**Proof for (2).** Let $C$ be a minimum vertex cover of the graph $G$. We show the set $C_{\min} = C_0 \cup (C \cap V_0)$ is also a minimum vertex cover of $G$.

For any edge $[v_i, v_j]$ in the graph $G$, if $v_i \notin C_{\min}$, then $v_i \in I_0$ or $v_i \in V_0 - C$. If $v_i \in I_0$ then $v_j \in C_0$. If $v_i \in V_0 - C$ then $v_j \notin I_0$. So either $v_j \in C_0$ or $v_j \in V_0$. Moreover, $v_i \notin C$ implies $v_j \in C$. Thus, $v_j$ must be in $C_0 \cup (V_0 \cap C)$. Combining these, we conclude that the set $C_{\min} = C_0 \cup (C \cap V_0)$ covers the edge $[v_i, v_j]$. This proves that $C_{\min}$ is a vertex cover of $G$.

Now we show $|C_{\min}| = |C|$. For this we first construct a vertex cover for the bipartite graph $B$. Define

$$T = C_0 \cup V_0 \cup (C \cap I_0) \qquad W = C \cap C_0$$

Also define two subsets of vertices in the bipartite graph $B$:

$$L_T = \{v_i^L \mid v_i \in T\} \qquad R_W = \{v_j^R \mid v_j \in W\}$$

We prove $C'_B = L_T \cup R_W$ is a vertex cover of the bipartite grpah $B$.

Let $[v_i^L, v_j^R]$ be an edge in $B$. By the definition, $[v_i, v_j]$ is an edge in $G$. If $v_i^L \notin L_T$, then $v_i \notin T = C_0 \cup V_0 \cup (C \cap I_0)$, so $v_i \in I_0 - C$, that is, $v_i \in I_0$ and $v_i \notin C$. Since $C$ must cover $[v_i, v_j]$, we have $v_j \in C$. From $v_i \in I_0$, we have $v_j \in C_0$. Therefore, in case $v_i^L \notin L_T$, we have $v_j \in C \cap C_0 = W$, which implies $v_j^R \in R_W$. Thus, $C'_B$ is a vertex cover of $B$. Now

$$|V_0| + 2|C_0| = |C_B| \leq |C'_B| = |L_T| + |R_W| = |C_0| + |V_0| + |C \cap I_0| + |C \cap C_0|$$

The inequality is because $C'_B$ is a vertex cover while $C_B$ is a minimum vertex cover of the bipartite graph $B$. From this we get immediately

$$|C_0| \leq |C \cap I_0| + |C \cap C_0| = |C \cap (I_0 \cup C_0)|$$

Therefore,

$$\begin{aligned} |C_{\min}| &= |C_0 \cup (C \cap V_0)| = |C_0| + |C \cap V_0| \\ &\leq |C \cap (I_0 \cup C_0)| + |C \cap V_0| = |C \cap (I_0 \cup C_0 \cap V_0)| = |C| \end{aligned}$$

Since $C_{\min}$ is a vertex cover and $C$ is a minimum vertex cover of the graph $G$, we must have $|C_{\min}| = |C|$ and $C_{\min}$ is a also a minimum vertex cover of the graph $G$. Since $C_0 \subseteq C_{\min}$, statement (2) in the theorem is proved.

**Proof for (3).** Let $C_1$ be a minimum vertex cover of the induced graph $G(V_0)$. Then by statement (1) of the theorem, $C_2 = C_0 \cup C_1$ is a vertex cover of the graph $G$. Now if we let $L_2 = \{v_i^L | v_i \in C_2\}$ and $R_2 = \{v_i^R | v_i \in C_2\}$, then clearly $L_2 \cup R_2$ is a vertex cover of the bipartite graph $B$. Therefore

$$|V_0| + 2|C_0| = |C_B| \leq |L_2 \cup R_2| = 2|C_2| = 2|C_0| + 2|C_1|$$

The inequality is because $C_B$ is a minimum vertex cover while $L_2 \cup R_2$ is a vertex cover of the bipartite graph $B$. This derivation gives immediately, $|V_0| \leq 2|C_1| = 2Opt(G(V_0))$. Statement (3) of the theorem follows. $\square$

**Corollary 9.4.7** *Let $G$ be a graph, and $C_0$ and $V_0$ be the subsets given in Theorem 9.4.6. For any vertex cover $C_V$ of the induced graph $G(V_0)$, $C_0 \cup C_V$ is a vertex cover of $G$ and*

$$\frac{|C_0 \cup C_V|}{Opt(G)} \leq \frac{|C_V|}{Opt(G(V_0))}$$

PROOF. The fact that $C_0 \cup C_V$ is a vertex cover of $G$ is given by statement (1) in Theorem 9.4.6.

By statement (2) of Theorem 9.4.6, there is a minimum vertex cover $C_{\min}$ of $G$ such that $C_0 \subseteq C_{\min}$. Let $C^-_{\min} = C_{\min} - C_0$. Then $C^-_{\min}$ covers all edges in the induced graph $G(V_0)$. In fact, $C^-_{\min}$ is a minimum vertex cover of the induced graph $G(V_0)$. This can be seen as follows. We first show that $C^-_{\min}$ is a subset of $V_0$. If $C^-_{\min}$ is not a subset of $V_0$, then the smaller set $C^-_{\min} \cap V_0$ is a vertex cover of $G(V_0)$. By statement (1) of Theorem 9.4.6, $(C^-_{\min} \cap V_0) \cup C_0$ is a vertex cover of $G$. Now $|(C^-_{\min} \cap V_0) \cup C_0| < |C^-_{\min} \cup C_0| = |C_{\min}|$ contradicts the definition of $C_{\min}$. This shows that $C^-_{\min}$ is a subset of $V_0$ thus $C^-_{\min}$ is a vertex cover of $G(V_0)$. $C^-_{\min}$ is also a minimum vertex cover of $G(V_0)$ since any smaller vertex cover of $G(V_0)$ plus $C_0$ would form a vertex cover of $G$ smaller than the minimum vertex cover $C_{\min} = C^-_{\min} \cup C_0$. Therefore

$$\frac{|C_0 \cup C_V|}{Opt(G)} = \frac{|C_0| + |C_V|}{|C_{\min}|} = \frac{|C_0| + |C_V|}{|C_0| + |C^-_{\min}|} \leq \frac{|C_V|}{|C^-_{\min}|} = \frac{|C_V|}{Opt(G(V_0))}$$

The inequality has used the fact $C^-_{\min}$ is a minimum vertex cover of $G(V_0)$ so $|C^-_{\min}| \leq |C_V|$. $\square$

Corollary 9.4.7 indicates that in order to improve the approximation ratio for MIN VERTEX COVER on the graph $G$, we only need to concentrate on the induced graph $G(V_0)$. Note that approximation ratio 2 is trivial for the induced graph $G(V_0)$: by statement (3) in Theorem 9.4.6, $|V_0|/Opt(G(V_0)) \leq 2$. Therefore, simply including all vertices in the graph $G(V_0)$ gives a vertex cover of size at most twice of $Opt(G(V_0))$.

By Lemma 9.4.1, the complement of a vertex cover is an independent set, the above observation suggests that in order to improve the approximation ratio for MIN VERTEX COVER, we can try to identify a large independent set in $G(V_0)$. Our first improvement is given in Figure 9.11.

**Theorem 9.4.8** *The algorithm* **VC-Apx-II** *for* MIN VERTEX COVER *runs in time $O(\sqrt{n}m)$ and has approximation ratio $2 - 2/(\Delta + 1)$, where $\Delta$ is the largest vertex degree in the given graph.*

> **Algorithm. VC-Apx-II**
>
> Input: a graph $G$
> Output: a vertex cover $C$ of $G$
>
> 1. apply Theorem 9.4.6 to construct the subsets $C_0$ and $V_0$;
> 2. $G_1 = G(V_0)$; $I = \emptyset$;
> 3. **while** $G_1$ is not empty **do**
>    pick any vertex $v$ in $G_1$;
>    $I = I \cup \{v\}$;
>    delete $v$ and all its neighbors from the graph $G_1$;
> 4. **return** $C = (V_0 - I) \cup C_0$.

Figure 9.11: Approximating vertex cover II

PROOF. The running time of the algorithm **VC-Apx-II** is dominated by step 1, which by Theorem 9.4.6 takes time $O(\sqrt{n}m)$.

Consider the loop in step 3. The constructed set $I$ is obviously an independent set in the graph $G(V_0)$. According to the algorithm, for each group of at most $\Delta + 1$ vertices in $G(V_0)$, we conclude a new vertex in $I$. Thus, the number of vertices in $I$ is at least $|V_0|/(\Delta + 1)$. Therefore, $V_0 - I$ is a vertex cover of $G(V_0)$ and $|V_0 - I| \leq (|V_0|\Delta)/(\Delta + 1)$. Now

$$\frac{|V_0 - I|}{Opt(G(V_0))} \leq \frac{(|V_0|\Delta)/(\Delta + 1)}{|V_0|/2} = 2 - \frac{2}{\Delta + 1}$$

where we have used the fact $Opt(G(V_0)) \geq |V_0|/2$ proved in Theorem 9.4.6. Now the theorem follows directly from Corollary 9.4.7. □

**Remark.** The value $\Delta + 1$ is the approximation ratio in Theorem 9.4.8 can be replaced by $\Delta$. In fact, since every proper subgraph of the input graph has a vertex of degree strictly smaller than $\Delta$, in step 3 of the algorithm **VC-Apx-II**, we can, except possibly for the first vertex, always pick a vertex of degree smaller than $\Delta$. Thus, in this case, we will include a vertex in $I$ from each group of at most $\Delta$ vertices. We leave the details to the readers.

For graphs of low degree, the approximation ratio of the algorithm **VC-Apx-II** is significantly better than 2. However, the value $\Delta$ can be as large as $n - 1$. Therefore, in the worst case, what we can conclude is only that the algorithm **VC-Apx-II** has an approximation ratio bounded by $2 - 2/n$.

We seek further improvement by looking for larger independent sets. We first show that for graphs with no short odd cycles, finding a larger

**Algorithm.** **Large-IS**$(G, k)$

Input: a graph $G$ of $n$ vertices and with no odd cycles of length $\leq 2k-1$, where $k$ is an integer satisfying $(2k-1)^k \geq n$

Output: an independent set $I$ in $G$

1. $I = \emptyset$;
2. **while** $G$ is not empty **do**
   pick any vertex $v$ in $G$;
   Breadth First Search starting from $v$;
   let $L_0, L_1, \ldots, L_k$ be the first $k+1$ levels of vertices in the Breadth First Search tree;
   define $D_{2t} = \bigcup_{i=0}^{t} L_{2i}$ and $D_{2t+1} = \bigcup_{i=0}^{t} L_{2i+1}$, for $t = 0, 1, \ldots$;
   let $s$ be the smallest index satisfying $|D_s| \leq (2k-1)|D_{s-1}|$;
   $I = I \cup D_{s-1}$;
   remove all vertices in $D_s \cup D_{s-1}$ from the graph $G$;
3. **return** $I$.

Figure 9.12: finding an independent set in a graph without short odd cycles

independent set is possible. Consider the algorithm given in Figure 9.12.

