Wrocław University of Technology

Business Information Systems

Adam Kasperski

DISCRETE OPTIMIZATION AND NETWORK FLOWS

Wrocław 2011

Reviewer: Jacek Mercik

ISBN 978-83-62100-00-5

Published by PRINTPAP Łódź, www.printpap.pl

Preface

Discrete optimization is an important area of operations research and applied computer science. Discrete optimization models have many applications in situations, where rational decisions have to be made. A wide and important field of discrete optimization contains the so-called network problems. A network can be seen as a set of points together with some connections between them. Networks are used to model physical systems for example roads, electrical lines, computer networks, ordering of tasks etc. They have many applications in management, industry, defense, communication, logistics, health care, ecology etc.

The aim of this book is to introduce some basic discrete optimization problems defined on networks and present some methods of solving them. Most of the content is presented at an elementary level and requires only some elementary mathematics. A reader not familiar with graph theory should first read Appendix A, where some basic notion used for networks is described. More advanced topics on linear programing and computational complexity are presented in Appendix B and Appendix C, but the material presented there is not necessary to understand the rest of the book. All the sections contain a number of examples, which illustrate the models and algorithms introduced. The sections describing particular problems also contain some applications of these problems in management. This book does not consider implementation of the algorithms presented. Computer programmers can find the material in the extensive literature devoted to algorithms and data structures.

This book is composed of three chapters and three appendices. In Chapter 1 the class of discrete optimization problems discussed in this book is introduced. These optimization problems are solved by using algorithms. So, in Chapter 1 the notion of an algorithm is described. The concept of an efficient algorithm, i.e. one that is able to solve large problems, is also explained. In Chapter 2 the class of network flow problems is discussed. This class contains some basic problems such as the shortest path, the maximum flow, the minimum cost assignment etc. Furthermore, all the problems described in this chapter can be solved efficiently. In particular, the network simplex algorithm is described, which efficiently solves the general minimum cost flow problem. Unfortunately, many important problems associated with networks are computationally hard. This means that no efficient algorithms are known for them. In Appendix C some elements of the theory of NP-completeness are described. This theory allows us to classify computational problems into hard and easy. Some general methods

of dealing with hard problems are described in Chapter 3. These methods include mathematical programming approach, branch and bound algorithm, dynamic programming and approximation algorithms (in particular, the local search technique).

There is extensive literature about discrete optimization and network flows. The book by Ahuja et al. [3] is strongly recommended. It contains an excellent description of network flow problems. Also, the books by Lawler [34], Bazaraa et al. [7], Ford and Fulkerson [22] contain the material presented in this book. Discrete and combinatorial optimization problems are discussed in the books by Garfinkel and Nemhauser [24], Papadimitriou and Steiglitz [41], Schrijver [44] and Chen et. al. [11]. The books on discrete mathematics [43], graph theory [17] and computational complexity [40] are recommended to learn more about the material discussed in this book. Additional literature will be given at the end of each section.

Adam Kasperski
Institute of Industrial Engineering and Management
Wrocław University of Technology
Email: adam.kasperski@pwr.wroc.pl
Wrocław 2011

Contents

Chapter 1

Introduction

1.1 What is a discrete optimization problem?

In many real life applications, we wish to find an object, which minimizes a given cost (or maximizes a given reward). Before we formally introduce the problem, let us show some representative examples.

Example 1. Suppose that we have a map with some cities connected by roads. We know the length of each road. A small map illustrating this example is shown in Figure 1.1.

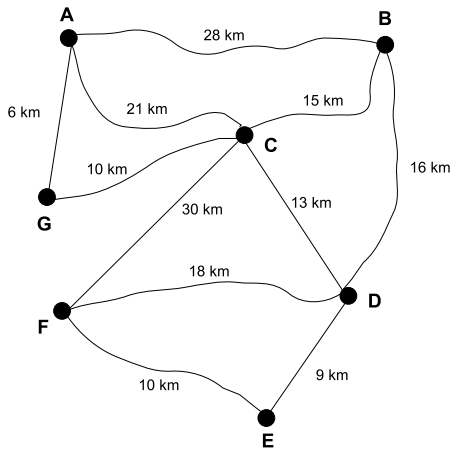


Figure 1.1: The map for Example 1.

Consider the following problems:

1. We wish to travel from **B** to **F**. What is the shortest route between these two cities?

2. We wish to connect all the cities by telephone lines and the lines must be placed along roads. Where should we place them to use the minimum length of lines?

3. We are in the city **A**. We must visit all the cities exactly once and return to **A**. What is the shortest tour which satisfies our requirements?

In each of the three problems we wish to find a different object in the given map. The answer to the first problem is clearly the route **B**–**D**–**F**, whose total length is 34 km. The answer to the second problem is the telephone network shown in Figure 1.2. Note that this network connects all the cities and its total length is equal to 63 km. Finally, the answer to the last problem is shown in Figure 1.3. This figure shows a closed tour that visits all cities exactly once and the length of this tour is 109 km. Problem 1 is an example of the *shortest path* problem, problem 2 is an example of the *minimum spanning tree* problem and problem 3 is an example of the *traveling salesperson* problem. All three problems will be explored later in this book.

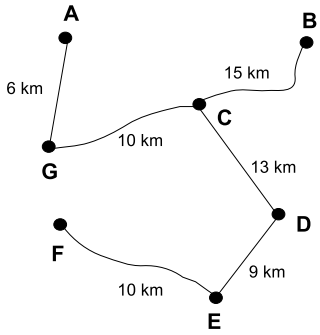


Figure 1.2: The shortest telephone network connecting all cities.

Example 2 (knapsack problem) There are n items in a store, each with a weight w_i and a value p_i, for $i = 1, \ldots, n$. We wish to determine, which items to take so that their total weight is not greater than a given capacity W and their total value is as large as possible. For example, consider the set of 5 items with weights 3,6,1,5,7 and values 2,4,1,4,7, respectively. We may take items whose total weight does not exceed 10. Which items should we take? One can take, for instance, items 1, 3 and 4 whose total weight equals 9 and total value equals 7. On the other hand, items 4 and 5 cannot be both taken, because their total weight exceeds 10. It turns out that the best choice is to take items 1 and 5, which yields a total value of 9.

Example 3. There are n jobs $J_1, J_2, \ldots, J_n$, which must be processed on a single machine. Job J_i has a processing time equal to p_i. We assume that the processing of any job cannot be interrupted and the next job is processed just

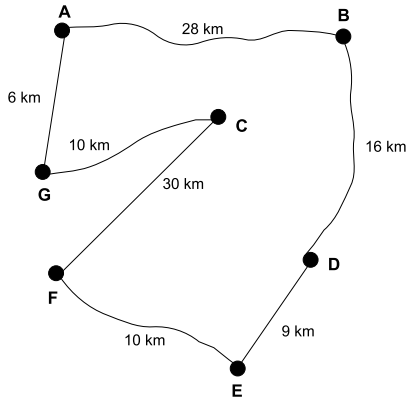


Figure 1.3: The shortest tour that visits all cities.

after the previous is finished. Consider, for example, five jobs $J_1, J_2, \ldots, J_5$ with processing times 2,4,1,5,3. These jobs can be processed, for instance, in order $(J_2, J_1, J_5, J_4, J_3)$. The completion time C_1 of job J_1 in this ordering equals 6. Similarly, $C_2 = 4$, $C_3 = 15$, $C_4 = 14$ and $C_5 = 9$ and the sum of completion times, called the *total flow time* is equal to $C_1 + \cdots + C_5 = 48$. We would like to find an ordering of jobs for which the total flow time is minimal. In this example, the best ordering is $(J_3, J_1, J_5, J_2, J_4)$, for which the total flow time equals 35.

Example 4. Consider the map shown in Figure 1.4. This map is similar to that from Example 1, but now some additional data are specified. In the city **B** there is a store, in which there are 20 units of some product. This product must be delivered to two shops. The first shop is located in the city **G** and requires 15 units of the product and the second shop is located in the city **E** and requires 5 units of the product. Now, for each road we have two numbers: the first is the unit transportation cost and the second is the maximal number of units which can be sent using this road. For example, we can send at most 3 units (that is 0,1,2 or 3 units) from **B** to **A** and sending 1 unit costs us \$2. Now the problem is how to send the product from the store to both shops in the cheapest possible way. The solution to this problem is shown in Figure 1.5. According to this solution, we should send 3 units from **B** to **A**, 10 units from **B** to **D**, etc. Observe that this solution is feasible because no road carries more units than its capacity. Also, the store sends exactly 20 units and every shop receives the required amount of the product.

We have shown 4 different problems. In all of them three basic elements can be distinguished. The first element consists of *input data*, that is all the information we know and which is necessary to solve the problem. In Examples 1 and 4, the input data consist of a map with some numbers associated with roads and cities. In Example 2 the input data consist of a set of items together

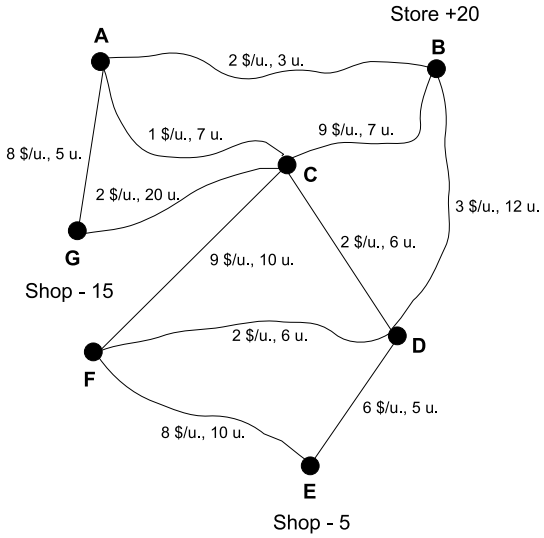


Figure 1.4: The map for Example 4.

with their weights and values and a limit W. Finally, in Example 3 the input data consist of a set of jobs together with their processing times. A particular realization of input data will be called an *instance*. For example, the map shown in Figure 1.1 is an instance of the shortest path problem. The second common element in all the examples is the set of *solutions*. A *solution* is an object which we would like to compute. In Example 1, any route from **B** to **F** is a solution and the set of solutions contains all such routes. One of them is the route **B**–**D**–**F** but there are many others, for example **B**–**A**–**C**–**F**. Similarly, the set of solutions may consist of all the possible telephone networks or all the possible closed tours visiting every city exactly once. In Example 2, a solution is any subset of the items, whose total weight does not exceed W. For the instance shown, the subset of items $\{1, 3, 4\}$ is a solution, but so are subsets $\{1, 5\}$, $\{2, 3\}$, $\{4\}$, etc. In Example 3, a solution is any ordering of jobs such as $(J_2, J_1, J_5, J_4, J_3)$, $(J_3, J_5, J_1, J_4, J_2)$ etc. A description of a solution in Example 4 is more complex. At this moment, we can informally define a solution as a precise transportation plan, such as the one shown in Figure 1.5. The last common element in all these problems is a *cost function*. Namely, we must be able to evaluate the total cost of any solution.

We can now formally define an optimization problem. An *optimization problem* Π consists of:

1. A set D_Π of input data; each $I \in D_\Pi$ is called an *instance*.
2. A set of solutions $sol(I)$ for a given instance $I \in D_\pi$.
3. A function $f(x)$ denoting the cost of solution $x \in sol(I)$.

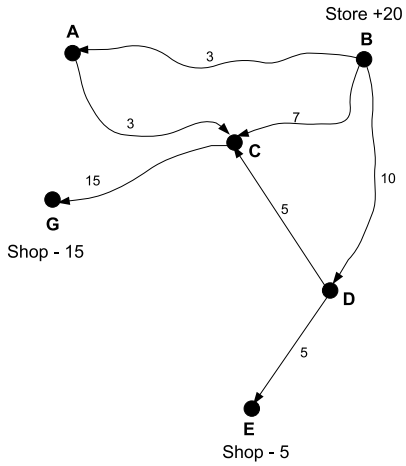


Figure 1.5: A transportation plan.