**Lemma 9.4.9** *For a graph $G$ of $n$ vertices with no odd cycles of length $\leq 2k-1$, where $k$ is an integer satisfying $(2k-1)^k \geq n$, the Algorithm* **Large-IS**$(G, k)$ *runs in time $O(nm)$ and constructs an independent set $I$ of size at least $n/(2k)$.*

PROOF. First we need to show that it is always possible to fnd the index $s$ such that $|D_s| \leq (2k-1)|D_{s-1}|$. Suppose such an index does not exist. Then we have $|D_i| > (2k-1)|D_{i-1}|$ for all $i = 1, \ldots, k$. Therefore (note $|D_0| = 1$ and $(2k-1)^k \geq n$),

$$|D_k| > (2k-1)|D_{k-1}| > (2k-1)^2|D_{k-2}| > \cdots > (2k-1)^k|D_0| \geq n$$

This is impossible, since $D_k$ is a subset of vertices in the graph $G$ while $G$ has $n$ vertices. Therefore, the index $s$ always exists.

Since $|D_s| \leq (2k-1)|D_{s-1}|$, we have $|D_{s-1}| \geq (|D_s| + |D_{s-1}|)/(2k)$. Therefore, each time when we remove $|D_s| + |D_{s-1}|$ vertices from the graph $G$, we include $|D_{s-1}| \geq (|D_s| + |D_{s-1}|)/(2k)$ vertices in the set $I$. In consequence, the set $I$ constructed by the algorithm **Large-IS** has at least $n/(2k)$ vertices.

What remains is to show that the set $I$ is an independent set in $G$. For a Breadth First Search tree, every edge in $G$ either connects two vertices at the same level, or connects two vertices in the adjacent levels (See Appendix A). Therefore, no edge is between two vertices that belong to different levels in the set $D_{s-1}$ (note that $D_{s-1}$ only contains either odd levels only or even levels only in the Breadth First Search tree). Moreover, any edge connecting two vertices at the same level in $D_{s-1}$ would form an odd cycle of length $\leq 2k-1$ (recall $s \leq k$), which contradicts our assumption that the graph $G$ has no odd cycles of length $\leq 2k-1$. In conclusion, no two vertices in the set $D_{s-1}$ are adjacent and the set $D_{s-1}$ is an independent set. Since in each execution of the loop body, we also remove vertices in the set $D_s$, there is also no edge between two sets $D_{s-1}$ constructed in different stages in the algorithm. Thus, the set $I$ is an independent set in the graph $G$.

The analysis of the algorithm is easy. Each execution of the **while** loop body is a Breadth First Search on the graph $G$, which takes time $O(m)$, and removes at least one vertex from the graph $G$. Therefore, the algorithm runs in time $O(nm)$. $\square$

The conditions in Lemma 9.4.9 are bit too strong. We need to take care of the situation where graphs contain short odd cycles. Suppose that the vertices $v_1$, $v_2$, and $v_3$ form a triangle in a graph $G$. Then we observe that *every* minimum vertex cover of $G$ must contain at least two of these three vertices. Therefore, if our objective is an approximation ratio larger than 1.5, then intuitively it will not hurt if we include all three vertices in our vertex cover since the "local" approximation ratio for this inclusion is 1.5. In general, for a subgraph $H$ of $h$ vertices in $G$, if we know the ratio $h/Opt(H)$ is not larger than our objective ratio, then it seems reasonable to simply include all vertices in the subgraph $H$ and remove $H$ from $G$. This intuition is confirmed by the following lemma.

**Lemma 9.4.10** *Let $G$ be a graph and $H$ be a subgraph induced by $h$ vertices in $G$. Let $G^- = G - H$. Suppose that $C^-$ is a vertex cover of the graph $G^-$. Then $C^- \cup H$ is a vertex cover of the graph $G$ and*

$$\frac{|C^- \cup H|}{Opt(G)} \leq \max\left\{\frac{|C^-|}{Opt(G^-)}, \frac{h}{Opt(H)}\right\}$$

PROOF. Let $[u,v]$ be an edge in the graph $G$. If one of $u$ and $v$ is in the graph $H$, then certainly $[u,v]$ is covered by $C^- \cup H$. If none of $u$ and $v$ is in $H$, then $[u,v]$ is an edge in $G^-$ and must be covered by $C^-$. Therefore, $C^- \cup H$ is a vertex cover of the graph $G$.

Let $C_{\min}$ be a minimum vertex cover of the graph $G$. Let $C_{\min}^-$ be the set of vertices in $C_{\min}$ that are in the graph $G^-$, and let $C_{\min}^H$ be the set of

**Algorithm. VC-Apx-III**

Input: a graph $G$ of $n$ vertices
Output: a vertex cover $C$ of $G$

1. $C_1 = \emptyset$;
2. let $k$ the smallest integer such that $(2k-1)^k \geq n$;
3. **while** $G$ contains an odd cycle of length $\leq 2k-1$ **do**
   find an odd cycle $X$ of length $\leq 2k-1$;
   add all vertices of $X$ to $C_1$;
   delete all vertices of $X$ from the graph $G$;
4. let the remaining graph be $G'$, apply Theorem 9.4.6 to $G'$ to construct the subsets $C_0$ and $V_0$ of vertices in $G'$;
5. apply the algorithm **Large-IS**$(G(V_0), k)$ to construct an independent set $I$ in $G(V_0)$;
6. $C_2 = C_0 \cup (V_0 - I)$;
7. **return** $C = C_1 \cup C_2$.

Figure 9.13: Approximating vertex cover III

vertices in $C_{\min}$ that are in $H$. Then $C^-_{\min}$ is a vertex cover of the graph $G^-$ and $C^H_{\min}$ is a vertex cover of the graph $H$. Therefore, we have

$$\begin{aligned}\frac{|C^- \cup H|}{Opt(G)} &= \frac{|C^- \cup H|}{|C_{\min}|} = \frac{|C^-| + h}{|C^-_{\min}| + |C^H_{\min}|} \\ &\leq \frac{|C^-| + h}{Opt(G^-) + Opt(H)} \leq \max\left\{\frac{|C^-|}{Opt(G^-)}, \frac{h}{Opt(H)}\right\}\end{aligned}$$

here we have used the facts $|C^-_{\min}| \geq Opt(G^-)$, $|C^H_{\min}| \geq Opt(H)$, and $(a+b)/(c+d) \leq \max\{a/c, b/d\}$ for any positive numbers $a$, $b$, $c$, and $d$. $\square$

If the subgraph $H$ is a cycle of length $h = 2k-1$, obviously we have $h/Opt(H) = (2k-1)/k = 2 - 1/k$. According to Lemma 9.4.10, if our objective approximation ratio is not smaller than $2 - 1/k$, then we can remove the cycle $H$ from the graph by simply including all vertices in $H$ in the vertex cover. Repeating this procedure, we will result in a graph $G'$ with no short odd cycles. Now applying the algorithm **Large-IS** on $G'$ gives us a larger independent set $I$, from which a better vertex cover is obtained. These ideas are implemented in the algorithm given in Figure 9.13.

**Theorem 9.4.11** *Approximation algorithm* **VC-Apx-III** *for* MIN VERTEX COVER *runs in time $O(nm)$, and has approximation ratio $2 - \frac{\log\log n}{2\log n}$.*

PROOF. The time complexity of all steps, except step 3, of the algorithm has been discussed and is bounded by $O(nm)$. To find an odd cycle of length bounded by $2k-1$ in step 3, we pick any vertex $v$ and perform Breadth First Search starting from $v$ for at most $k+1$ levels. Either we will find an edge connecting two vertices at the same level, which gives us an odd cycle of length bounded by $2k-1$, or we do not find such an odd cycle. In the former case, the cycle will be removed from the graph $G$, while in the latter case, the vertex $v$ is not contained in any odd cycle of length bounded by $2k-1$. Therefore, the vertex $v$ can be removed from the graph in the latter search for odd cycles. In any case, each Breadth First Search removes at least one vertex from the graph. We conclude that at most $n$ Breadth First Searches are performed in step 3. Since each Breadth First Search takes time $O(m)$, the time complexity of step 3 is $O(nm)$. Summarizing all these, we conclude that the time complexity of the algorithm **VC-Apx-III** is $O(nm)$.

We prove that the approximation ratio of the algorithm **VC-Apx-II** is bounded by $2-1/k$, where $k$ is defined in step 2 of the algorithm.

Let $H$ be the subgraph of $G$ consisting of all the odd cycles removed in step 3. Since each cycle $X$ in $H$ has length $2j-1$, where $j \leq k$, we have $(2j-1)/Opt(X) = (2j-1)/j = 2-1/j \leq 2-1/k$. Since all cycles in $H$ are disjoint, we have $h/Opt(H) \leq 2-1/k$, where $h$ is the number of vertices in $H$. By Lemma 9.4.10, to prove that the algorithm **VC-Apx-III** has an approximation ratio bounded by $2-1/k$, it suffices to proved that the set $C_2$ constructed in step 6 is a vertex cover of the graph $G' = G-H$ satisfying $|C_2|/Opt(G') \leq 2-1/k$.

By Lemma 9.4.9, $I$ is an independent set of at least $|V_0|/(2k)$ vertices in the graph $G(V_0)$. Therefore, $V_0 - I$ is a vertex cover of $G(V_0)$ with at most $|V_0| - |V_0|/(2k) = |V_0|(1-1/(2k))$ vertices. Therefore,

$$\frac{|V_0 - I|}{Opt(G(V_0))} \leq \frac{|V_0|(1-1/(2k))}{Opt(G(V_0))} \leq \frac{|V_0|(1-1/(2k))}{|V_0|/2} = 2 - \frac{1}{k}$$

From this and Corollary 9.4.7, $C_2 = C_0 \cup (V_0 - I)$ is a vertex cover of the graph $G'$ satisfying

$$\frac{|C_2|}{Opt(G')} \leq \frac{|V_0 - I|}{Opt(G(V_0))} \leq 2 - \frac{1}{k}$$

Now $|C|/Opt(G) \leq 2-1/k$ follows from Lemma 9.4.10. Thus, the approximation ratio of the algorithm **VC-Apx-II** is bounded by $2-1/k$. Since $k$ is the smallest integer satisfying $(2k-1)^k \geq n$, we can derive from elementary mathematics that $k \leq (2 \log n)/(\log \log n)$. This completes the proof of the theorem. $\square$

The ratio in Theorem 9.4.11 is best known result for polynomial time approximation algorithms for MIN VERTEX COVER. We point out that the above techniques can be extended to design approximation algorithms with the same ratio for the weighted version of MIN VERTEX COVER. Interested readers are referred to [11].

# Chapter 10

# Probabilistic Methods

Probabilistic methods have been developed recently and become a very powerful and widely used tool in combinatorics and computer algorithm design. In particular, randomized algorithms have found widespread applications in many problem domains. A randomized algorithm is an algorithm that can use the outcome of a random process. Typically, such an algorithm would contain an instruction to "flip a coin," and the result of that coin flip would influence the algorithm's subsequent execution and output. Two reasons that have made randomized algorithms popular are their simplicity and efficiency. For many applications, randomized algorithms often provide the simplest, most natural and most efficient solutions.

The original ideas of the probability methods, initiated by Paul Erdos, can be described as follows: in order to prove the existence of a combinatorial object with a specified property $A$, we construct an appropriate probabilistic space for all related objects, with or without the property $A$, and show that a randomly chosen element in this space has property $A$ with positive probability. Note that this method is somehow "non-constructive" in the sense that it does not tell how to find an object with property $A$. A comprehensive discussion for probabilistic methods is given in Alon and Spencer [2].

An implementation of the above probabilistic methods in randomized algorithms is for certain combinatorial structures to prove that a randomly chosen object has property $A$ with a high probability. This in general implies a simple and efficient randomized algorithm for finding an object with property $A$: just randomly pick a few objects, then with a very high probability, an object with property $A$ should be picked. Readers are referred to Motwani and Raghavan [98] for more systematic discussions on randomized algorithms.

Sometimes a randomized algorithm can be "derandomized". Derandomization is a process that converts a randomized algorithm into an efficient deterministic algorithm that performs equally well. Therefore, the probabilistic methods have also become an important technique in designing efficient deterministic algorithms.

A common misconception regarding the probabilistic methods is that one must have deep knowledge in probability theory in order to use the methods. This is far from the truth. In fact, a basic understanding of probability theory along with familiarity with some clever combinatorial reasoning is sufficient in many cases to derive interesting results using the probabilistic methods and develop very powerful randomized algorithms. In this chapter, we illustrate how efficient approximation algorithms for optimization problems can be developed based on the probabilistic methods. We start with a few basic concepts and useful principles in probability theory that are directly related to our discussion. We then describe a general derandomization technique, using Johnson's algorithm for the MAX-SAT problem as an illustration (see Figure 9.5). Randomized approximation algorithms for a variety of NP-hard optimization problems are then presented. These randomized algorithms can be derandomized based on the derandomization techniques.

## 10.1 Linearity of expectation

In this section, we describe several basic concepts and a few useful principles in probability theory that are directly related to our discussion. The reader may read Appendix C in this book for a very quick review of the fundamentals of probability theory.

Let $(\Omega, P)$ be a probabilistic space, where we assume that the sample space $\Omega$ is countable (i.e., either finite or countably infinite). Let $X$ be a random variable over the probabilistic space $(\Omega, P)$. Recall that the *expectation* $E(X)$ of the random variable $X$ on the probabilistic space $(\Omega, P)$ is defined by

$$E(X) = \sum_{q \in \Omega} X(q)P(q)$$

provided that the series $\sum_{q \in \Omega} X(q)P(q)$ is absolutely convergent. In this case we say that the expectation $E(X)$ *exists*.

Two random variables $X$ and $Y$ are *independent* if for any two real numbers $y_1$ and $y_2$, we have

$$P(X_1 = y_1, X_2 = y_2) = P(X_1 = y_1) \cdot P(X_2 = y_2)$$

The most fundamental tool of the probabilistic methods is the *First Moment Method*. The essence of the First Moment Method lies in the following simple yet surprisingly powerful fact:

**Theorem 10.1.1 (The First Moment Principle)** *Let $X$ be a random variable over a probabilistic space $(\Omega, P)$ and let $t$ be a real number. If $E(X) \leq t$ then $P(X \leq t) > 0$.*

PROOF. By the definition

$$E(X) = \sum_{q \in \Omega} X(q)P(q) = \sum_{q:X(q) \leq t} X(q)P(q) + \sum_{q:X(q) > t} X(q)P(q)$$

If $P(X \leq t) = 0$ then $P(X > t) = 1$, and for any sample point $q$ with $X(q) \leq t$ we have $P(q) = 0$. Therefore

$$E(X) = \sum_{q:X(q) > t} X(q)P(q) > t \cdot P(X > t) = t$$

This contradicts the assumption $E(X) \leq t$. $\square$

**Lemma 10.1.2** *Let $X$ and $Y$ be random variables on a probabilistic space $(\Omega, P)$. If $E(X)$ and $E(Y)$ exist, then so do the expectations $E(X+Y)$ and $E(aX)$, where $a$ is any real number, and*

$$E(X+Y) = E(X) + E(Y) \qquad \text{and} \qquad E(aX) = a \cdot E(X)$$

PROOF. The absolute convergence of the series $\sum_{q \in \Omega}(X(q) + Y(q))P(q)$ and $\sum_{q \in \Omega}(a \cdot X(q))P(q)$ follows directly from the absolute convergence of the series $\sum_{q \in \Omega} X(q)P(q)$ and $\sum_{q \in \Omega} Y(q)P(q)$. Thus, the expectations $E(X + Y)$ and $E(aX)$ exist.

According to the definition, we have

$$\begin{aligned} E(X+Y) &= \sum_{q \in \Omega}(X(q) + Y(q))P(q) \\ &= \sum_{q \in \Omega} X(q)P(q) + \sum_{q \in \Omega} Y(q)P(q) \\ &= E(X) + E(Y) \end{aligned}$$

where the second equality is valid because of the absolute convergence of the series $\sum_{q\in\Omega} X(q)P(q)$ and $\sum_{q\in\Omega} Y(q)P(q)$, and

$$E(aX) = \sum_{q\in\Omega}(a \cdot X(q))P(q) = a \cdot \sum_{q\in\Omega} X(q)P(q) = a \cdot E(X)$$

This proves the lemma. □

A very useful principle, the *linearity of expectation*, follows directly from Lemma 10.1.2.

**Theorem 10.1.3 (Linearity of Expectation)** *Let $X_1$, …, $X_n$ be random variables on a probabilistic space $(\Omega, P)$ such that $E(X_i)$ exists for all $i$. Then for any real numbers $a_1$, …, $a_n$, $E(a_1X_1 + \cdots + a_nX_n)$ exists and*

$$E(a_1X_1 + \cdots + a_nX_n) = a_1E(X_1) + \cdots + a_nE(X_n)$$

A remarkable property of Theorem 10.1.3 is that it has no restrictions on the independence of the random variables $X_1$, …, $X_n$.

Theorem 10.1.3 does not generalize to the product of random variables. On the other hand, for independent random variables, we have the following theorem.

**Theorem 10.1.4** *Let $X$ and $Y$ be two independent random variables on a probabilistic space $(\Omega, P)$ such that $E(X)$ and $E(Y)$ exist. Then $E(XY)$ exists and*

$$E(XY) = E(X)E(Y)$$

PROOF. Again we give a proof under the assumption that the sample space $\Omega$ is countable.

Let $R_X$ and $R_Y$ be the ranges of $X$ and $Y$, respectively. Since the sample space $\Omega$ is countable, the ranges $R_X$ and $R_Y$ are also countable. For each value $x$ in $R_X$ and each value $y$ in $R_Y$, let

$$A_{x,y} = \{\ q \mid X(q) = x \text{ and } Y(q) = y\}$$

We have

$$\begin{aligned} E(X)E(Y) &= (\sum_{q\in\Omega} X(q)P(q))(\sum_{q\in\Omega} Y(q)P(q)) \\ &= (\sum_{x\in R_X}(\sum_{q:X(q)=x} x \cdot P(q)))(\sum_{y\in R_Y}(\sum_{q:Y(q)=y} y \cdot P(q))) \end{aligned}$$

$$
\begin{aligned}
&= \Big(\sum_{x \in R_X} x \cdot P(X = x)\Big)\Big(\sum_{y \in R_Y} y \cdot P(Y = y)\Big) \\
&= \sum_{x \in R_X} \sum_{y \in R_Y} xy \cdot P(X = x) P(Y = y) \\
&= \sum_{x \in R_X} \sum_{y \in R_Y} xy \cdot P(X = x, Y = y) \\
&= \sum_{x \in R_X} \sum_{y \in R_Y} xy \cdot P(A_{x,y}) \qquad (10.1)
\end{aligned}
$$

The fifth equality is because of the independence of the random variables $X$ and $Y$. Note that the validity of many derivations in (10.1) is heavily based on the absolute convergence of the series $\sum_{q \in \Omega} X(q)P(q)$ and $\sum_{q \in \Omega} Y(q)P(q)$, i.e., the existence of the expectations $E(X)$ and $E(Y)$. In particular, the series $\sum_{x \in R_X} \sum_{y \in R_Y} xy \cdot P(A_{x,y})$ is also absolutely convergent. Therefore,

$$
\begin{aligned}
&\sum_{x \in R_X} \sum_{y \in R_Y} xy \cdot P(A_{x,y}) = \sum_{x \in R_X} \sum_{y \in R_Y} xy \sum_{q \in A_{x,y}} P(q) \\
= &\sum_{x \in R_X} \sum_{y \in R_Y} \sum_{q \in A_{x,y}} X(q)Y(q)P(q) = \sum_{q \in \Omega} X(q)Y(q)P(q) = E(XY)
\end{aligned}
$$

This proves the theorem. $\square$

The concept of conditionality of probability can be conveniently generalized to the expectation of random variables. Let $B$ be an event with $P(B) > 0$ and let $X$ be a random variable. The *conditional expectation* of the random variable $X$ *relative to* $B$ is written as $E(X|B)$ and defined by

$$E(X|B) = \sum_{q \in \Omega} X(q)P(q|B)$$

Thus, we simply replace in the formula $E(X) = \sum_{q \in \Omega} X(q)P(q)$ the probabilities by the conditional ones. Intuitively, $E(X|B)$ is the "average value" of the random variable $X$ over the set $B$. Note that if $E(X)$ exists then so does $E(X|B)$ because $P(q|B) \leq P(q)/P(B)$ so the absolute convergence of the series $\sum_{q \in \Omega} X(q)P(q|B)$ follows directly from the absolute convergence of the series $\sum_{q \in \Omega} X(q)P(q)$. In particular, the event $B$ can be given by another random variable $Y$. For example, $E(X|Y = y) = E(X|B_Y)$ where the event $B_Y$ is defined by $B_Y = \{\, q \mid Y(q) = y \}$.

**Theorem 10.1.5** *Let $A_1, \ldots, A_n$ be a partition of the sample space $\Omega$, and let $X$ be a random variable over the probabilistic space $(\Omega, P)$ such that $E(X)$ exists. Then*

$$E(X) = \sum_{i=1}^{n} P(A_i)E(X|A_i)$$

PROOF. By the definition, $E(X) = \sum_{q \in \Omega} X(q)P(q)$. By Proposition C.2(1) in Appendix C, we have

$$P(q) = \sum_{i=1}^{n} P(q|A_i)P(A_i)$$

Therefore,

$$\begin{aligned} E(X) &= \sum_{q \in \Omega} X(q) \cdot (\sum_{i=1}^{n} P(q|A_i)P(A_i)) \\ &= \sum_{i=1}^{n} P(A_i) \sum_{q \in \Omega} X(q)P(q|A_i) \\ &= \sum_{i=1}^{n} P(A_i)E(X|A_i) \end{aligned}$$

The validity of the second equality is ensured by the absolute convergence of the series $\sum_{q \in \Omega} X(q)P(q)$. $\square$

Theorem 10.1.3 can also be extended to conditional expectations, with simple modifications in the proof. We state the result as follows and leave the proof to the reader.