For a given instance $I \in D_\Pi$ we seek an *optimal solution* $x^* \in sol(I)$, which minimizes (or maximizes) the function $f(x)$, that is

$$f(x^*) = \min(\max)\{f(x) : x \in sol(I)\}.$$

Let us go back again to Example 1. In this example, D_Π is the set of all possible maps with specified road lengths. An instance $I \in D_\Pi$ is a particular map, like the one shown in Figure 1.1. The set $sol(I)$ contains all routes between two specified cities in a given map I. Finally, $f(x)$ is the length of the route $x \in sol(I)$. A similar interpretation can be provided for all the remaining examples and we leave this as an exercise.

Problem Π is called a *discrete optimization problem* if it can be formulated as the following mathematical programming problem:

$$\begin{array}{lll} \min(\max) & f(x_1, x_2, \ldots, x_n) & \\ & g_i(x_1, x_2, \ldots, x_n) = (\leq) b_i & i = 1, \ldots, m \\ & x_j \text{ integer} & j \in D \end{array} \tag{1.1}$$

In this formulation, every solution x is described by a vector of *decision variables* $(x_1, x_2, \ldots, x_n)$, where some variables, those whose indices belong to the set $D \subseteq \{1, \ldots, n\}$, must take integer values. The problem consists of an *objective function*, which is minimized or maximized and a set of m *constraints*. The parameters (constants) appearing in (1.1) describe an instance I of the problem and the constraints of (1.1) describe the solution set $sol(I)$. Finally, the objective function, $f(x_1, \ldots, x_n)$, expresses the cost of a solution. In this book we will consider the class of problems for which the objective function and all the constraints are *linear*. This class includes most of the important problems arising in practice.

Example 5. Consider the knapsack problem from Example 2. This problem can be easily formulated in the form of (1.1). We introduce a variable x_j for each item $j = 1, \ldots, n$. The variable x_j takes the value 1 if we take item j and takes the value 0 otherwise. Hence $0 \leq x_j \leq 1$ and x_j is an integer, which we can concisely denote as $x_j \in \{0, 1\}$. The total value of a solution can be now computed as $\sum_{j=1}^{n} p_j x_j$ and this quantity should be maximized. We do not want to exceed the capacity W, so the constraint $\sum_{j=1}^{n} w_j x_j \leq W$ must be satisfied. Hence the model takes the following form:

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} p_j x_j \\ & \sum_{j=1}^{n} w_j x_j \leq W \\ & x_j \in \{0, 1\} \qquad j = 1, \ldots, n \end{aligned}$$

1.2 Algorithms and Complexity

What does it mean to solve an optimization problem? Is it possible to get an optimal solution for any such problem? If so, how large problems can be solved? In this section we provide some answers to these questions. If we are asked to find a shortest route from **B** to **F** on the map shown in Figure 1.1, then we probably will not have any problems with providing the correct answer. Consider, however, the whole map of some country containing thousands of cities and roads. Then finding a shortest tour between two cities becomes a highly nontrivial task and, for very large maps computer software must be used. But before an optimal tour is computed, we must design a precise, step by step method of solving the problem, namely we must provide an *algorithm* for it.

We can imagine an algorithm as a program written in some computer programming language such as C or Pascal. Such an algorithm consists of a *description of the input*, a *description of the output* and a *body*, which consists of a set of instructions permitted in a programming language (see Fgure 1.6). In our case, the input to the algorithm is any instance I of an optimization problem and the output is an optimal solution to this problem. The finite set of instructions contained in the body tells us precisely, step by step, how this optimal solution can be computed from the input. Observe that each step may contain some elementary instructions in a computer language or a call of another algorithm.

The first problem that arises is encoding an instance I. For the problems shown in Examples 2 and 3, this is not difficult. Everyone familiar with programming languages can easily encode an instance of the knapsack problem as two numbers n, W and two tables containing the weights and values of items. Encoding an instance of the scheduling problem from Example 3 is also easy. However, encoding such an instance as a map is more complex and, at this point, you may read Appendix A to learn how to do this.

```
ALGORITHM A
INPUT: Description of input data
OUTPUT: Description of output
1:  Step 1 (elementary instructions)
2:  Step 2 (call of another algorithm)
3:  Step 3
...
```

Figure 1.6: The outline of an algorithm.

Every encoded instance I of an optimization problem is stored in computer memory as a sequence of bits. Therefore, it is natural to define the size of an instance I as the number of bits required to store it. This is, however, not practical and in computer science the size of an instance is typically measured by providing some significant parameters describing it. For example, the size of a collection of 32-bit integers $a_1, a_2, \ldots, a_n$ equals n. This collection requires $32n$ bits to be stored. However, as we will see, constants such as 32 can be omitted. We must, however, be careful if the size of the numbers in the collection is not specified. Then, the size of an instance must be described by two numbers, namely n and $\log a$, where $a = \max\{a_1, \ldots, a_n\}$. Notice that $\log a$ is the number of bits required to store the integer number a.

It is clear that larger problems require more time to be solved. We now define the running time of an algorithm.

Definition 1 *We say that an algorithm runs in $f(n)$ time if the number of elementary steps performed for every instance of size n is at most $f(n)$.*

There are several important elements in this definition which should be well understood. First, the running time of an algorithm is a function of its input size n. There may be many instances of a fixed size n, say $n = 100$. For one such instance the algorithm may perform 10 elementary steps, but the number of steps for another instance may be equal to 20. It is important that $f(n)$ is the maximal number of steps performed over all instances of size n. So, the running time is a pessimistic measure describing the behavior of the algorithm in the worst case. It is also important that we measure the number of elementary steps, so we must be careful if another algorithm is called as a subroutine at some step. Having such a function $f(n)$, we can estimate the running time of an algorithm by assuming that every elementary step takes some constant time, say Δt, which depends on the computer used. The running time is then estimated as $\Delta t f(n)$.

Example 1. Consider the following, very simple problem. We have a collection of n numbers $a_1, a_2, \ldots, a_n$ of a fixed size, say 32-bit integers. We wish to find the maximal number in this collection. We can easily encode any instance of this problem as n and the table $A = [a_1, a_2, \ldots, a_n]$. The problem can be then solved by using the algorithm shown in Figure 1.7.

```
ALGORITHM MAX
INPUT: Array A = [a_1, ..., a_n] of 32-bit integers and n.
OTUTPUT: The largest number in A
1:  max := a_1
2:  for i := 2 to n
3:      if a_i > max then max := a_i
4:  next i
5:  return max
```

Figure 1.7: Algorithm for computing the largest number in a given collection.

It is obvious that the algorithm correctly solves the problem, that is it returns the right answer for every valid instance. Each step of this algorithm contains some elementary instructions, so every step is elementary. The size of the input is equal to n. For a fixed n the algorithm performs step 1 and step 5 exactly once. Furthermore, it performs exactly $3(n-1)$ steps in the loop 2-4. In consequence, the running time of the algorithm is $f(n) = 3n - 1$.

Analyzing the above example, we can conclude that the obtained running time $f(n) = 3n - 1$ seems to be too detailed. In particular, it depends on the way in which the pseudo code of the algorithm is presented. For instance, if we split step 3 into two steps by moving the instruction $max := a_i$ to the new line, then the loop would contain 4 steps and the running time would be $f(n) = 4n - 2$. We thus can see that the constants in $f(n)$ should be omitted and the estimation of the running time should not depend on the presentation of the pseudo code or implementation details. This can be done by using the *big O notation*, which is defined as follows:

Definition 2 *Let $f(n)$ and $g(n)$ be two functions, where $n \in N$. We say that $f(n) = O(g(n))$ if there exist two constants c and n_0 such that $f(n) \leq cg(n)$ for all $n > n_0$.*

The notation $f(n) = O(g(n))$ means that function $f(n)$ does not grow faster than $g(n)$ up to some constants c and n_0.

Example 2. We will show that $3n + 7 = O(n)$. According to the definition, we must find constants c and n_0 such that $3n + 7 \leq cn$ for all $n > n_0$. We can rewrite this as $(c-3)n \geq 7$ for $n > n_0$ and choose, for example, $c = 4$ and $n_0 = 7$.

It is not difficult to show that if $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0$ is a polynomial, then $f(n) = O(n^k)$. So, the polynomial $f(n)$ grows as fast as n^k up to some constant factor. This fact is very useful in the analysis of the running time of an algorithm. The constants, such as those obtained in Example 1, depend on the presentation of the pseudo code, implementation details and language used to implement the algorithm. Notice, however, that both $3n-1 = O(n)$ and $4n-2 = O(n)$. We thus can say that the algorithm runs in $O(n)$ time,

regardless of the details. We can also say that the problem from Example 1 can be solved in $O(n)$ time. It can be shown that $2^n \neq O(n^k)$ for any $k > 0$. This means that the exponential function grows faster than any polynomial.

An algorithm that runs in $O(n)$ time is obviously very fast. Its running time grows linearly with the problem size, which allows us to solve very large problems. In particular, we can find the maximal number in a collection containing millions of numbers. We can say that this problem can be solved efficiently. Can every optimization problem be solved efficiently? For example, is it possible to solve efficiently the traveling salesperson problem with a million cities (Example 1 in Section 1.1)? Let us start by observing that it is not difficult to construct a correct algorithm for all the sample optimization problems shown in the previous section. This follows from the fact that for every instance I the set of solutions $sol(I)$ is finite. Given an instance I, we can simply enumerate all the solutions from $sol(I)$, compute their costs and output the best one. Such an algorithm is called a *brute force* method. We will investigate now how efficient such an algorithm can be.

Consider an instance of the traveling salesperson problem. We can measure the size of this instance by the number of cities, denoted by n. Each tour can be represented as a permutation of the n cities. This permutation simply says in which order the cities are visited. Of course, not every permutation represents a solution because there may be no road between two neighboring cities in this permutation. We can, however, assume that such an infeasible permutation has a very large cost and it cannot represent an optimal tour. It is well known that the number of permutations is $n! = 1 * 2 * 3 * \cdots * n$. Suppose that we can enumerate one permutation and compute its cost within 10^{-6} seconds. What is the time required to compute the best permutation using brute force? Clearly, this time is about $10^{-6}n!$. Table 1.1 shows the times required to solve the problem for various values of n.

n	$10^{-6}n!$
10	3.6 s.
12	8 min.
15	363 h.
18	203 years
21	1 620 000 years

Table 1.1: Estimation of the running time of a brute force algorithm for the traveling salesperson problem.

The results shown in Table 1.1 clearly demonstrate that the brute force algorithm is useless. One can apply this algorithm only for very small instances, where the number of cities does not exceed 15.

What about other problems? Consider the knapsack problem. We can measure the size of an instance of this problem as the number of items n. The brute force algorithm would explore all the possible subsets of the items. For

each such subset it computes its cost and checks whether the capacity W is not exceeded (if so, then the subset is rejected). We can again assume that every particular subset can be checked within a small time, say 10^{-6} s. The number of subsets of an n-element set equals 2^n. In consequence, the brute force runs in a time of $2^n 10^{-6}$ s. Table 1.2 shows the times required to solve the problem for different values of n. Again, we can see that brute force is useless for quite small instances.

n	$10^{-6}2^n$
10	0.001 s.
20	1.048 s.
50	35.7 years
100	10^{16} years

Table 1.2: Estimation of the running time of a brute force algorithm for the knapsack problem.

We thus can see that algorithms running in $O(n!)$ or $O(2^n)$ time are efficient only for very small n. On the other hand, an $O(n)$ algorithm runs in reasonable time, even for very large n. We now provide a distinction between efficient and inefficient algorithms.