**Theorem 10.1.6** *Let $X_1, \ldots, X_n$ be random variables on a probabilistic space $(\Omega, P)$ such that $E(X_i)$ exists for all $i$, and let $B$ be any event with $P(B) > 0$. Then for any real numbers $a_1, \ldots, a_n$, $E(a_1X_1 + \cdots + a_nX_n|B)$ exists and*

$$E(a_1X_1 + \cdots + a_nX_n|B) = a_1E(X_1|B) + \cdots + a_nE(X_n|B)$$

## 10.2 Derandomization

As we mentioned at the beginning of this chapter, the probabilistic methods in many cases supply effective randomized algorithms for various computation problems. In some cases, these randomized algorithms can be "derandomized" and converted into deterministic algorithms. The most common technique for derandomization is based on the *method of conditional probabilities* due to Erdös and Selfridge [36]. In this section, we first describe a randomized algorithm for the MAX-SAT problem, and show how the algorithm is derandomized using the above method. This discussion re-interprets Johnson's algorithm (see Figure 9.5) for the MAX-SAT problem as a derandomization of a randomized algorithm. Based on this interpretation, we show how to improve the approximation ratio of Johnson's algorithm for the MAX-SAT problem.

Recall that an instance of the MAX-SAT problem is a set of clauses $F = \{C_1, \ldots, C_m\}$ on a set of boolean variables $\{x_1, \ldots, x_n\}$, with the objective to find an assignment $\tau$ on the boolean variables that maximizes the number of satisfied clauses.

Consider the following randomized algorithm: for each boolean variable $x_i$, we *independently* assign $x_i$ with value TRUE with probability $1/2$ and FALSE with probability $1/2$. This builds a probabilistic space $(\Omega, P)$ as follows: each sample point $\tau$ in the sample space $\Omega$ is an assignment to the boolean variables $\{x_1, \ldots, x_n\}$. Therefore, the sample space $\Omega$ is finite and has totally $2^n$ sample points. For a boolean value $b =$ TRUE or FALSE, let $B_{x_i=b}$ be the event that includes all assignments in $\Omega$ that assign the boolean variable $x_i$ with value $b$. To simplify the expressions, we write the probability $P(B_{x_i=b})$ in a more compact form $P(x_i = b)$. From our construction, we know $P(x_i = b) = 1/2$ for all boolean variables $x_i$ and for all boolean value $b$. For an assignment $\tau$, we will denote by $\tau(x_i)$ the value of the boolean variable $x_i$ under the assignment $\tau$. For each assignment $\tau \in \Omega$, the probability $P(\tau)$ is naturally defined by (note that by our construction, the events $B_{x_i=\tau(x_i)}$ and $B_{x_j=\tau(x_j)}$ for $i \neq j$ are independent):

$$\begin{aligned} P(\tau) &= P(x_1 = \tau(x_1), \ldots, P(x_n = \tau(x_n)) \\ &= P(x_1 = \tau(x_1)) \cdots P(x_n = \tau(x_n)) \\ &= \frac{1}{2^n} \end{aligned}$$

It is easy to verify that $(\Omega, P)$ makes a probabilistic space. Note that this also matches our intuition: we independently assign each boolean variable $x_i$ by the value TRUE or FALSE with equal probability $1/2$, thus all assignments to the boolean variables $\{x_1, \ldots, x_n\}$ should be equally like: each of the $2^n$ assignments in the sample space $\Omega$ has probability $1/2^n$.

Recall that we say an assignment satisfies a clause if the assignment makes the clause have value TRUE. For each clause $C_j$ in the set $F$, $1 \leq j \leq m$, define a random variable $X_j$ on the sample space $\Omega$ such that for each assignment $\tau$ to the boolean variables $\{x_1, \ldots, x_n\}$, we have

$$X_j(\tau) = \begin{cases} 1 & \text{if } \tau \text{ does not satisfy } C_j \\ 0 & \text{if } \tau \text{ satisfies } C_j \end{cases}$$

The linear combination of these random variables $X_j$, $1 \leq j \leq m$, $X = X_1 + \cdots + X_m$ defines a new random variable on the sample space $\Omega$ such that for each assignment $\tau$ in $\Omega$, $X(\tau)$ is the number of clauses in $F$ not satisfied by the assignment $\tau$. We consider the expectations of these random variables.

Let $C_j$ be a clause of $k$ literals in $F$: $C_j = (a_1 \vee \cdots \vee a_k)$, where $a_h$ are literals in $\{x_1, \ldots, x_n\}$. Without generality, we assume that no variable $x_i$ has both $x_i$ and $\overline{x_i}$ in the clause $C_j$. Let $B_j$ be the set of assignments in $\Omega$ that do not satisfy the clause $C_j$. Then $B_j$ is an event in $\Omega$. It is easy to see that an assignment $\tau$ is in the event $B_j$ if and only if $\tau$ makes all literals $a_1$, ..., $a_k$ have value FALSE. Thus,

$$\begin{aligned} P(B_j) &= P(\tau(a_1) = \text{FALSE}, \ldots, \tau(a_k) = \text{FALSE}) \\ &= P(\tau(a_1) = \text{FALSE}) \cdots P(\tau(a_k) = \text{FALSE}) \\ &= \frac{1}{2^k} \end{aligned} \tag{10.2}$$

Note that the second equality is because the values for two different boolean variables $x_i$ and $x_j$ were assigned independently and the clause $C_j$ does not contain both $x_i$ and $\overline{x_i}$ for any variable $x_i$, so the events $B_{\tau(a_i)=\text{FALSE}}$ and $B_{\tau(a_j)=\text{FALSE}}$ are independent.

By the definition of expectation

$$E(X_j) = \sum_{\tau \in \Omega} X_j(\tau) P(\tau) = \sum_{\tau \in B_j} P(\tau) = P(B_j) = \frac{1}{2^k}$$

## 10.3 Linear Programming relaxation

Let $F = \{C_1, \ldots, C_m\}$ be a set of clauses on boolean variables $x_1$, $x_2$, ..., $x_n$, which is an instance of the MAX-SAT problem. For each boolean variable $x_i$, randomly assign $x_i = 1$ with probability $p_i$. The expectation value of the number of clauses to be satisfied by this random assignment is

$$\sum_{j=1}^{m} \left( 1 - \prod_{x_i \in C_j} (1 - p_i) \prod_{\overline{x}_i \in C_j} p_i \right) \tag{10.3}$$

As we discussed in the previous section, the extended Johnson's algorithm **Johnson-Extended** assigns each variable $x_i$ with value either 0 or 1 (deterministically) and keeps the expectation value of the number of clauses to be satisfied nondeceasing. Therefore, at the end of the algorithm when every boolean variable gets an assigned value while the expectation value of the number of clauses to be satisfied is at least as large as the value in (10.3), we obtain an assignment of the boolean variables such that the number of clauses satisfied by this assignment is at least as large as (10.3).

In this section, we introduce a powerful technique, *linear programming relaxation*, and illustrate how this technique is used to improve the approximation ratio for the MAX-SAT problem.

We first reduce the instance $F = \{C_1, \ldots, C_m\}$ of the MAX-SAT problem to an instance of the integer linear programming problem INTEGER LP, as we introduced in Section 5.1.

$$
\begin{array}{lll}
(IP_F): & \text{maximize} & z_1 + z_2 + \cdots + z_m \\
& \text{subject to} & \\
& & \displaystyle\sum_{x_i \in C_j} x_i + \sum_{\overline{x}_i \in C_j} (1 - x_i) \geq z_j \qquad j = 1, \ldots, m \\
& & x_i, z_j = 0 \text{ or } 1 \qquad i = 1, \ldots, n; \quad j = 1, \ldots, m
\end{array}
$$

Clearly, an optimal solution $(x_1^o, \ldots, x_n^o, z_1^o, \ldots, z_m^o)$ to the instance $IP_F$ of INTEGER LP gives an optimal assignment $\sigma_o = (x_1^o, \ldots, x_n^o)$ to the instance $F$ of MAX-SAT with maximized objective function value $Opt(F) = Opt(IP_F) = z_1^o + \cdots + z_m^o$. Unfortunately, according to Theorem ??, INTEGER LP is NP-hard.

Since general linear programming problem LP is solvable in polynomial time (Theorem ??), we try to "relax" the integral constraint in the instance $IP_F$ for INTEGER LP and see how this relaxation would help us in deriving a good approximation for the instance $F$ of MAX-SAT.

$$
\begin{array}{lll}
(LP_F): & \text{maximize} & z_1 + z_2 + \cdots + z_m \\
& \text{subject to} & \\
& & \displaystyle\sum_{x_i \in C_j} x_i + \sum_{\overline{x}_i \in C_j} (1 - x_i) \geq z_j \qquad j = 1, \ldots, m \\
& & 0 \leq x_i, z_j \leq 1 \qquad i = 1, \ldots, n; \quad j = 1, \ldots, m
\end{array}
$$

Let $A^* = (x_1^*, \ldots, x_n^*, z_1^*, \ldots, z_m^*)$ be an optimal solution to the instance $LP_F$ with optimal objective function value $Opt(LP_F) = z_1^* + \cdots + z_m^*$. By Theorem ??, $A^*$ can be constructed from $LP_F$ in polynomial time. Since

**Algorithm. LP-Relaxation**

Input: a set $F = \{C_1, \ldots, C_m\}$ of clauses
Output: an assignment to the boolean variables in $F$

1. solve the linear programming problem $LP_F$ and let the optimal solution be $(x_1^*, \ldots, x_n^*, z_1^*, \ldots, z_m^*)$;
2. **for** $i = 1$ **to** $n$ **do** $p_i = x_i^*$;
3. call algorithm **Johnson-Extended** with the probability assignment $(p_1, \ldots, p_n)$;
4. output the assignment constructed in step 3.

Figure 10.1: Approximating MAX-SAT by LP Relaxation

each feasible solution to the instance $IP_F$ is also a feasible solution to the instance $LP_F$, we have $Opt(LP_F) \geq Opt(IP_F)$, which gives an upper bound $Opt(LP_F)$ for the optimal value $Opt(F)$. This estimation of the value $Opt(F)$ is obviously more precise than the bound $m$, which is the total number of clauses in $F$, as we have used in the analysis for Johnson's algorithm.

Unfortunately, the values $(x_1^*, \ldots, x_n^*)$ in $A^*$ cannot be directly used as an assignment to the boolean variables $x_1$, …, $x_n$ in the instance $F$ of MAX-SAT. In general, the value $x_i^*$ can be a non-integral number between 0 and 1 while assignining a boolean variable $x_i$ with a non-integral value makes no sense. However, the values $(x_1^*, \ldots, x_n^*)$ do provide us with useful information for a good assignment to the variables $x_1$, …, $x_n$. For example, suppose that $x_1^* = 0.95$ and $x_2^* = 0.03$. Then we would expect that in order to maximize the objective function value, the variable $x_1$ seems to need to take a large value while the variable $x_2$ seems to need to take a small value. In particular, if $x_1$ and $x_2$ are boolean variables, then it seems that $x_1$ should be likely to take value 1 while $x_2$ should be likely to take value 0.

A natural implementation of the above idea is to assign each boolean variable $x_i = 1$ with probability $x_i^*$ (note we have $0 \leq x_i^* \leq 1$), then to run the algorithm (**Johnson-Extended**). This algorithm is illustrated in Figure 10.1.