Definition 3 *An algorithm runs in polynomial time if its running time is $O(n^k)$ for some fixed $k > 0$. Otherwise, an algorithm is called exponential.*

Let us look now at Table 1.3, where the running times of polynomial algorithms are compared with the running times of exponential ones.

	$n = 10$	$n = 20$	$n = 50$	$n = 100$
$O(n)$	0.00001 s.	0.00002 s.	0.00005 s.	0.0001 s.
$O(n^2)$	0.0001 s.	0.0004 s.	0.025 s.	0.01 s.
$O(n^3)$	0.001 s.	0.008 s.	0.125 s.	1 s.
$O(2^n)$	0.001 s.	1.048 s.	35.7 years	10^{16} years
$O(n!)$	3.6 s.	77 146 years	10^{50} years	!!

Table 1.3: Estimation of the running time of different algorithms. We assume that an elementary step can be performed in 10^{-6} s.

The polynomial algorithms, which run in $O(n)$, $O(n^2)$ and $O(n^3)$ time, are efficient and allow us to solve large instances. On the other hand, an exponential algorithm may be inefficient even for small instances. We must, however, be aware of two things. By an efficient polynomial algorithm, we mean an algorithm running in $O(n^k)$ time for small k, say $k = 1, 2, 3, 4$. It is clear that an algorithm running in $O(n^{10})$ time might be useless in practice. Fortunately, for almost all known polynomial algorithms k rarely exceeds 4. On the other hand, an exponential algorithm may not be as bad as one might expect. Recall that

we estimate the running time of an algorithm according to the worst case. So, if an algorithm runs in $O(2^n)$ time, then it might perform 2^n elementary steps for rare and very artificial instances of size n. For typical instances the number of performed steps may be much smaller. A well known example is the famous simplex algorithm (see Appendix B), which is not a polynomial one. However, the average number of elementary steps performed by this algorithm is polynomial and poor behavior appears only for very artificial instances.

For the problems discussed in this book it is not difficult to design an algorithm for solving them. An idea based on brute force almost always works. Our goal, however, should be to design algorithms which are also efficient and this task is much more challenging. As we will see, such efficient polynomial algorithms are known for some problems, while for others the situation is more complex. For example, no polynomial algorithm is known for the traveling salesperson problem and it is widely accepted in the computer science community that no such algorithm exists. Such hard problems require special treatment. You may now read Appendix C to learn more about the complexity of optimization problems.

1.3 Summary

1. An optimization problem Π consists of a set of input data D_Π, a set of solutions $sol(I)$ for each $I \in D_\Pi$ and a cost function $f(x)$ for each $x \in sol(I)$. Given an instance $I \in D_\Pi$, the aim is to find a solution $x^* \in sol(I)$, which minimizes or maximizes the function f.

2. Π is called a *discrete optimization problem* if it can be formulated as a mathematical programming problem in which some variables are restricted to take integer values.

3. We solve optimization problems using algorithms. Each algorithm is a finite step by step procedure, which takes an instance of an optimization problem as the input and gives an optimal solution as the output.

4. We measure the running time of an algorithm as the maximal number of elementary steps $f(n)$ performed on input data of size n. We use big O notation to hide all the constants dependent on the implementation details.

5. An algorithm is polynomial if its running time is $O(n^k)$ for some fixed value of k. Only polynomial algorithms with small k are efficient, where by efficient we mean that they are able to solve large instances in reasonable time.

6. Almost all discrete optimization problems have a trivial brute force algorithm, which simply enumerates all the solutions. However, a brute force algorithm generally requires exponential time and becomes useless even for small problems.

7. There are a lot of discrete optimization problems for which no efficient polynomial algorithm is known. An example is the traveling salesperson problem.

   There are many books on discrete and combinatorial optimization. The book by Ahuja et al. [3] is recommended. The material presented in this book can also be found in books by Papadimitriou and Steiglitz [41], Lawler [34], Bazaara et al. [7] and Garfinkel and Nemhauser [24]. More about the analysis of algorithms and data structures can be found in the books by Cormen et al. [14] and Aho et al. [1].

1.4 Exercises

1. Describe the sets D_Π, $sol(I)$ for $I \in D_\Pi$ and the cost function $f(x)$ for all the sample problems presented in Section 1.1.

2. Show that $2^n \neq O(n^k)$ and $n! \neq O(k^n)$ for any fixed $k > 0$.

3. Consider the following algorithm:

   ```
   INDEX
   INPUT: Array A = [a_1, ..., a_n] of 32-bit integers, an integer a
   OUTPUT: The first index of a in A or null if a is not in A
   1:  for i := 1 to n
   2:      if a_i = a then return i and stop
   3:  next i
   4:  return null
   ```

   Using the big O notation, describe the running time of this algorithm.

4. The knapsack problem can be solved in $O(nW)$ time. Does it mean that the knapsack problem can be solved in polynomial time?

5. Design a polynomial algorithm for the sequencing problem from Example 3 in Section 1.1.

6. Describe a brute force method of solving the shortest path problem (Example 1.1, Section 1.1). What is the running time of this brute force?

Chapter 2

Network Flows

In this chapter we consider a wide and important class of discrete optimization problems, which we call *network flows*. At this point a reader who is not familiar with graph theory should read Appendix A, where some basic notions regarding networks and some basic network algorithms are described. Networks are used to model many real physical systems, such as electrical lines, telephone networks, road systems, computer networks etc. They are also used to model production processes. They have many applications in scheduling, planning, logistics, defense and industrial engineering. In a typical network flow problem we wish to move some entity from one point to another in an underlying network at the smallest possible cost. If this entity is not divisible, then we get a discrete optimization problem defined on some network.

We will start with the simplest and most fundamental problem, called the shortest path problem. In this problem we would like to find a shortest directed path between two given nodes of a network. This problem has a lot of applications and has several efficient algorithms. Next, we consider the maximum flow and the minimum cut problems, which also arise in many applications including network reliability and analysis of traffic networks. Both the shortest path and maximum flow are special cases of the minimum cost flow problem, which is the most general problem considered in this chapter. The minimum cost flow problem can be solved by using an adaptation of the simplex algorithm designed to solve linear programming problems. This network simplex algorithm is very efficient in practice. Furthermore, it allows us to perform a sensitivity analysis of the optimal solution obtained. We will show how to apply this algorithm to the transportation problem, which is a special case of the minimum cost flow. In this chapter we will also consider the minimum cost assignment and the minimum spanning tree problems. In the first problem we wish to pair some objects at a minimal possible cost. In the second one we would like to connect all the nodes of a given network at the minimal possible cost.

Network flow problems constitute a wide class of discrete optimization problems which are polynomially solvable. This means that we can solve problems in which the size of the input network is very large in reasonable time. This

property follows from the special algebraic structure of the linear programming formulation of such problems. The theoretical properties of network flow problems and their relationship with linear programming are discussed in Appendix B. The material presented in this chapter is mainly based on the book by Ahuja et al. [3].

2.1 Shortest path

In this section we discuss one of the most natural and fundamental network problems, namely the *shortest path problem*. Let $G = (N, A)$ be a directed network, where $|N| = n$ and $|A| = m$. Each arc $(i, j) \in A$ has an associated cost c_{ij}. We wish to determine a cheapest (shortest) directed path between two given nodes s and t in G. A sample problem is shown in Figure 2.1. The path 1-3-5 is the shortest one between nodes 1 and 5 and its total cost (length) is equal to 3.

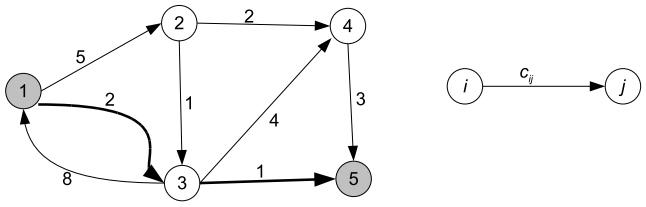


Figure 2.1: The path 1-3-5 is the shortest path from 1 to 5 in this network.

If we do not restrict the arc costs to be nonnegative, then we must be careful. Consider the network shown in Figure 2.2. If we wish to find a shortest path from 1 to 5, then we have a problem. Traversing the directed cycle 1-2-3-1 incurs a total cost equal to -1. We can move along this cycle as many times as we wish before going to node 5, each time decreasing the cost of the path. Therefore, the optimal value is unbounded.

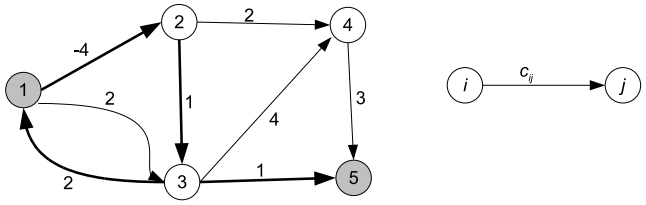


Figure 2.2: There is no shortest path between 1 and 5.

We may assume that it is prohibited to visit any node more than once, so we are not allowed to move along any cycle. However, this assumption completely changes the problem and makes it very difficult to solve (see Appendix C). Therefore, we will assume that if there exists a cycle with a negative cost, then the problem has no solution. So, we must be careful if negative arc costs in the network are allowed and any algorithm for solving the shortest path problem should be able to detect negative cycles in such networks.

In some applications the network may be undirected. If all arc costs are nonnegative, then we can transform such an undirected network into a directed

one as shown in Figure 2.3. This transformation is not correct if some arcs have negative costs, since it creates negative cycles.

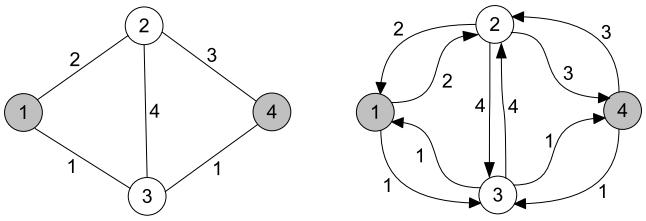


Figure 2.3: Transforming an undirected network into a directed one.

Sometimes we wish to determine the shortest paths from node s to all the other nodes in a network. The paths obtained can be represented by the so-called *tree of shortest paths*. Consider the sample problem shown in Figure 2.4. We would like to compute the shortest paths from node 1 to all the other nodes in the network. These paths can be represented as a tree rooted at node 1. For each node $j \neq 1$, there is a unique path from 1 to j in this tree, representing the shortest path from 1 to j. Observe that the tree of shortest paths can be stored in a single array $[pred(1), \ldots, pred(n)]$, where $pred(i)$ is the direct predecessor of node $i \neq s$ in the tree and $pred(s) = 0$. We can encode the tree shown in Figure 2.4 as $[0, 3, 1, 2, 3]$. This simple representation allows us to retrieve the shortest path from 1 to any other node in G.

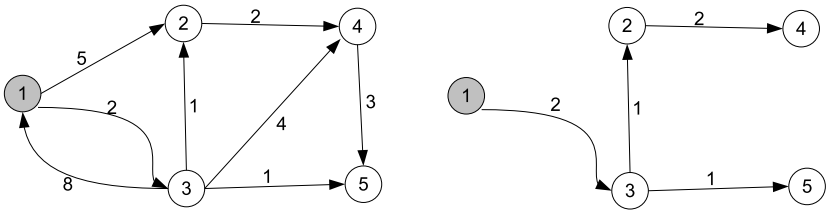


Figure 2.4: The tree of shortest paths from node 1.

Distance labels

All the algorithms computing the shortest paths in G work with *distance labels*. A distance label is a number $d(i)$ associated with node $i \in N$ of network G and $d(i)$ is an upper bound on the length of the shortest path from node s to i. Initially, we may fix $d(s) = 0$ and $d(i) = \infty$ for all $i \in N \setminus \{s\}$. During its execution, an algorithm updates all the distance labels systematically and when it terminates $d(i)$ represents the length of a shortest path from s to i for each $i \in N$. The basic operation performed by all these algorithms is the

so-called *distance label updating*. Consider a pair of nodes $i, j \in N$ linked by arc $(i, j) \in A$. If $d(j) > d(i) + c_{ij}$, then there must be a path from s to j shorter than $d(j)$ and we update $d(j)$ so that $d(j) = d(i) + c_{ij}$. The algorithms use different orders of label updating depending on the structure of the input network G. We will describe the algorithms later in this section.

Longest path

In some applications we would like to find a *longest path* from s to t in a given network G, that is a directed path from s to t of maximum cost (length). We will assume that in this case the network is acyclic. Thus the problem is well defined for both negative and nonnegative arc costs. A sample problem is shown in Figure 2.5.

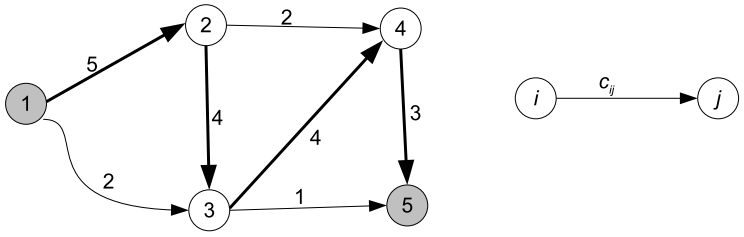


Figure 2.5: A sample network with the longest path from 1 to 5 shown in bold.

2.1.1 Applications

Application 1 (planning a route). The shortest path problem has obvious applications in planning routes. Given a map of roads, say in Poland, we would like to find a shortest tour between two given points on this map. We can model the road system as a network. The nodes of this network represent crossroads and the arcs represent roads. The nonnegative arc costs may represent the lengths or travel times of the roads.

Application 2 (production/inventory model). A car factory wants to establish a production plan for the next k periods. In every period the customers' demand is equal to Q and it must be fully satisfied. Producing j cars costs $c(j)$, where $j = 0, \ldots, 2Q$ and $c(0) = 0$. Unsold cars can be stored at a cost of m_l per unit in periods $l = 1, \ldots, k$. We assume that up to Q cars can be stored in every period. How many cars should the factory produce during each period to minimize the total cost? This problem can be represented as the directed network shown in Figure 2.6.

The numbers at the nodes of the network denote the number of cars stored in each period. Before the first and at the end of the kth period the store is empty. In every period $1, \ldots, k-1$ we can store $0, 1, \ldots, Q$ cars. The arcs of the

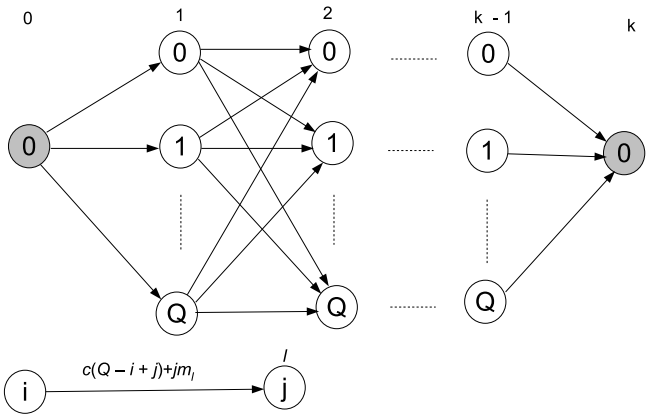


Figure 2.6: The network for the production/inventory model.

network represent transitions between two subsequent inventory states and the cost of the transition (i, j) in the lth period can be computed in the following way. The factory has i cars and must have j cars at the end of the period. So, the factory must produce $Q - i + j$ cars to satisfy demand and the production cost is $c(Q - i + j)$. Storing j cars costs jm_l. Hence, the total cost of the transition (i, j) is $c(Q-i+j)+jm_l$. The optimal production plan can be found by computing the shortest path in the network constructed.

Application 3 (renting a crane). A factory needs a crane, which will be used during the next K months. The cost of renting a crane depends on the month and on the number of months for which it is rented. If the factory rents a crane at the beginning of the ith month and uses it until the end of the jth month, then it must pay c_{ij+1}. We would like to determine the best strategy for renting a crane. This problem can be represented as the directed network shown in Figure 2.7. The nodes represent the beginning of the months and the arcs represent all the possible ways of renting a crane. We obtain a best strategy by computing a shortest path from node 1 to node $K + 1$ in this network.

2.1.2 A dynamic algorithm for acyclic networks

Suppose that the network is acyclic, i.e. it does not contain any directed cycle. The networks from Examples 2 and 3 in previous section are acyclic. Observe that in this case we do not need to worry about negative arc costs, because the network cannot contain any directed cycle of negative cost. In an acyclic network it is possible to label the nodes so that $i < j$ for each arc $(i, j) \in A$. Such a labeling is called a *topological ordering* and can be efficiently performed in $O(m)$ time, where m is the number of arcs (see Appendix A).

Let us associate two numbers with each node $i \in N$: $d(i)$ denoting the

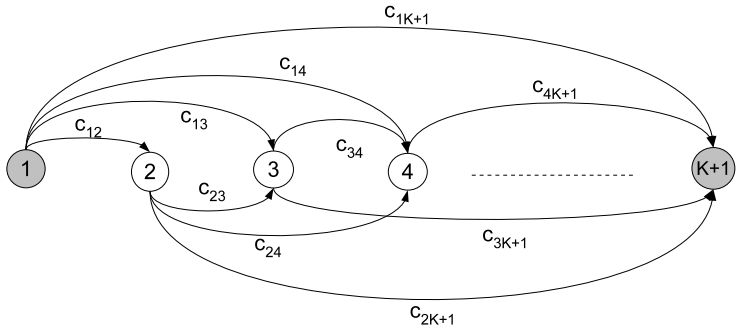


Figure 2.7: The network for the renting crane problem.

distance from node s to node i and $pred(i)$ denoting the direct predecessor of node i on a path from s to i. Our algorithm will iteratively modify these numbers and at the end $d(i)$ will be equal to the length of the shortest path from s to i and the set of $pred(j)$, $j \in N$, will allow us to retrieve the shortest path from s to i. The algorithm is shown in Figure 2.8.

1: Establish a topological ordering of the nodes.
2: $d(1) = 0$, $d(i) = \infty$ for each node $i = 2, \ldots, n$
3: $pred(i) = 0$ for all $i = 1, \ldots, n$
4: **for** $i = 1$ **to** n **do**
5: **for all** j such that $(i, j) \in A$ **do**
6: **if** $d(j) > d(i) + c_{ij}$ **then**
7: $d(j) := d(i) + c_{ij}$, $pred(j) = i$
8: **end if**
9: **end for**
10: **end for**

Figure 2.8: A dynamic algorithm for acyclic networks.

Consider the sample network shown in Figure 2.9. This network is acyclic and the nodes are numbered according to a topological ordering. We wish to compute the shortest paths from $s = 1$ to all the other nodes in the network. Initially $d(1) = 0$, $pred(1) = 0$ and $d(i) = \infty$ and $pred(i) = 0$ for all $i > 1$. This means that the distance from 1 to 1 equals 0 and we have no information about any path from s to i, where $i > 1$, at this moment. We will now consider the nodes in order of the topological ordering. So, we start with node 1. There are two arcs $(1, 2)$ and $(1, 3)$ leaving node 1. We thus update $d(2) = d(1) + c_{12} = 2$, $pred(2) = 1$ and $d(3) = d(1) + c_{13} = 5$, $pred(3) = 1$ (see Figure 2.10a). The next node is 2. There are three arcs $(2, 3)$, $(2, 4)$ and $(2, 5)$ which leave node 2. So we update the distance and predecessor labels of nodes 3,4 and 5 as shown in Figure 2.10b. We then consider the remaining nodes in order 3,4 and 5.

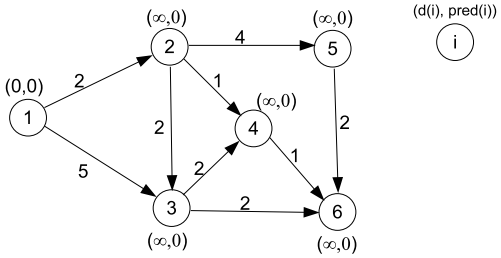


Figure 2.9: A sample network after initialization.

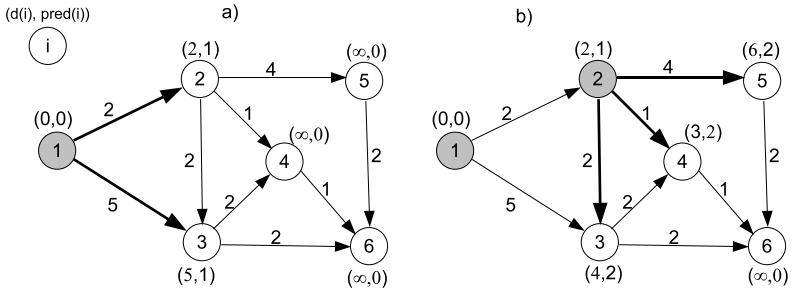


Figure 2.10: Two steps of the dynamic algorithm.

The final results are shown in Figure 2.11. In Figure 2.11a all the nodes contain the shortest distance from node 1 and the direct predecessor on the shortest path from node 1. In Figure 2.11b the obtained tree of shortest paths is shown.

The dynamic algorithm can easily be modified to compute a longest path in G. It is enough to replace line 2 with $d(1) = 0$, $d(i) = -\infty$, $i = 2, \ldots, n$ and the condition in line 6 with $d(j) < d(i) + c_{ij}$.

Correctness and running time of the algorithm

Theorem 4 *The dynamic algorithm solves the shortest path problem in acyclic networks in $O(m)$ time.*

Proof. The crucial fact is that we consider the nodes in order of the topological ordering. Initially $d(1) = 0$, so we have the correct shortest distance from node 1 to itself. We can now use induction to prove the correctness of the algorithm. Suppose that during the ith iteration, steps 4-10 of the algorithm, $d(1), \ldots, d(i)$ are the shortest distances from node 1. Consider iteration $i + 1$, where node $i + 1$ is considered. Let j be the direct predecessor of node $i + 1$ on a shortest path from s to $i + 1$. From the topological ordering, it follows that node j

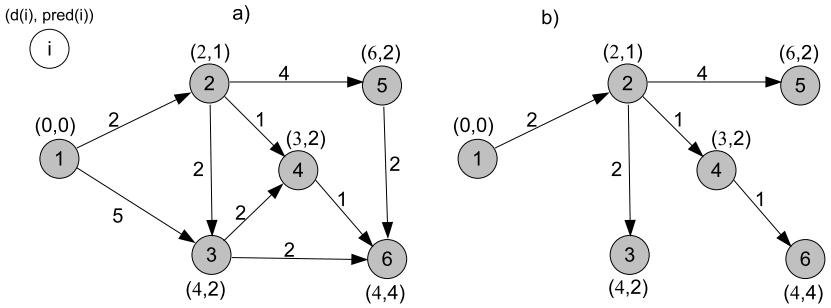


Figure 2.11: The final results and the tree of shortest paths.

must be one of $1, \ldots, i$ and, at this point, $d(j)$ is the shortest distance from 1 to j. Hence, $d(i+1) = d(j) + c_{ij+1}$ is the shortest distance from 1 to $i+1$ and $pred(i+1) = j$. The dynamic algorithm scans every arc of G exactly once and can be easily implemented to run in $O(m)$ time. So it is linear with respect to the number of arcs. ■

2.1.3 Dijkstra's algorithm

If a network contains a directed cycle, then the simple dynamic algorithm, shown in the previous section, does not work. This simply follows from the fact that it is not possible to establish a topological ordering of nodes in a network which is not acyclic (see Appendix A). Consider now a general network in which, however, all arc costs are nonnegative. Let us partition the set of nodes into two subsets: S - the set of marked nodes and $\overline{S}$ - the set of unmarked nodes. During the execution of the algorithm we will know the shortest paths from s to i for all nodes i in S. This algorithm, called Dijkstra's algorithm, is shown in Figure 2.12.