Suppose that upon this random assignment, the algorithm **Johnson-Extended** results in an assignment $\sigma_*$ to the boolean variables $x_1$, …, $x_n$ in the instance $F$ of MAX-SAT. By the discussion in the previous section, the number of clauses in $F$ satisfied by this assignment $\sigma_*$ is

$$E^* = \sum_{j=1}^{m} \left( 1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \right) \qquad (10.4)$$

Therefore, the approximation ratio of this assignment $\sigma_*$ is bounded by $Opt(F)/E^*$. Now we estimate the value $E^*$ in terms of the optimal solution value $Opt(F)$.

**Lemma 10.3.1** *Suppose that the clause $C_j$ in $F$ has $k$ literals. Then*

$$1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \geq \beta_k z_j^*$$

*where $(x_1^*, \ldots, x_n^*, z_1^*, \ldots, z_m^*)$ is an optimal solution to the instance $LP_F$, and $\beta_k = 1 - (1 - 1/k)^k$. In particular, for any clause $C_j$ in $F$, we have*

$$1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \geq \left(1 - \frac{1}{e}\right) z_j^*$$

*where $e$ is the base of natural logarithms $e = \sum_{n=0}^{\infty} 1/n! \approx 2.718$.*

PROOF. It is well-known that for any $k$ nonnegative numbers $a_1$, $a_2$, $\ldots$, $a_k$, the arithmetic mean is at least as large as the geometric mean (for a proof, see [??]):

$$\frac{a_1 + a_2 + \cdots + a_k}{k} \geq \sqrt[k]{a_1 a_2 \cdots a_k}$$

Therefore, we have

$$\begin{array}{rc} & \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \\ \leq & \left( \frac{\sum_{x_i \in C_j} (1 - x_i^*) + \sum_{\overline{x}_i \in C_j} x_i^*}{k} \right)^k \\ = & \left( 1 - \frac{\sum_{x_i \in C_j} x_i^* + \sum_{\overline{x}_i \in C_j} (1 - x_i^*)}{k} \right)^k \\ \leq & \left( 1 - \frac{z_j^*}{k} \right)^k \end{array}$$

The last inequality is because $(x_1^*, \ldots, x_n^*, z_1^*, \ldots, z_m^*)$ is a solution to the instance $LP_F$ of LP so we have

$$\sum_{x_i \in C_j} x_i^* + \sum_{\overline{x}_i \in C_j} (1 - x_i^*) \geq z_j^*$$

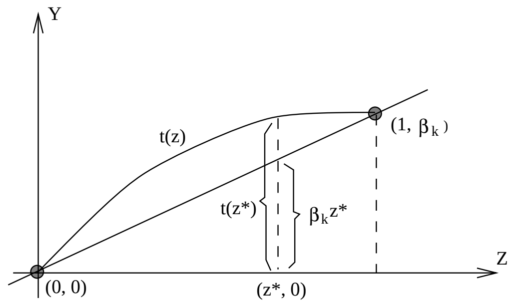



Figure 10.2: The value $f(z)$ is larger than $\beta_k z$.

From this inequality, we get immediately

$$1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \geq 1 - \left(1 - \frac{z_j^*}{k}\right)^k \tag{10.5}$$

Define a function $t(z) = 1 - (1 - z/k)^k$, then $t(0) = 0$ and $t(1) = \beta_k$. Since the second derivative of the function $t(z)$ is not larger than 0: $t''(z) = -(k-1)(1-z/k)^{k-2}/k$ in the interval $[0,1]$ (we assume $k \geq 1$), the function $t(z)$ is concave in the interval $[0,1]$. This implies that the curve $y = f(z)$ is above the line $y = \beta_k z$ connecting the two points $(0,0)$ and $(1, \beta_k)$ in the interval $[0,1]$ (see Figure 10.2). In particular, since $0 \leq z_j^* \leq 1$, we have $f(z_j^*) \geq \beta_k z_j^*$. Combining this with (10.5), we obtain

$$1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \geq \beta_k z_j^*$$

Since $\beta_k$ is nonincreasing in terms of $k$, and since $\lim_{k \to \infty} \beta_k = 1 - 1/e$, we conclude that for any clause $C_j$, we have

$$1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \geq \left(1 - \frac{1}{e}\right) z_j^*$$

This completes the proof. □

Lemma 10.3.1 gives immediately the following theorem.

**Theorem 10.3.2** *The algorithm* **LP Relaxation** *for* MAX-SAT *runs in polynomial time and has approximation ratio bounded by* $e/(1-e) \approx 1.58$.

PROOF. Since linear programming problem LP is solvable in polynomial time, the algorithm **LP Relaxation** runs in polynomial time.

According to the discussion in the previous section, the assignment constructed by the algorithm satisfies at least

$$E^* = \sum_{j=1}^{m} \left( 1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \right)$$

clauses in the instance $F$. According to Lemma 10.3.1,

$$1 - \prod_{x_i \in C_j} (1 - x_i^*) \prod_{\overline{x}_i \in C_j} x_i^* \geq \left(1 - \frac{1}{e}\right) z_j^*$$

and notice that $\sum_{j=1}^{m} z_j^*$ is the value of the optimal solution to the instance $LP_F$, which is at least as large as $Opt(F)$, we obtain

$$E^* \geq \left(1 - \frac{1}{e}\right) \sum_{j=1}^{m} z_j^* \geq \left(1 - \frac{1}{e}\right) Opt(F)$$

This implies directly that the approximation ratio of the algorithm **LP Relaxation** is bounded by

$$Opt(F)/E^* \leq \frac{1}{1 - 1/e} = \frac{e}{e-1}$$

This completes the proof. □

A recent analysis [19] shows that the approximation ratio for Johnson's original algorithm is actually 1.5. Therefore, the ratio $e/(1-e) \approx 1.58$ of the algorithm **LP Relaxation** is actually not better than that of Johnson's original algorithm.

It is interesting to observe that the algorithm **LP Relaxation** and Johnson's algorithm in some sense complement each other. According to Theorem ??, the number of clauses satisfied by the assignment constructed by Johnson's algorithm is at least

$$\sum_{j=1}^{m} \left(1 - \frac{1}{2^{|C_j|}}\right) = \sum_{k \geq 1} \sum_{|C_j| = k} \alpha_k$$

where $\alpha_k = 1 - 1/2^k$, and by Lemma 10.3.1, the number of clauses satisfied by the assignment constructed by **LP Relaxation** is at least

$$\sum_{k \geq 1} \sum_{|C_j| = k} \beta_k z_j^*$$

> **Algorithm. MaxSAT-Improved**
>
> Input: a set $F = \{C_1, \ldots, C_m\}$ of clauses
> Output: an assignment to the boolean variables in $F$
>
> 1. call algorithm **LP Relaxation** to construct an assignment $\sigma_1$ for $F$;
> 2. call Johnson's original algorithm to construct an assignment $\sigma_2$ for $F$;
> 3. output the better one of $\sigma_1$ and $\sigma_2$.

Figure 10.3: Combining the LP Relaxation and Johnson's algorithm

where $\beta_k = 1 - (1 - 1/k)^k$. Note that the value $\alpha_k$ incrases in terms of $k$ while the value $\beta_k$ descreases in terms of $k$. More specifically, Johnson's algorithm does better for clauses with more literals while the algorithm **LP Relaxation** does better for clauses with fewer literals. This observation motivates the idea of combining the two algorithms to result in a better approximation ratio. Consider the algorithm given in Figure 10.3.

**Theorem 10.3.3** *The algorithm* **MaxSAT-Improved** *for* MAX-SAT *runs in polynomial time and has approximation ratio bounded by* $4/3$.

PROOF. The algorithm obviously runs in polynomial time.

Let $m_J$ be the number of clauses in $F$ satisfied by the assignment constructed by Johnson's original algorithm, and let $m_L$ be the number of clauses in $F$ satisfied by the assignment constructed by the algorithm **LP Relaxation**. By the above discussion, we have

$$m_J \geq \sum_{k\geq 1} \sum_{|C_j=k|} \alpha_k \qquad \text{and} \qquad m_L \geq \sum_{k\geq 1} \sum_{|C_j=k|} \beta_k z_j^*$$

where $\alpha_k = 1-1/2^k$, and $\beta_k = 1-(1-1/k)^k$. According to the algorithm, the number of clauses satisfied by the assignment constructed by the algorithm **MaxSAT-Improved** is

$$\begin{aligned}
& \max\{m_J, m_L\} \geq (m_J + m_L)/2 \\
\geq & \left(\textstyle\sum_{k\geq 1} \sum_{|C_j=k|} \alpha_k + \sum_{k\geq 1} \sum_{|C_j=k|} \beta_k z_j^*\right)/2 \\
\geq & \left(\textstyle\sum_{k\geq 1} \sum_{|C_j=k|} \alpha_k z_j^* + \sum_{k\geq 1} \sum_{|C_j=k|} \beta_k z_j^*\right)/2 \\
= & \textstyle\sum_{k\geq 1} \sum_{|C_j=k|} \left(\frac{\alpha_k+\beta_k}{2}\right) z_j^*
\end{aligned}$$

In the second inequality, we have used the fact $0 \leq z_j^* \leq 1$ for all $j$. Now it is not difficult to verify that for $k \geq 1$, $\alpha_k + \beta_k \geq 3/2$. We conclude

$$\max\{m_J, m_L\} \geq \frac{3}{4}\sum_{j=1}^{m} z_j^* \geq \frac{3}{4} Opt(F)$$

Here we have used the fact that $\sum_{j=1}^{m} z_j^*$ is the value of an optimal solution to the instance $LP_F$, which is at least as large as $Opt(IP_F) = Opt(F)$.

This implies immediately that the approximation ratio of the algorithm **MaxSAT-Improved** is bounded by $4/3$. $\square$

Approximation algorithms for MAX-SAT have been a very active research area in the last two decades (for research before 1990 see [60], and for more recent research see [8]). We make two remarks before we close this section.

A natural generalization of the MAX-SAT problem is the WEIGHTED MAX-SAT problem in which each clause has a weight and we are looking for assignments to the boolean variables that maximize the sum of the weights of the satisfied clauses. All algorithms we have discussed can be easily modified to work for WEIGHTED MAX-SAT without affecting the approximation ratio.

Relaxation techniques have been very successful in the recent study of approximation algorithms for MAX-SAT. After the discovery of the approximation algorithm for MAX-SAT based on linear programming relaxation, as we discussed in this section, relaxation of other mathematical programmings has also been investigated. In particular, relaxations on semidefinite programming have been investigated carefully for further improvement of approximation ratio for MAX-SAT. On the other hand, the study of inapproximability of MAX-SAT has also been making significant progress. We refer our readers to [61, 75] for recent updates of the research.

## 10.4 Semidefinite Program

# Chapter 11

# APX Completeness Theory

# Part IV

# Miscellany

# Chapter 12

# Non-Approximable Problems

# Chapter 13

# Exponential Time Algorithms

## 13.1 Satisfiability

## 13.2 Independent set

## 13.3 Parameterized vertex cover

# Bibliography


[1] A. V. Aho, J. E. Hopcropt, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] N. Alon and J. H. Spencer, *The Probabilistic Method*, John Wiley & Sons, Inc., New York, NY, 1992.