1: $S := \emptyset$, $\overline{S} := N$
2: $d(i) := \infty$ for each node $i \in N$
3: $d(s) := 0$, $pred(s) := 0$
4: **while** $\overline{S} \neq \emptyset$ **do**
5: Let $i \in \overline{S}$ be a node for which $d(i) = \min\{d(j) : j \in \overline{S}\}$
6: $S := S \cup \{i\}$
7: $\overline{S} := \overline{S} \setminus \{i\}$
8: **for all** j such that $(i, j) \in A$ **do**
9: **if** $d(j) > d(i) + c_{ij}$ **then** $d(j) := d(i) + c_{ij}$, $pred(j) := i$
10: **end for**
11: **end while**

Figure 2.12: Dijkstra's algorithm.

Consider the sample network shown in Figure 2.13. Note that this network is not acyclic but all the arc costs are nonnegative. We would like to compute the shortest paths from node 1 to all the other nodes. In Figure 2.13 the network after initialization is shown. Now, according to Dijkstra's algorithm, we must find a node $i \in \overline{S}$, which has the smallest value of $d(i)$. This is node 1. So, we add node 1 to S and remove it from $\overline{S}$. Furthermore, we update nodes 2 and 3 (see Figure 2.14a). We again seek a node $i \in \overline{S}$, which has the smallest value of $d(i)$ and now it is node 3. We add node 3 to S, remove it from $\overline{S}$ and update nodes 4 and 6 (see Figure 2.14b). In the next step we need to choose node 4 and update node 2 (see Figure 2.14c). The algorithm proceeds until the set $\overline{S}$ becomes empty. You may perform the remaining three steps and draw the tree of shortest paths as an exercise.

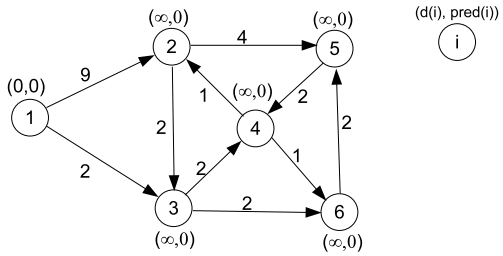


Figure 2.13: A sample network after initialization, $S = \emptyset$, $\overline{S} = \{1, \ldots, 6\}$.

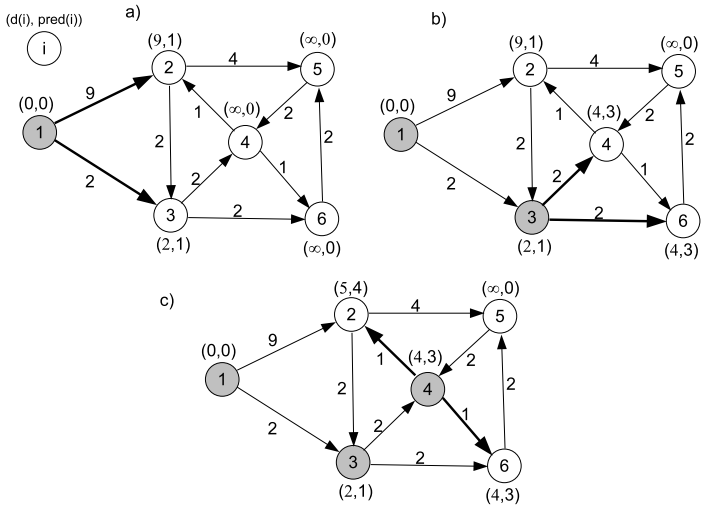


Figure 2.14: Three steps of Dijkstra's algorithm. The grey nodes belong to S.

Correctness and running time of the algorithm

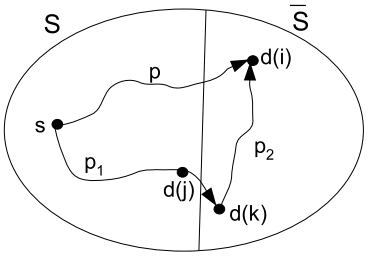


Figure 2.15: Illustration of the proof.

Theorem 5 *Dijksta's algorithm solves the shortest path problem in networks with nonnegative arc costs in $O(n^2)$ time, where n is the number of nodes.*

Proof. We will prove that if node $i \in \overline{S}$ is added to S in step 6 of the algorithm, then $d(i)$ is the shortest distance from s to i. Suppose, by contradiction, that i is the first node added to S, which violates this property. This means that path p of length $d(i)$ is not the shortest one and there exists a path p^* whose length is smaller than $d(i)$. It is easy to see that this path must use at least one node from the set $\overline{S}$ other than i. Let k be the first node in p^* which belongs to $\overline{S}$. So the path p^* consists of two subpaths: p_1 from s to k and p_2 from k to i, which belongs entirely to $\overline{S}$. The node k is the first node in subpath p_2. This situation is shown in Figure 2.15. Since all arc costs are nonnegative, the length of p_2 is nonnegative and, consequently, the length of p_1 is smaller than the length of p. Now observe that p_1 must be the shortest path from s to k (otherwise p^* is not the shortest path from s to i). Let j be the direct predecessor of k in p^*. Since $j \in S$, node k was updated from node j and $d(k) \leq d(j) + c_{jk}$. So $d(k)$ is the length of p_1 and $d(k) < d(i)$. This is a contradiction, because node i has the smallest value of $d(i)$ among all nodes in $\overline{S}$. Let us now estimate the running time of the algorithm. The loop 4-11 is performed n times, because we increase the set S by one in every iteration. Inside this loop we must find a node $i \in \overline{S}$ with the smallest $d(i)$, which takes $O(n)$ time and update some nodes, which also takes $O(n)$ time in the worst case. Hence, the overall running time of Dijkstra algorithm is $O(n^2)$. ■

2.1.4 Floyd-Warshall algorithm

In this section we describe an algorithm which works for all directed networks, containing both directed cycles and negative arc costs. This algorithm will be surprisingly simple. Its running time will be, however, $O(n^3)$, which is larger than the running time of the dynamic and Dijkstra's algorithms. In some applications, this running time may be prohibitive.

Let us arbitrarily number the nodes of the network from 1 to n. Let $d^k[i,j]$ represent the length of a shortest path from node i to node j subject to the condition that this path can use only the nodes $1, 2, \ldots, k-1$ as *internal nodes* (for the path $i_1 - i_2 - \cdots - i_{l-1} - i_l$, the nodes $i_2, \ldots, i_{l-1}$ are internal). If there is no such a path, then $d^k[i,j] = \infty$. The algorithm described in this section will be based on the following equality:

$$d^{k+1}[i,j] = \min\{d^k[i,j], d^k[i,k] + d^k[k,j]\}. \tag{2.1}$$

The proof of equality (2.1) is quite simple. A shortest path that uses only the nodes $1, \ldots, k$ as internal nodes fulfills one of the following two conditions: (1) it does not use node k, in which case $d^{k+1}[i,j] = d^k[i,j]$, (2) it does use node k, in which case $d^{k+1}[i,j] = d^k[i,k] + d^k[k,j]$.

1: **for all** node pairs $[i,j] \in N \times N$ $d[i,j] := \infty, pred[i,j] := 0$
2: **for all** nodes $i \in N$ $d[i,i] := 0$
3: **for all** arcs $(i,j) \in A$ $d[i,j] := c_{ij}$, $pred[i,j] := i$
4: **for** $k = 1$ **to** n **do**
5: **for all** $[i,j] \in N \times N$ **do**
6: **if** $d[i,j] > d[i,k] + d[k,j]$ **then**
7: $d[i,j] := d[i,k] + d[k,j]$
8: $pred[i,j] := pred[k,j]$
9: **end if**
10: **end for**
11: **end for**

Figure 2.16: The Floyd-Warshall algorithm.

We can now proceed by computing $d^1[i,j], d^2[i,j], \ldots, d^{n+1}[i,j]$ for all $(i,j) \in A$. Obviously, $d^{n+1}[i,j]$ represents the shortest distance from i to j. This is exactly what the Floyd-Warshall algorithm does. The algorithm is shown in Figure 2.16. It additionally uses predecessor indices, $pred[i,j]$, for each node pair i,j. The value of $pred[i,j]$ is the direct predecessor of node j on the path from i to j. At any step of the algorithm, a finite value of $d[i,j]$ means that the network contains a directed path from node i to node j of length $d[i,j]$ and this path can be retrieved using the predecessor indices. In steps 6-9 the algorithm updates the distances and predecessor indices, which is illustrated in Figure 2.17.

We now illustrate the algorithm using the sample network shown in Figure 2.18.

In steps 1-3 we initialize $d[i,j]$ and $pred[i,j]$ for all node pairs $i,j = 1, \ldots, n$. We get:

$$d = \begin{bmatrix} 0 & 3 & 1 & 7 & \infty \\ 1 & 0 & 6 & \infty & 2 \\ \infty & \infty & 0 & -3 & -1 \\ \infty & \infty & 6 & 0 & \infty \\ \infty & -1 & \infty & 7 & 0 \end{bmatrix} \quad pred = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 3 & 3 \\ 0 & 0 & 4 & 0 & 0 \\ 0 & 5 & 0 & 5 & 0 \end{bmatrix}$$

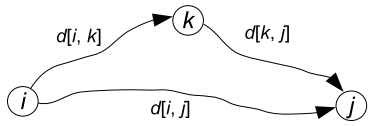


Figure 2.17: If $d[i,j] > d[i,k] + d[k,j]$, then $d[i,j] := d[i,k] + d[k,j]$ and $pred[i,j] := pred[k,j]$.

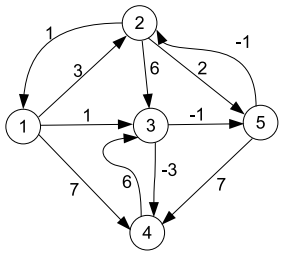


Figure 2.18: A sample network.