[3] K. Appel and W. Haken, Every planar map is four colorable, Part I: Discharging, *Illinois J. Math. 21*, (1977), pp. 429-490.

[4] K. Appel and W. Haken, Every planar map is four colorable, Part I: Reducibility, *Illinois J. Math. 21*, (1977), pp. 491-567.

[5] S. Arora, Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems, *Journal of the ACM 45* (1998), pp. 753-782.

[6] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, Proof verification and the hardness of approximation problems, *Journal of the ACM 45*, (1998), pp. 501-555.

[7] S. Arora and S. Safra, Probabilistic checking of proofs: a new characterization of NP, *Journal of the ACM 45*, (1998), pp. 70-122.

[8] T. Asano, T. Ono, and T. Hirata, Approximation algorithms for the maximum satisfiability problem, *Proc. 5th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science 1097, pp. 100-111, (1997).

[9] G. Ausiello, P. Crescenzi, and M. Protasi, Approximate solution of NP optimization problems, *Theoretical Computer Science 150*, (1995), pp. 1-55.

[10] B. S. BAKER, Approximation algorithms for NP-complete problems on planar graphs, *Journal of ACM 41*, (1994), pp. 153-180.

[11] R. BAR-YEHUDA AND S. EVEN, A local-ratio theorem for approximating the weighted vertex cover problem, *Annals of Discrete Mathematics 25*, (1985), pp. 27-46.

[12] P. BERMAN AND G. SCHNITGER, On the complexity of approximating the independent set problem, *Information and Computation 96*, (1992), pp. 77-94.

[13] P. BERMAN AND V. RAMAIYER, Improved approximations for the Steiner tree problem, *Proc. 3rd Ann. ACM-SIAM Symp. on Discrete Algorithms*, (1992), pp. 325-334.

[14] M. BLUM, R. FLOYD, V. PRATT, R. RIVEST, AND R. TARJAN, Time bounds for selection, *Journal of Computer and System Science 7*, (1973), pp. 448-461.

[15] L. CAI AND J. CHEN, On the amount of nondeterminism and the power of verifying, *SIAM Journal on Computing 26*, pp. 733-750, 1997.

[16] L. CAI AND J. CHEN, On fixed-parameter tractability and approximability of NP-hard optimization problems, *Journal of Computer and System Sciences 54*, pp. 465-474, 1997.

[17] L. CAI, J. CHEN, R. DOWNEY, AND M. FELLOWS, On the structure of parameterized problems in NP, *Information and Computation 123*, (1995), pp. 38-49.

[18] J. CHEN AND D. K. FRIESEN, The complexity of 3-dimensional matching, *Tech. Report*, Dept. Computer Science, Texas A&M University, (1995).

[19] J. CHEN, D. K. FRIESEN, AND H. ZHENG, Tight bound on Johnson's algorithm for maximum satisfiability, *Journal of Computer and System Sciences 58*, pp. 622-640, (1999).

[20] J. CHEN, S. P. KANCHI, AND A. KANEVSKY, A note on approximating graph genus, *Information Processing Letters 61*, pp. 317-322, 1997.

[21] J. CHEN, I. A. KANJ, AND W. JIA, Vertex cover: further observations and further improvements, *Lecture Notes in Computer Science 1665* (WG'99), pp. 313-324, 1999.

[22] J. CHEN AND C.-Y. LEE, General multiprocessor task scheduling, *Naval Research Logistics 46*, pp. 57-74, 1999.

[23] J. CHEN AND A. MIRANDA, A polynomial time approximation scheme for general multiprocessor job scheduling, *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pp. 418-427, 1999.

[24] N. CHRISTOFIDES, Worst-case analysis of a new heuristic for the traveling salesman problem, *Tech. Report*, GSIA, Carnegie-Mellon University, (1976).

[25] K. L. CHUNG, *Elementary Probability Theory with Stochastic Processes*, Pringer-Verlag, New York, NY, 1979.

[26] E. G. COFFMAN, M. R. GAREY, AND D. S. JOHNSON, Approximation algorithms for bin packing – an updated survey, in *Algorithm Design for Computer System Design*, (ed. G. Ausiello, M. Lucertini, and P. Serafini), Springer-Verlag, 1984.

[27] S. A. COOK, The complexity of theorem-proving procedures, *Proc. 3rd Ann. ACM Symp. on Theory of Computing*, (1971), pp. 151-158.

[28] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, McGraw-Hill Book Company, New York, 1992.

[29] P. CRESCENZI AND V. KANN, A compendium of NP optimization problems, *Manuscript*, (1995).

[30] P. CRESCENZI AND A. PANCONESI, Completeness in approximation classes, *Information and Computation 93*, (1991), pp. 241-262.

[31] G. B. DANTZIG, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[32] U. DERIGS, A shortest augmenting path method for solving minimal perfect matching problems, *Networks 11*, (1981), pp. 379-390.

[33] E. A. DINITS, Algorithm for solution of a problem of maximum flow in a network with power estimation, *Soviet Math. Dokl. 11*, (1970), pp. 1277-1280.

[34] J. EDMONDS, Paths, trees and flowers, *Canad. J. Math. 17*, (1965), pp. 449-467.

[35] J. Edmonds and R. M. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, *Journal of ACM 19*, (1972), pp. 248-264.

[36] P. Erdös and J. Selfridge, On a combinatorial game, *Journal of Combinatorial Theory, Series A 14*, (1979), pp. 298-301.

[37] R. Fagin, Generalized first-order spectra and polynomial-time recognizable sets, *SIAM-AMS Proc.*, (1974), pp. 43-73.

[38] U. Feige and M. Goemans, Approximating the value of two prover proof systems, with applications to MAX 2SAT and MAX DICUT, *Proc. 3rd Israel Symp. of Theory and Computing and Systems*, (1995), pp. 182-189.

[39] W. Feller, *An Introduction to Probability Theory and Its Applications*, Vol. I, John Wiley, New York, 1968.

[40] W. Feller, *An Introduction to Probability Theory and Its Applications*, Vol. II, John Wiley, New York, 1968.

[41] W. Fernandez de la Vega and G. S. Lueker, Bin packing can be solved within $1+\epsilon$ in linear time, *Combinatorica 1*, (1981), pp. 349-355.

[42] L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[43] D. K. Friesen, Tighter bounds for the multifit processor scheduling algorithm, *SIAM Journal on Computing 13*, (1984), pp. 170-181.

[44] M. Fürer and B. Raghavacheri, Approximating the minimum-degree spanning tree to within one from the optimal degree, *Proc. of the 3rd ACM-SIAM Symp. on Discrete Algorithms*, (1992), pp. 317-324.

[45] H. N. Gabow, An efficient implementation of Edmonds' algorithm for maximum matching on graphs, *Journal of ACM 23*, (1976), pp. 221-234.

[46] Z. Galil, Efficient algorithms for finding maximum matching in graphs, *Computing Surveys 18*, (1986), pp. 23-38.

[47] M. R. Garey, R. L. Graham, and D. S. Johnson, Some NP-complete geometric problems, *Proc. 8th Ann. ACM Symp. on Theory of Computing*, (1976), pp. 10-22.

[48] M. R. Garey, R. L. Graham, D. S. Johnson, and A. C. Yao, Resource constrained scheduling as generalized bin packing, *Journal of Combinatorial Theory Series A 21*, (1976), pp. 257-298.

[49] M. R. Garey and D. S. Johnson, Strong NP-completeness results: motivation, examples, and implications, *Journal of ACM 25*, (1978), pp. 499-508.

[50] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, San Fransico, CA, 1979.

[51] M. R. Garey, D. S. Johnson, and L. J. Stockmeyer, Some simplified NP-complete graph problems, *Theoretical Computer Science 1*, (1976), pp. 237-267.

[52] M. X. Goemans and D. P. Williamson, New 3/4-approximation algorithms for the maximum satisfiability problem, *SIAM Journal on Disc. Math. 7*, (1994), pp. 656-666.

[53] M. X. Goemans and D. P. Williamson, Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming, *Journal of ACM 42*, (1995), pp. 1115-1145.

[54] A. V. Goldberg, E. Tardos, and R. E. Tarjan, Network flow algorithms, *Tech. Report STAN-CS-89-1252*, Dept. Computer Science, Stanford Univ., (1989).

[55] A. V. Goldberg and R. E. Tarjan, A new approach to the maximum-flow problem, *Journal of ACM 35*, (1988), pp. 921-940.

[56] R. L. Graham, Bounds for certain multiprocessing anomalies, *Bell Systems Technical Journal 45*, (1966), pp. 1563-1581.

[57] M. Grigni, E. Koutsoupias, and C. Papadimitriou, An approximation scheme for planar graph TSP, *Proc. 36st Ann. IEEE Symp. on the Foundation of Computer Science*, (1995), to appear.

[58] J. L. Gross and T. W. Tucker, *Topological Graph Theory*, John Wiley & Sons, New York, 1987.

[59] J. L. Gross and J. Yellen, *Graph Theory and Its Applications*, CRC Press, Boca Raton, 1999.

[60] P. Hansen and B. Jaumard, Algorithms for the maximum satisfiability problem, *Computing 44*, (1990), pp. 279-303.

[61] J. Hastad, Some optimal inapproximability results, *Proc. 28th Annual ACM Symposium on Theory of Computing*, (1997), pp. 1-10.

[62] D. S. Hochbaum, Approximation algorithms for the set covering and vertex cover problems, *SIAM Journal on Computing 3*, (1982), pp. 555-556.

[63] D. S. Hochbaum and D. B. Shmoys, Using dual approximation algorithms for scheduling problems: theoretical and practical results, *Journal of ACM 34*, (1987), pp. 144-162.

[64] D. S. Hochbaum and D. B. Shmoys, A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach, *SIAM Journal on Computing 17*, (1988), pp. 539-551.

[65] D. S. Hochbaum and D. B. Shmoys, A linear-time approximation scheme for scheduling on identical parallel machines, *Information Processing Letters*, (1997), to appear.

[66] I. Holyer, The NP-completeness of edge coloring, *SIAM Journal on Computing 10*, (1981), pp. 718-720.

[67] J. E. Hopcroft and R. M. Karp, A $n^{5/2}$ algorithm for maximum matching in bipartite graphs, *SIAM Journal on Computing 2*, (1973), pp. 225-231.

[68] J. E. Hopcroft and R. E. Tarjan, Efficient planarity testing, *Journal of ACM 21*, (1974), pp. 549-568.

[69] C. A. J. Hurkens and A. Schrijver, On the size of systems of sets every $t$ of which have an SDR, with an application to the worst-case ratio of heuristics for packing problems *SIAM Journal on Discrete Mathematics 2*, (1989), pp. 68-72.

[70] O. H. Ibarra and C. E. Kim, Fast approximation algorithms for the knapsack and sum of subset problems, *Journal of ACM 22*, (1975), pp. 463-468.

[71] D. S. Johnson, Approximation algorithms for combinatorial problems, *Journal of Computer and System Sciences 9*, (1974), pp. 256-278.

[72] D. S. JOHNSON, The NP-completeness column: an ongoing guide, *Journal of Algorithms 13*, (1992), pp. 502-524.

[73] D. S. JOHNSON, A. DEMERS, J. D. ULLMAN, M. R. GAREY, AND R. L. GRAHAM, Worst-case performance bounds for simple one-dimensional packing algorithms, *SIAM Journal on Computing 3*, (1974), pp. 299-325.

[74] V. KANN, Maximum bounded 3-dimensional matching is MAX SNP-complete, *Information Processing Letters 37*, (1991), pp. 27-35.

[75] H. KARLOFF AND U. ZWICK, A 7/8-approximation algorithm for MAX-SAT? *Proc. 38th IEEE Symposium on the Foundation of Computer Science*, (1997), pp. 406-415.

[76] N. KARMAKAR, A new polynomial-time algorithm for linear programming, *Combinatorica 4*, (1984), pp. 373-395.

[77] N. KARMAKAR AND R. M. KARP, An efficient approximation scheme for the one-dimensional bin packing problem, *Proc. 23rd Ann. IEEE Symp. on Foundation of Computer Science*, (1982), pp. 312-320.

[78] D. KARGER, R. MOTWANI, AND G. D. S. RAMKUMAR, On approximating the longest path in a graph, *Lecture Notes in Computer Science 709*, (1993), pp. 421-432.

[79] R. M. KARP AND V. RAMACHANDRAN, Parallel algorithms for shared-memory machines, in *Handbook of Theoretical Computer Science, Volume A. Algorithms and Complexity*, (ed. J. van Leeuwen), The MIT Press/Elsevier, 1990.

[80] A. V. KARZANOV, Determining the maximum flow in the network with the method of preflows, *Soviet Math. Dokl. 15*, (1974), pp. 434-437.

[81] L. G. KHACHIAN, A polynomial algorithm for linear programming, *Doklady Akad. Nauk USSR 244*, (1979), pp. 1093-1096.

[82] S. KHANNA, R. MOTWANI, M. SUDAN, AND U. VAZIRANI, On syntactic versus computational views of approximability, *Proc. 35th Ann. IEEE Symp. on Foundation of Computer Science*, (1994), pp. 819-836.

[83] D. E. KNUTH, *The Art of Computer Programming. Volume I: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

[84] D. E. Knuth, *The Art of Computer Programming. Volume III: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[85] P. G. Kolaitis and M. N. Thakur, Logical definability of NP optimization problems, *Information and Computation 115*, (1994), pp. 321-353.

[86] P. G. Kolaitis and M. N. Thakur, Approximation properties of NP minimization classes, *Journal of Computer and System Sciences 50*, (1995), pp. 391-411.

[87] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart&Winston, 1976.

[88] E. L. Lawler, Fast approximation algorithms for knapsack problems, *Mathematics of Operations Research 4*, (1979), pp. 339-356.

[89] H. W. Lenstra, Integer programming with a fixed number of variables, *Mathematics of Operations Research 8*, (1983), pp. 538-548.

[90] K. J. Lieberherr and E. Specker, Complexity of partial satisfaction, *Journal of ACM 28*, (1981), pp. 411-421.

[91] R. J. Lipton and R. E. Tarjan, A separator theorem for planar graphs, *SIAM J. Appl. Math. 36*, (1979), pp. 177-189.

[92] R. J. Lipton and R. E. Tarjan, Applications of a planar separator theorem, *SIAM Journal on Computing 9*, (1980), pp. 615-627.

[93] C. Lund and M. Yannakakis, On the hardness of approximating minimization problems, *Journal of ACM 41*, (1994), pp. 960-981.

[94] S. Mahajan and H. Ramesh, Derandomizing semidefinite programming based approximation algorithms, *Proc. 36th Ann. IEEE Symp. on the Foundation of Computer Science*, (1995), pp. 162-169.

[95] S. Micali and V. V. Vazirani, An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs, *Proc. 21st Ann. IEEE Symp. on the Foundation of Computer Science*, (1980), pp. 17-27.

[96] B. Monien, How to find long paths efficiently, *Annals of Discrete Mathematics 25*, (1985), pp. 239-254.

[97] R. Motwani, *Lecture Notes on Approximation algorithms*, Dept. of Computer Science, Stanford University, 1995.

[98] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, New York, NY, 1995.

[99] G. L. Nemhauser and L. E. Trotter, Vertex packing: structural properties and algorithms, *Mathematical Programming 8*, (1975), pp. 232-248.

[100] C. H. Papadimitriou, Euclidean TSP is NP-complete, *Theoretical Computer Science 4*, (1977), pp. 237-244.

[101] C. H. Papadimitriou, *Combinatorial Complexity*, Addison-Wesley, Reading, MA, 1993.

[102] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Englewood Cliffs, NJ: Prentice Hall, 1982.

[103] C. H. Papadimitriou and M. Yannakakis, Optimization, approximation, and complexity classes, *Journal of Computer and System Sciences 43*, (1991), pp. 425-440.

[104] C. H. Papadimitriou and M. Yannakakis, The traveling salesman problem with distances one and two, *Mathematics of Operations Research 18*, (1993), pp. 1-11.

[105] C. H. Papadimitriou and M. Yannakakis, On limited nondeterminism and the complexity of the V-C dimension, *Journal of Computer and System Sciences*, (1995), to appear.

[106] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[107] R. C. Prim, Shortest connection networks and some generalizations, *Bell System Technical Journal 36*, (1957), pp. 1389-1401.

[108] N. Robertson, D. Sanders, P. Seymour, and R. Thomas, Efficiently four-coloring planar graphs, *Proc. 28th ACM Symposium on Theory of Computing*, (1996), pp. 571-575.

[109] S. Sahni, Algorithms for scheduling independent tasks, *Journal of ACM 23*, (1976), pp. 116-127.

[110] S. Sahni and T. Gonzalez, P-complete approximation problems, *Journal of ACM 23*, (1976), pp. 555-565.

[111] D. B. SHMOYS, Computing near-optimal solutions to combinatorial optimization problems, *DIMACS Series in Discrete Mathematics*, (1995), to appear.

[112] R. E. TARJAN, *Data structures and network algorithms*, Society for Industrial and Applied Mathematics, 1983.

[113] C. THOMASSEN, The graph genus problem is NP-complete, *Journal of Algorithms*, (1989), pp. 568-576.

[114] V. G. VIZING, On an estimate of the chromatic class of a $p$-graph (in Russian), *Diskret. Analiz 3*, (1964), pp. 23-30.

[115] M. YANNAKAKIS, On the approximation of maximum satisfiability, *Journal of Algorithms 17*, (1994), pp. 475-502.

[116] D. ZUCKERMAN, On unapproximable versions of NP-complete problems, *SIAM Journal on Computing*, (1995), to appear.

# Part V

# Appendix

# Appendix A

# Basic Graph Theory

## Preliminaries

representations: adjacency matrix and adjacency list;
trees, connected components,

## Basic algorithms

depth first search. depth first search number.

breadth first search. For breadth first seach, show that every in the graph connects two vertices either at the same level or at adjacent levels.

## Euler graphs

**Definition A.1** An undirected connected graph $G$ is an *Euler graph* if there is a closed walk in $G$ that traverses each edge of $G$ exactly once.

**Theorem A.1** *An undirected connected graph $G$ is an Euler graph if and only if every vertex of $G$ has an even degree.*

PROOF. Suppose that $G$ is an Euler graph. Let $W$ be a closed walk in $G$ that traverses each edge of $G$ exactly once.

Let $v$ be a vertex of $G$. Since $W$ is a closed walk, each time $W$ enters the vertex $v$ from an edge, $W$ must leave the vertex $v$ by another edge incident on $v$. Therefore, each edge incident on $v$ that is an "incoming" edge for $W$ must be paired with an edge incident on $v$ that is an "outgoing" edge for

$W$. Since $W$ traverses each edge exactly once, we conclude that the number of edges incident on $v$, i.e., the degree of $v$, is even.

Conversely, suppose that all vertices of the graph $G$ have even degree. We prove the theorem by induction on the number of edges in $G$. The minimum such a graph $G$ in which all vertices have even degree consists of two vertices connected by two (multiple) edges. This graph is clearly an Euler graph.

Now suppose that $G$ has more than two edges. Let $v_0$ be any vertex of $G$. We construct a maximal walk $W_0$ starting from the vertex $v_0$. That is, starting from $v_0$, on each vertex if there is an unused edge, then we extend $W_0$ along that edge (if there are more than one such edge, we pick any one). The process stops when we hit a vertex $u$ on which there is no unused incident edge. We claim that the ending vertex $u$ must be the starting vertex $v_0$. In fact, for each interior vertex $w$ in the walk $W_0$, each time $W_0$ passes through, $W_0$ uses one edge to enter $w$ and uses another edge to leave $w$. Therefore, if the process stops at $u$ and $u \neq v_0$, then the walk $W_0$ has only used an odd number of edges incident on $u$. This contradicts our assumption that the vertex $u$ is of even degree. This proves the claim. Consequently, the walk $W_0$ is a closed walk.

The closed walk $W_0$ can also be regarded as a graph. By the definition, the graph $W_0$ itself is an Euler graph. According to the proof for the first part of this theorem, all vertices of the graph $W_0$ have even degree. Now removing all edges in the walk $W_0$ from the graph $G$ results in a graph $G_0 = G - W_0$. The graph $G_0$ may not be connected. However, all vertices of $G_0$ must have an even degree because each vertex of the graphs $G$ and $W_0$ has an even degree.

Let $C_1$, $C_2$, …, $C_h$ be the connected components of the graph $G_0$. By the inductive hypothesis, each connected component $C_i$ is an Euler graph. Let $W_i$ be a closed walk in $C_i$ that traverses each edge of $C_i$ exactly once, for $i = 1, \ldots, h$. Moreover, for each $i$, the closed walk $W_0$ contains at least one vertex $v_i$ in the connected component $C_i$ (if $W_0$ does not contain any vertex from $C_i$, then the vertices of $C_i$ have no connection to the vertices in the walk $W_0$ in the original graph $G$, this contradicts the assumption that the graph $G$ is connected).

Therefore, it is easy to insert each closed walk $W_i$ into the closed walk $W_0$ (by replacing any vertex occurrence of $v_i$ in $W_0$ by the list $W_i$, where $W_i$ is given by beginning and ending with $v_i$), for all $i = 1, \ldots, h$. This forms a closed walk $W$ for the original graph $G$ such that the walk $W$ traverses each edge of $G$ exactly once. Thus, the graph $G$ is an Euler graph. $\square$

The proof of Theorem A.1 suggests an algorithm that constructs a closed walk $W$ for an Euler graph $G$ such that the walk $W$ traverses each edge of $G$ exactly once. This walk will be called an *Euler tour*. By a careful implementation, one can make this algorithm run in linear time. We leave the detailed implementation to the reader. Instead, we state this result without a proof as follows.

**Theorem A.2** *There is an algorithm that, given an Euler graph, constructs an Euler tour in linear time.*

PROOF. A formal proof is needed here. $\square$

## Planar graphs

# Appendix B

# Basic Linear Algebra

vector, matrix, matrix multiplication, linearly independent, singular and nonsingular matrix, solving a system of linear equations. show that if $v_1$, …, $v_n$ are $n$ linearly independent vector in the $n$-dimensional space, and $v_0$ is an $n$-dimensional vector such that $v_1$ can be represented by a linear combination of $v_0$, $v_2$, …, $v_n$, then $v_0$, $v_2$, …, $v_n$ are linearly independent.