We then fix $k = 1$ and seek better paths that can use 1 as an internal node. For example, $d[2,3] > d[2,1] + d[1,3]$, so we update the distance $d[2,3] := d[2,1] + d[1,3] = 2$ and $pred[2,3] := pred[1,3] = 1$. Similarly, $d[2,4] > d[2,1] + d[1,3]$, so we update $d[2,4]$ and $pred[2,4]$. After this we get the following distances and predecessor indices (the updated values are shown in boxes):

$$d = \begin{bmatrix} \mathbf{0} & \mathbf{3} & \mathbf{1} & \mathbf{7} & \boldsymbol{\infty} \\ \mathbf{1} & 0 & \boxed{2} & \boxed{8} & 2 \\ \boldsymbol{\infty} & \infty & 0 & -3 & -1 \\ \boldsymbol{\infty} & \infty & 6 & 0 & \infty \\ \boldsymbol{\infty} & -1 & \infty & 7 & 0 \end{bmatrix} \quad pred = \begin{bmatrix} \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{2} & 0 & \boxed{1} & \boxed{1} & 2 \\ \mathbf{0} & 0 & 0 & 3 & 3 \\ \mathbf{0} & 0 & 4 & 0 & 0 \\ \mathbf{0} & 5 & 0 & 5 & 0 \end{bmatrix}$$

We fix $k = 2$ and seek better paths which can use 2 as an internal node. After performing these computations, we obtain the following distances and predecessor indices:

$$d = \begin{bmatrix} 0 & \mathbf{3} & 1 & 7 & \boxed{5} \\ \mathbf{1} & \mathbf{0} & \mathbf{2} & \boldsymbol{\infty} & \mathbf{2} \\ \infty & \boldsymbol{\infty} & 0 & -3 & -1 \\ \infty & \boldsymbol{\infty} & 6 & 0 & \infty \\ \boxed{0} & \mathbf{-1} & \boxed{1} & 7 & 0 \end{bmatrix} \quad pred = \begin{bmatrix} 0 & \mathbf{1} & 1 & 1 & \boxed{2} \\ \mathbf{2} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{2} \\ 0 & \mathbf{0} & 0 & 3 & 3 \\ 0 & \mathbf{0} & 4 & 0 & 0 \\ \boxed{2} & \mathbf{5} & \boxed{1} & 5 & 0 \end{bmatrix}$$

For $k = 3$ we get:

$$d = \begin{bmatrix} 0 & 3 & \mathbf{1} & \boxed{-2} & \boxed{0} \\ 1 & 0 & \mathbf{2} & \boxed{-1} & \boxed{1} \\ \mathbf{\infty} & \mathbf{\infty} & \mathbf{0} & \mathbf{-3} & \mathbf{-1} \\ \infty & \infty & \mathbf{6} & 0 & \boxed{5} \\ 0 & -1 & \mathbf{1} & \boxed{-2} & 0 \end{bmatrix} \quad pred = \begin{bmatrix} 0 & 1 & \mathbf{1} & \boxed{3} & \boxed{3} \\ 2 & 0 & \mathbf{1} & \boxed{3} & \boxed{3} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{3} & \mathbf{3} \\ 0 & 0 & \mathbf{4} & 0 & \boxed{3} \\ 2 & 5 & \mathbf{1} & \boxed{3} & 0 \end{bmatrix}$$

For $k = 4$ we get:

$$d = \begin{bmatrix} 0 & 3 & 1 & \mathbf{-2} & 0 \\ 1 & 0 & 2 & \mathbf{-1} & 1 \\ \infty & \infty & 0 & \mathbf{-3} & -1 \\ \mathbf{\infty} & \mathbf{\infty} & \mathbf{6} & \mathbf{0} & \mathbf{5} \\ 0 & -1 & 1 & \mathbf{-2} & 0 \end{bmatrix} \quad pred = \begin{bmatrix} 0 & 1 & 1 & 3 & 3 \\ 2 & 0 & 1 & 3 & 3 \\ 0 & 0 & 0 & 3 & 3 \\ 0 & 0 & 4 & 0 & 3 \\ 2 & 5 & 1 & 3 & 0 \end{bmatrix}$$

And finally for $k = 5$ we obtain:

$$d = \begin{bmatrix} 0 & \boxed{-1} & 1 & -2 & \mathbf{0} \\ 1 & 0 & 2 & -1 & \mathbf{1} \\ \boxed{-1} & \boxed{-2} & 0 & -3 & \mathbf{-1} \\ \boxed{5} & \boxed{4} & 6 & 0 & \mathbf{5} \\ \mathbf{0} & \mathbf{-1} & \mathbf{1} & \mathbf{-2} & \mathbf{0} \end{bmatrix} \quad pred = \begin{bmatrix} 0 & \boxed{5} & 1 & 3 & \mathbf{3} \\ 2 & 0 & 1 & 3 & \mathbf{3} \\ \boxed{2} & \boxed{5} & 0 & 3 & \mathbf{3} \\ \boxed{2} & \boxed{5} & 4 & 0 & \mathbf{3} \\ \mathbf{2} & \mathbf{5} & \mathbf{1} & \mathbf{3} & \mathbf{0} \end{bmatrix}$$

The last two matrices contain the complete information about the shortest paths between all the pairs of nodes i and j in the sample network. For example, if we wish to get the shortest paths from node 4, then we should look at the fourth row of the matrix *pred*, which is $[2, 5, 4, 0, 3]$. This row describes the tree of shortest paths from node 4.

Detection of negative cycles

Consider the network shown in Figure 2.19.

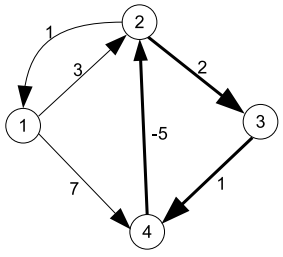


Figure 2.19: A sample network with a negative cycle.

This network contains a cycle, 4-2-3-4, of negative length equal to -2. According to our assumption, there is no shortest path from node 4 to node 2

in this network. Let us see what will happen if we apply the Floyd-Warshall algorithm to this network. After 3 iterations ($k = 3$) we get:

$$d = \begin{bmatrix} 0 & 3 & 5 & 6 \\ 1 & 0 & 2 & 3 \\ \infty & \infty & 0 & 1 \\ -4 & -5 & -3 & \boxed{-2} \end{bmatrix} \quad pred = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 2 & 0 & 2 & 3 \\ 0 & 0 & 0 & 3 \\ 2 & 4 & 2 & \boxed{3} \end{bmatrix}$$

We can see that the distance $d[4,4] = -2$, which means that there is a path from node 4 to itself (a cycle) of negative length equal to -2. This cycle can be retrieved from the fourth row of the matrix *pred* in the following way: $pred[4,4] = 3$, so we get a partial path $3-4$, then $pred[4,3] = 2$, so the partial path is $2-3-4$ and finally $pred[4,2] = 4$, which leads to the cycle $4-2-3-4$. Alternatively, we may try to draw the tree of shortest paths from node 4 and we will also encounter the cycle $4-2-3-4$.

The Floyd-Warshall algorithm is able to detect negative cycles in network G. If at any step $d[i,i] < 0$ for some node i, then there is a cycle of negative length from i to i. Furthermore, this cycle can be easily obtained from the ith row of the matrix *pred*.

Correctness and running time of the algorithm

Theorem 6 *The Floyd-Warshall algorithm solves the problem of finding the shortest path between all pairs of nodes in general networks in $O(n^3)$ time, where n is the number of nodes.*

Proof. The correctness of the algorithm follows directly from equality (2.1). It is also easy to see that the algorithm runs in $O(n^3)$ time, because the loop 4-11 is performed $O(n^3)$ times. ■

The running time of the Floyd-Warshall algorithm does not depend on the network structure. It simply manipulates two square matrices of size $n \times n$ and it is very easy to implement. However, the running time $O(n^3)$ may be prohibitive in some applications.

2.1.5 Project scheduling

In this section we show an important application of the longest path problem to the project scheduling. Recall that in the longest path problem the input network $G = (N, A)$ is assumed to be acyclic and we seek a longest directed path (i.e. a path of maximum cost) between nodes s and t in G. Such a path can be computed by using a slight modification of the dynamic algorithm presented in Section 2.1.2. Consider the following example. A project consists of six activities and their description is shown in Table 2.1. So we can see that instructing the workers and buying materials should be done first. When these two activities are completed, we can start producing tools 1 and tools 2. The tools 2 should be additionally tested. Finally, tools 1 and 2 should be merged to obtain a final product. Each activity has some *duration time*. For example, instructing

the workers takes 2 units of time, the materials can be bought within 5 units of time etc. We would like to answer the following questions. What is the duration time of the project and which activities are critical, that is their duration times cannot be increased without increasing the project duration?

Activities	Direct predecessors	Duration times
A (Instruct the workers)	-	2
B (Buy materials)	-	5
C (Produce tools 1)	**A**,**B**	6
D (Produce tools 2)	**A**,**B**	2
E (Test tools 2)	**D**	1
F (Merge tools 1 and 2)	**C**,**E**	5

Table 2.1: Description of the sample project.

We can model the project as a directed and acyclic network $G = (N, A)$ whose arcs represent the activities and nodes the events denoting the start and finish of the corresponding activities. Since the network is acyclic, we can use a topological ordering to label the nodes. The network for the sample project is shown in Figure 2.20. For example, the activity **C** is represented by the arc $(3, 5)$, where node 3 denotes the start and node 5 denotes the finish of **C**. The duration time of the activity **C** is $t_{35} = 6$. The dashed arc represents a special *dummy activity*, which expresses that the activities **C** and **D** must start after **A** and **B** are completed.

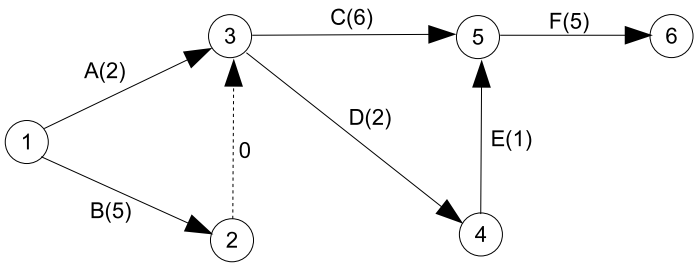


Figure 2.20: The network representation of the sample project.

Let $ET(i)$ denote the earliest starting time of the event i. The values of $ET(i)$ for all $i \in N$ can be computed by using the following formula:

$$\begin{cases} ET(1) = 0 \\ ET(i) = \max\limits_{\{j:\ (j,i)\in A\}} (ET(j) + t_{ji}) \quad i = 2, \ldots, n \end{cases}$$

Note that the values $ET(i)$ represent the longest distances from node 1 to i and they can be computed by applying a dynamic algorithm to the network.

Because n is the earliest time when the last activity is finished, $ET(n)$ is the earliest completion time of the project, so it is the project duration time. Let $LT(i)$ denote the latest starting time of event $i \in N$, which does not increase the project duration time. The values of $LT(i)$ for all $i \in N$ can be computed in the following way:

$$\begin{cases} LT(n) = ET(n) \\ LT(i) = \min\limits_{\{j:\ (i,j)\in A\}} (LT(j) - t_{ij}) \quad i = n-1, \ldots, 1 \end{cases}$$

Hence, we can compute the values of $LT(i)$, $i \in N$, by performing a backward dynamic computation, starting from the node n. The quantity $TF(i,j) = LT(j) - ET(i) - t_{ij}$ is called the *total float* of activity (i,j). The activities for which $TF(i,j) = 0$ are called *critical*. Their duration times cannot be increased without increasing the duration time of the project. Each longest (critical) path in G is composed of some critical activities.

The values of $ET(i)$, $LT(i)$ and $TF(i,j)$ for the sample project are shown in Figure 2.21. We can see that the project duration time is equal to 16. The longest (critical) path is $1-2-3-5-6$ and is composed of the critical activities **B**, **C** and **F**. All these activities have the total flow equal to 0, which means that increasing their duration times increases the duration time of the project. On the other hand, the activities **A**, **D** and **E** have the total flow equal to 3. So we can increase the duration time of one of these activities by at most 3 units without increasing the project duration time. Alternatively, we can delay the starting time of each of these activities by at most 3 units without increasing the project duration time.

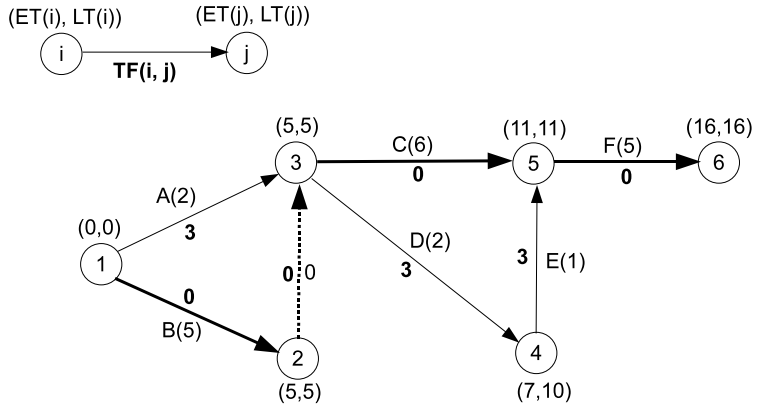


Figure 2.21: Computing the time characteristics for the sample project.

The project can also be represented in the form of the *Gantt chart* (see Figure 2.22). This chart clearly shows which activities should be executed at each time and which activities can be delayed without increasing the project duration time.