Cramer's Rule for solving a linear system.

determinant of a matrix.

# Appendix C

# Basic Probability Theory

In this appendix, we review some basic facts in probability theory that are most related to the discussions we give in other chapters in this book. For a more comprehensive study of probability theory, readers are referred to standard probability theory textbooks, such as Chung [25] and Feller [39, 40].

## Basic definitions

A probabilistic statement must be based on an underlying *probabilistic space*, which consists of a *sample space* and a *probabilistic measure* imposed on the sample space. A sample space can be an arbitrary set. Each element in the sample space is called a *sample point*, and each subset of the sample space is referred to as an *event*. To avoid a diverting complication and to make this review more intuitive, we will always assume that the sample space is countable (i.e., either finite or countably infinite).

**Definition C.1** A *probability measure* on a sample space $\Omega$ is a function $P(\cdot)$ on the events of $\Omega$ satisfying the following three axioms:

(1) For every event $A \subseteq \Omega$, we have $0 \le P(A) \le 1$.

(2) $P(\Omega) = 1$.

(3) For any two disjoint events $A$ and $B$ in $\Omega$, i.e., $A \cap B = \emptyset$, we have $P(A \cup B) = P(A) + P(B)$.

A *probabilistic space*, $(\Omega, P)$, consists of a sample space $\Omega$ with a probability measure $P$ defined on $\Omega$.

Intuitively, the sample space $\Omega$ represents the set of all possible outcomes in a probabilistic experiment, and an event specifies a collection of the outcomes that share certain properties. The probability value $P(A)$ measures the "likelihood" of the event $A$.

From the definition, a number of simple facts can be deduced. We list these facts as follows and leave the proofs to the reader.

**Proposition C.1** *Let $(\Omega, P)$ be a probabilistic space, then*

(4) *For any two events $A$ and $B$ such that $A \subseteq B$, we have* $P(A) \leq P(B)$ *and* $P(B - A) = P(B) - P(A)$.

(5) *For a set $\mathcal{S} = \{A_1, \ldots, A_n\}$ of mutually disjoint events, we have* $P(\cup_{A_i \in \mathcal{S}} A_i) = \sum_{A_i \in \mathcal{S}} P(A_i)$.

(6) *For a set $\mathcal{S} = \{A_1, \ldots, A_n\}$ of arbitrary events, we have* $P(\cup_{A_i \in \mathcal{S}} A_i) \leq \sum_{A_i \in \mathcal{S}} P(A_i)$.

Properties (5) and (6) in Proposition C.1 also hold when the set $\mathcal{S}$ is countably infinite. Using these facts, we can interpret the probability value $P(A)$ for an event $A$ in a more intuitive way. Assuming the sample space $\Omega$ is countable, then the value $P(A)$ can be expressed by

$$P(A) = \frac{P(A)}{P(\Omega)} = \frac{P(\cup_{q \in A}\{q\})}{P(\cup_{q \in \Omega}\{q\})} = \frac{\sum_{q \in A} P(q)}{\sum_{q \in \Omega} P(q)}$$

here we have written $P(q)$ for $P(\{q\})$ for a sample point $q$ in the sample space $\Omega$. Therefore, the probability $P(A)$ is essentially the "proportion" of the event $A$ in the sample space $\Omega$ when each sample point $q$ is given a "weight" $P(q)$.

## Conditional probability

Let $(\Omega, P)$ be a probabilistic space. Let $A$ and $B$ be two events in the sample space $\Omega$. Suppose that we already know that the outcome is in the set $B$ and we are interested in knowing under this condition, what is the probability that the outcome is also in the set $A$. Thus, here we have switched our attention from the original sample space $\Omega$ to the set $B$, and are asking the proportion of the set $A \cap B$ with respect to the set $B$. According to the remark following Proposition C.1, this proportion is given by the value $(\sum_{q \in A \cap B} P(q))/(\sum_{q \in B} P(q)) = P(A \cap B)/P(B)$. This motivates the following definition.

**Definition C.2** Assuming $P(B) > 0$ for an event $B$. The *conditional probability* of event $A$ relative to $B$ is denoted by $P(A|B)$ and defined by

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Other phrases for the relativity condition "relative to $B$", such as "given $B$", "knowing $B$", or "under the hypothesis $B$" may also be used.

**Proposition C.2** *Let $A_1$, ..., $A_n$ be a partition of the sample space $\Omega$, that is, $\cup_{i=1}^n A_i = \Omega$ and $A_i \cap A_j = \emptyset$ for $i \neq j$. Then for any event $B$, we have*

(1) $P(B) = \sum_{i=1}^n P(B|A_i)P(A_i)$.

(2) $P(A_k|B) = (P(A_k)P(B|A_k))/(\sum_{i=1}^n P(B|A_i)P(A_i))$ *for all* $A_k$.

PROOF. We have

$$\begin{aligned} P(B) &= P(B \cap \Omega) = P(B \cap (\cup_{i=1}^n A_i)) = P(\cup_{i=1}^n (B \cap A_i)) \\ &= \sum_{i=1}^n P(B \cap A_i) = \sum_{i=1}^n P(B|A_i)P(A_i) \end{aligned}$$

The fourth equality follows from Proposition C.1(5) and the fact that $A_1$, ..., $A_n$ are mutually disjoint implies the mutual disjointness of the sets $B \cap A_1$, ..., $B \cap A_n$, and the last equality follows from the definition of the conditional probability $P(B|A_i)$. This proves (1).

To prove (2), note that by the definitions of the conditional probabilities $P(A_k|B)$ and $P(B|A_k)$ we have

$$P(B \cap A_k) = P(B)P(A_k|B) = P(A_k)P(B|A_k)$$

The second equality gives $P(A_k|B) = P(A_k)P(B|A_k)/P(B)$. Now using the expression of $P(B)$ in (1) gives (2). $\square$

Two events $A$ and $B$ are *independent* if $P(A \cap B) = P(A)P(B)$. More general, $n$ events $A_1$, ..., $A_n$ are *independent* if for any subset $\mathcal{S}$ of $\{A_1, \ldots, A_n\}$, we have

$$P(\bigcap_{A_i \in \mathcal{S}} A_i) = \prod_{A_i \in \mathcal{S}} P(A_i)$$

The independence of two events $A$ and $B$ indicates that the condition $B$ has no effect on the probability of event $A$. This can be well-explained using the definition of conditional probability: for two independent events $A$ and $B$, we have

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A)P(B)}{P(B)} = P(A)$$

## Random variables

**Definition C.3** A real-valued function $X$ over a sample space $\Omega$ is called a *random variable* on $\Omega$.

Note that a random variable can be defined on a sample space even before any probability measure is mentioned for the sample space. For a probabilistic space $(\Omega, P)$ and a random variable $X$ defined on $\Omega$, since the random variable $X$ is always associated with the sample space $\Omega$, we can write $P(\{ q \mid q \in \Omega \text{ and } X(q) \leq y\})$ in a more compact form $P(X \leq y)$ without any ambiguity. Here $X \leq y$ specifies a condition on the random variable $X$, or equivalently it can be regarded as to define the event of sample points $q$ that satisfy the condition $X(q) \leq y$. More generally, suppose that $X_1, \ldots, X_n$ are random variables defined on the same probabilistic space $(\Omega, P)$, and that $C_1(X_1), \ldots, C_n(X_n)$ are arbitrary conditions on the random variables $X_1, \ldots, X_n$, respectively, then we write the probability

$$P(\{ q \mid q \in \Omega \text{ and } C_1(X_1(q)) \text{ and } \cdots \text{ and } C_n(X_n(q))\})$$

in a more compact form $P(C_1(X_1), \ldots, C_n(X_n))$.

The function

$$F_X(y) = P(X \leq y)$$

is called the *distribution function* of the random variable $X$.

Starting with some random variables, we can make new random variables by operating on them in various ways. In particular, if $X$ and $Y$ are two random variables on a probabilistic space $(\Omega, P)$, then so is $X + Y$, where $X + Y$ is defined by $(X + Y)(q) = X(q) + Y(q)$ for all sample points $q$ in $\Omega$. Similarly, $aX - bY$ and $XY$ are also random variables over $(\Omega, P)$, where $a$ and $b$ are arbitrary real numbers.

**Definition C.4** Two random variables $X_1$ and $X_2$ are said to be *independent* if for any two real numbers $y_1$ and $y_2$, we have

$$P(X_1 = y_1, X_2 = y_2) = P(X_1 = y_1) \cdot P(X_2 = y_2)$$

## Expectation and variance

The definition of the expectation of a random variable is motivated by the intuitive notion of the "average value" of the random variable over all sample points in the sample space.

**Definition C.5** The *expectation* of a random variable $X$ on a probabilistic space $(\Omega, P)$ is written as $E(X)$ and defined by (recall that we assume the sample space $\Omega$ is countable)

$$E(X) = \sum_{q \in \Omega} X(q)P(q)$$

provided that the series $\sum_{q\in\Omega} X(q)P(q)$ is absolutely convergent (that is, $\sum_{q\in\Omega} |X(q)P(q)| = \sum_{q\in\Omega} |X(q)|P(q) < \infty$). In this case we say that the expectation of $X$ *exists.*

The *kth moment* of a random variable $X$ is defined to be $E(X^k)$. In particular, the first moment of $X$ is simply the expectation of $X$.

**Definition C.6** The *variant* of a random variable $X$, denoted $\sigma^2(X)$, is defined to be

$$\sigma^2(X) = E((X - E(X))^2)$$

Intuitively, for any sample point $q$, $(X - E(X))(q)$ is the deviation of the value $X(q)$ from the "average value" $E(X)$ of $X$, which can take both positive and negative values. Thus, simply averaging over all these deviations may not be informative if we are interested in the *magnitude* of the deviations. The square of this deviation, $(X - E(X))^2$, still reflects, in a looser way, this deviation and always gives a nonnegative value. Therefore, the expectation $\sigma^2(X) = E((X - E(X))^2)$ is a indicator of the average of the magnitudes of this deviation.

# Appendix D

# List of Optimization Problems

Minimum Spanning Tree (MSP)

$I_Q$: the set of all weighted 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

Shortest Path

$I_Q$: the set of all weighted graphs $G$ with two specified vertices $u$ and $v$ in $G$

$S_Q$: $S_Q(G)$ is the set of all paths connecting $u$ and $v$ in $G$

$f_Q$: $f_Q(G, u, v, P)$ is the length of the path $P$ (measured by the weight of edges) connecting $u$ and $v$ in $G$

$opt_Q$: min

Knapsack

$I_Q$: the set of tuples $T = \{s_1, \ldots, s_n; v_1, \ldots, v_n; B\}$, where $s_i$ and $v_i$ are for the size and value of the $i$th 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

MAKESPAN

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

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

$f_Q$: $f_Q(T, P)$ is equal to the processing time of the largest subset in the partition $P$, that is,

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

$opt_Q$: min

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

$S_Q$: $S_Q(T)$ 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 $S$

$opt_Q$: min

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 $S$

$opt_Q$: max

# Contents

# List of Figures