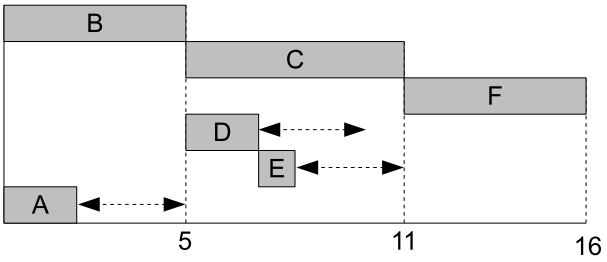


Figure 2.22: The Gantt chart of the sample project.

2.1.6 Summary

1. In the shortest path problem we seek a directed path from node s to node t of minimum total cost (length). Most algorithms are able to compute a tree of shortest paths, which contains a shortest path from s to every other node of G.

2. If the input network has a directed cycle of negative cost, then the problem has no solution.

3. If the network is acyclic, then a simple dynamic algorithm can be applied to compute the shortest paths from node s to all other nodes of G. This algorithm runs in $O(m)$ time and can be easily modified to compute longest paths in the network. Furthermore, the dynamic algorithm works with both nonnegative and negative arc costs.

4. For general networks with nonnegative arc costs, Dijkstra's algorithm can be applied. This algorithm runs in $O(n^2)$ time and can be improved for some sparse networks.

5. For general networks with any arc costs, the Floyd - Warshall algorithm can be applied. This algorithm computes the shortest paths between each pair of nodes and is able to detect negative cycles. The running time of the algorithm is $O(n^3)$ and it may be too slow if the input network is very large. The algorithm is very simple to implement and works with two matrices containing distance and predecessor indices.

   The implementation details of all the algorithms presented in this section can be found, for example, in [14] and [45]. Dijkstra's algorithm was first described in [18]. The simplest and original version of Dijkstra's algorithm runs in $O(n^2)$ time. This running time can be improved for various classes of networks and the corresponding references can be found in [3]. The Floyd-Warshall algorithm is from [20] and [49].

2.2 Maximum flow

Let $G = (N, A)$, $|N| = n$, $|A| = m$, be a directed network with a *source node* s and a *sink node* t. Each arc $(i, j) \in A$ has an associated *capacity* $u_{ij} \geq 0$. A *flow* $f = (x_{ij})_{(i,j)\in A}$ in G is defined by numbers x_{ij}, specified for each arc $(i, j) \in A$, fulfilling the following two conditions:

1. $0 \leq x_{ij} \leq u_{ij}$ for all $(i, j) \in A$.
2. $\sum_{\{i:(i,k)\in A\}} x_{ik} = \sum_{\{j:(k,j)\in A\}} x_{kj}$ for all nodes $k \in N \setminus \{s, t\}$.

Condition 1 means that the flow x_{ij} along arc (i, j) is nonnegative and cannot exceed the arc capacity u_{ij}. Condition 2 means that the flow is not created nor destroyed at any node k, other that s and t. Namely, the total *inflow* to k is equal to the total *outflow* from k. The *value of a flow* f is defined as $|f| = \sum_{\{j:(s,j)\in A\}} x_{sj} - \sum_{\{i:(i,s)\in A\}} x_{is}$. So, the value of f is the flow created at the source node s and sent through the network to the sink node t. In the maximum flow problem, we wish to determine a flow in G of maximum value. A sample problem is shown in Figure 2.23. The numbers in brackets fulfill conditions 1 and 2, so they establish a flow in this network. The value of this flow is equal to 13 and, in fact, this is the maximum flow in this network.

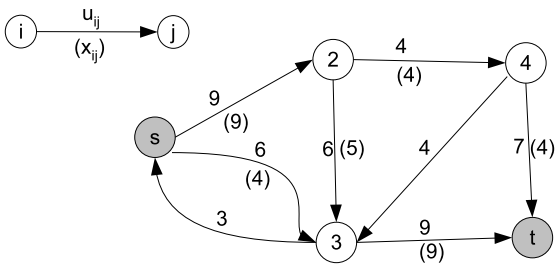


Figure 2.23: A sample network with the solution to the maximum flow problem.

Flows and cuts

Recall that an $s - t$ *cut* $[S, \overline{S}]$ in G is a partition of the node set N into two subsets S and $\overline{S}$ such that $s \in S$ and $t \in \overline{S}$. We refer to an arc (i, j) with $i \in S$ and $j \in \overline{S}$ as a *forward arc*, and an arc (i, j) with $i \in \overline{S}$ and $j \in S$ as a *backward arc*. The *capacity* $u[S, \overline{S}]$ of an $s - t$ cut $[S, \overline{S}]$ is the sum of capacities of the forward arcs in the cut. A sample $s - t$ cut $[S, \overline{S}]$ is shown in Figure 2.24. We have $S = \{s, 2, 3\}$ and $\overline{S} = \{4, t\}$. The forward arcs in this cut are $(2, 4)$ and $(3, t)$. Hence the capacity of this cut is equal to $u_{24} + u_{3t} = 13$. A cut of minimum capacity in G is called a *minimum cut*. One can verify that the cut shown in Figure 2.24 is the minimum cut.

The notions of the maximum flow and the minimum cut are closely related. Intuitively, if there is a cut in G of capacity U, then the maximum flow in

G cannot exceed U. This follows from the fact that any flow which can be sent through this cut cannot exceed the total capacity of its forward arcs. The relationship between the maximum flow and minimum cut is formally stated in the following theorem (its formal proof can be found in the literature, see. e.g. [3]):

Theorem 7 *The value of the maximum flow in G is equal to the capacity of the minimum cut in G.*

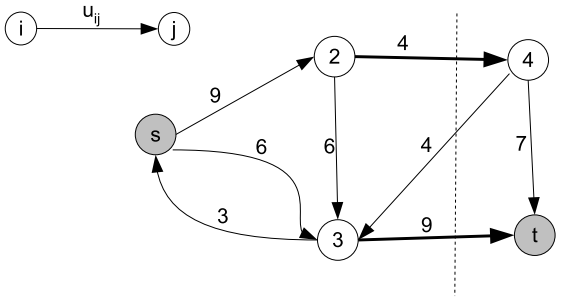


Figure 2.24: A sample $s-t$ cut of capacity 13.

Observe that the capacity of the cut shown in Figure 2.24 equals the value of the flow f from Figure 2.23. So, the flow f is maximal.

Augmenting flow along paths and cycles

In all network algorithms considered in this, and also in the next sections, a procedure called *flow augmentation* will often be used. So, it is very important to understand this basic operation on a flow. Consider the example shown in Figure 2.25.

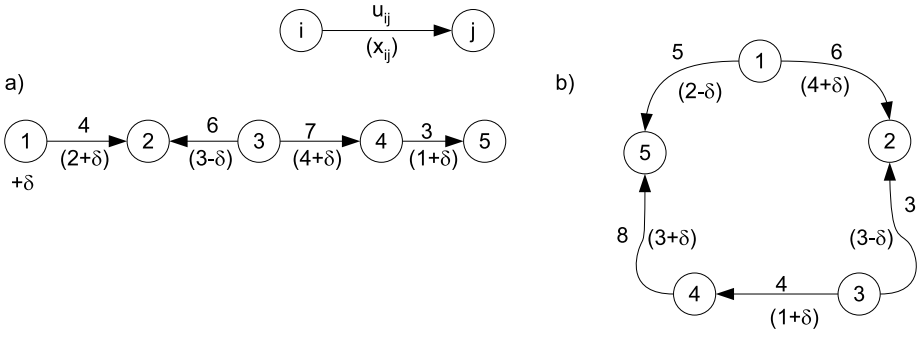


Figure 2.25: Augmenting a flow along (a) a path and (b) a cycle.

In Figure 2.25a a sample path (not necessarily directed) 1-2-3-4-5, being a part of some network, is shown. There is a flow along the arcs of this path. Suppose that an additional δ units of flow appear at node 1 and should be delivered to node 5 along this path. What is the maximal value of δ? The additional flow is first sent along arc $(1,2)$. Since the flow along this arc is 2 and the capacity of this arc is 4, we can send maximum 2 units of flow along $(1,2)$. The next arc on the path is $(3,2)$. In order to satisfy condition 2 for the flow at node 2, we must decrease the flow along arc $(3,2)$ by δ. Since the flow is nonnegative we can subtract at most 3 units from arc $(3,2)$. Proceeding in this way, we can see that $\delta = \min\{4-2, 3, 7-4, 3-1\} = 2$. Hence, we can send a maximum of $\delta = 2$ units of flow from 1 to 5 along the path by increasing the flow by δ along arcs $(1,2), (3,4), (4,5)$ and decreasing the flow by δ along arc $(3,2)$. We can also send a flow from node 5 to 1 along this path and we leave this as a an exercise.

Sending a flow around a cycle can be considered in a similar way. Consider the sample cycle shown in Figure 2.25b. We can send a flow along this cycle in a clockwise or counterclockwise direction. Consider the clockwise direction. So we increase the flow by δ along arcs $(1,2)$, $(3,4)$, $(4,5)$ and decrease the flow by δ along arcs $(3,2)$ and $(1,5)$. In consequence, $\delta = \min\{6-4, 3, 4-1, 8-3, 2\} = 2$ and we can send a maximum of 2 units along the cycle. We leave the computation of δ for the counterclockwise direction as an exercise.

2.2.1 Applications

Application 1 (analysis of pipeline systems). Suppose that k water pumps are located at points $A_1, \ldots, A_k$, which can provide $s_1, s_2, \ldots, s_k$ liters of water per minute. The water is delivered to points $B_1, \ldots, B_l$, which require $d_1, d_2, \ldots, d_l$ liters of water per minute. The pipeline system is modeled by a directed network, whose arcs represent pipes with specified capacities (in liters per minute). Is it possible to satisfy the water demand of all points? If not, which parts of the system should be modernized? A sample problem is shown in Figure 2.26.

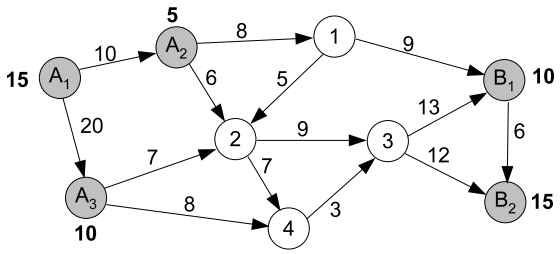


Figure 2.26: A sample pipeline system.

The solution to this problem is shown in Figure 2.27. We first modify the network by adding a source s and a sink t. We then add arcs (s, A_1), (s, A_2) and

(s, A_3) with capacities 5, 15 and 10 respectively and arcs (B_1, t) and (B_2, t) with capacities 10 and 15. The capacities of these additional arcs represent the water supplies and demands of the nodes. We then solve the maximum flow problem for this modified network. The solution is shown in Figure 2.27. As we can see, a maximum of 20 liters of water can flow through this system in one minute and the water demand of both B_1 and B_2 cannot be satisfied. According to the solution obtained, point B_1 receives 8 liters per minute and point B_2 receives 12 liters per minute. In order to find the weakest parts of this system, we compute a minimum cut in the network. This minimum cut is shown in Figure 2.27 and the forward arcs in this cut are $(A_2, 1)$, $(2, 3)$ and $(4, 3)$. So, if we wish to increase the flow in this system we should consider increasing the capacity of these three arcs (pipes) first.

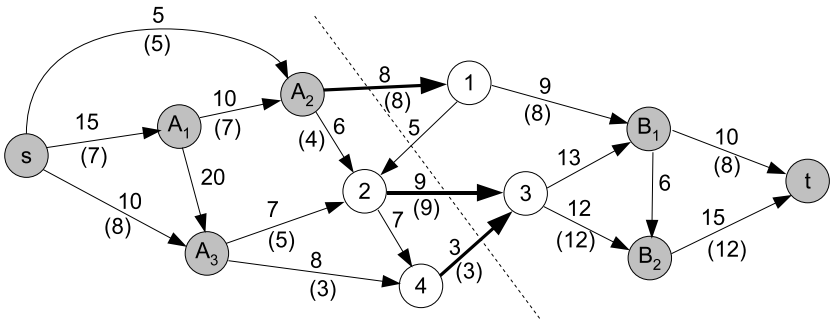


Figure 2.27: The solution to the sample problem.

Application 1 (analysis of traffic networks). A system of roads and crossroads in a city is given. Each road (crossroad) has an associated capacity, which represents the maximum number of cars which can pass through it within one minute under certain traffic conditions. How many cars can pass through this system in one minute? Where are the weakest parts of this system located? A sample problem is shown in Figure 2.28. The system consists of three crossroads and some roads connecting the crossroads. For each road and crossroad the number of cars that can pass through it within 1 minute are shown.

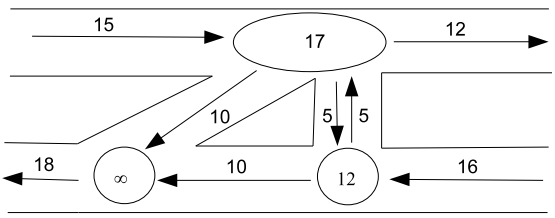


Figure 2.28: A sample road system.

In order to analyze this sample road system, we build the network shown in Figure 2.29. We split every crossroad with finite capacity into two points - the start and the end of the crossroad. The arc joining these two points has a capacity equal to the capacity of the crossroad. After computing the maximum flow in this network, we find that a maximum of 27 cars can pass through this system within 1 minute. The minimum cut is composed of arcs (s, A_1) and (B_1, B_2) and they represent the weakest parts of the road system.

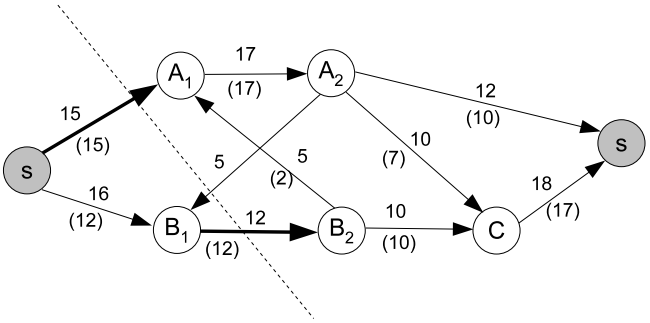


Figure 2.29: A sample pipeline system.

Application 3 (network reliability). Given a network $G = (N, A)$, what is the minimum number of arcs that we should remove from the network so that it contains no directed path from node s to node t? This problem arises in a number of applications. For instance, G may represent an electrical network providing energy from s to t. We may ask what is the robustness of this network, namely how many lines must be damaged before energy is not delivered to t. This problem can be solved by computing a minimum cut in G. Consider the sample network shown in Figure 2.30.

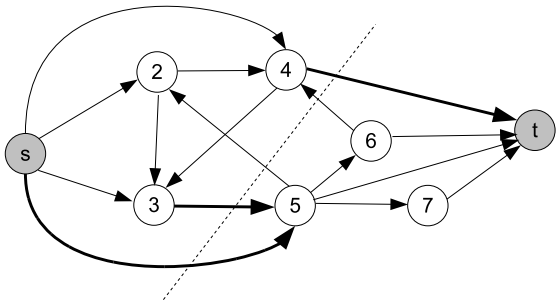


Figure 2.30: A sample problem.

Suppose that energy must be delivered from s to t and energy is delivered if there is a directed path from s to t. We first assume that the capacity of

every arc is equal to 1. We then compute a minimum cut in this network (see Figure 2.30). This minimum cut contains 3 forward arcs, so at least three arcs (representing lines) must be damaged before energy is not delivered from s to t.

2.2.2 The Ford - Fulkerson algorithm

In this section we present an algorithm for solving the maximum flow problem. This algorithm will also give us a minimum cut as a byproduct. The idea of the algorithm is as follows. It starts with flow equal to 0 along every arc and then iteratively increases the value of the current flow until no additional increase is possible. So we must address two questions: (1) how can we increase the current flow and (2) how can we check whether the current flow is maximal? Consider arc $(i, j) \in A$ with capacity u_{ij} and flow $0 \leq x_{ij} \leq u_{ij}$. We can modify the flow along this arc in two ways. We can increase the flow by at most $u_{ij} - x_{ij}$ or decrease the flow by at most x_{ij}. This observation leads to the very useful concept of a *residual network*. Let $G = (N, A)$ be a network with a flow $f = (x_{ij})_{(i,j)\in A}$. We construct a *residual network* $G(f)$ in the following way:

1. For each arc $(i, j) \in A$ we create two arcs:
   - (i, j) with capacity $r_{ij} = u_{ij} - x_{ij}$
   - (j, i) with capacity $r_{ji} = x_{ij}$
2. We remove all arcs in $G(f)$ with 0 capacity (that is, with $r_{ij} = 0$).

An example of a residual network $G(f)$ is shown in Figure 2.31. In Figure 2.31a a network with a sample flow f is shown. In Figure 2.31b the residual network $G(f)$ is shown. Notice that all the arcs in $G(f)$ have positive capacities. Notice also that each directed path in the residual network $G(f)$ is also a path (but not necessarily directed) in the original network G.

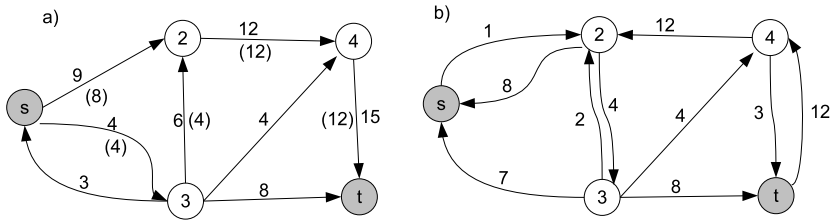


Figure 2.31: a) A network G with a flow f b) Residual network $G(f)$.

How can we check, whether the flow shown in Figure 2.31a is maximal? We can send an additional positive flow from s to t in G if there is a directed path from s to t in $G(f)$. This follows from the fact that every arc in $G(f)$ shows some possibility of modifying the flow in the original network G. If we look at the residual network $G(f)$ in Figure 2.31, then we can easily discover such a

path, for example $s-2-3-t$. This directed path in $G(f)$ corresponds to the same path $s-2-3-t$ in the undirected version of the original network G. We can now increase the flow by $\delta > 0$ units along $s-2-3-t$. In order to determine δ, we check the residual capacities of all the arcs on the path $s-2-3-t$ in $G(f)$ and choose the minimum of these capacities, which in our case is equal to 1. We now augment the flow along the path $s-2-3-t$ in G by $\delta = 1$ unit. Namely, we increase the flow along arc $(s,2)$ by 1, decrease the flow along arc $(3,2)$ by 1 and increase the flow along arc $(3,t)$ by 1. We have thus increased the value of the current flow by 1 (see Figure 2.32).

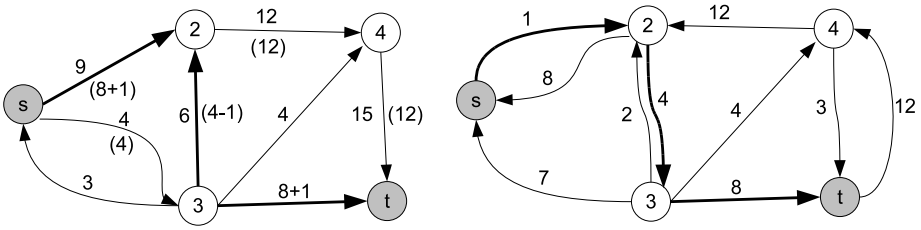


Figure 2.32: a) A network G with a flow f b) Residual network $G(f)$. The directed path $s-2-3-t$ in $G(f)$ corresponds to the augmenting path $s-2-3-t$ in G.

Consider now the example shown in Figure 2.33.

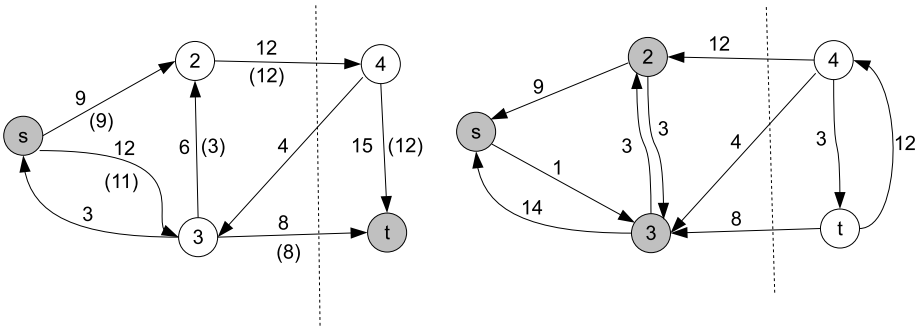


Figure 2.33: a) A network G with a flow f b) Residual network $G(f)$.

The residual network $G(f)$ for the flow f does not contain any directed path from s to t. This means that it is not possible to increase the flow and f is the maximum flow in G. We can however send some flow from s to the nodes 2 and 3. But this flow cannot be sent further. Define $S = \{s, 2, 3\}$ and $\overline{S} = \{4, t\}$. Then $[S, \overline{S}]$ defines a cut in G. The capacity of this cut cannot be less than 20, because the current flow is equal to 20. However, it also cannot be greater than 20, because otherwise we could discover a directed path in $G(f)$ to some

node in $\overline{S}$. So $[S, \overline{S}]$ is a minimum cut in G. Summarizing, we can compute the minimum cut in G in the following way. Let f be the maximum flow and let S be the set containing the source node s and all the nodes $i \in N$ for which there is a directed path from s to i in $G(f)$. Let $\overline{S} = N \setminus S$ be the set of all the remaining nodes. Then $[S, \overline{S}]$ is the minimum cut in G.

Theorem 8 *A flow f is maximal in G if and only if there is no directed path between s and t in the residual network $G(f)$. Furthermore, if S is the set of all the nodes that are reachable from s in $G(f)$, then $[S, N \setminus S]$ is a minimum s-t cut in G.*

Using Theorem 8, we can design an algorithm for computing the maximum flow in G, which is shown in Figure 2.34. This algorithm is illustrated in Figure 2.35. We start with the flow along every arc being equal to 0, in which case networks G and $G(f)$ are the same (see Figure 2.35a). We then discover a directed path $s-1-2-t$ in $G(f)$ and augment by 2 units of flow along this path in G obtaining a flow f of value 2. We again construct the residual network $G(f)$ and seek a directed path in $G(f)$. The algorithm finishes after performing 3 augmentations and the resulting flow of value 6 is shown in Figure 2.35d. This figure also shows the minimum cut in G.

1: $f := 0$
2: **while** $G(f)$ contains a directed path from s to t **do**
3: Identify a directed path p from node s to t in $G(f)$
4: $\delta := \min\{r_{ij} : (i,j) \in p\}$
5: Augment δ units of flow along p in G and update $G(f)$
6: **end while**

Figure 2.34: The augmenting path (Fulkerson - Ford) algorithm.

Observe that if all the arc capacities are integer, then the algorithm always sends an integer flow along augmenting paths (i.e. δ is always integer). In consequence, according to the obtained maximum flow, the flows along all the arcs are integer. Let us emphasize this important result.

Theorem 9 *If all the arc capacities are integer, then there is a maximum flow $f = (x_{ij})_{(i,j)\in A}$ such that all the x_{ij} are integer.*

Running time of the algorithm

Theorem 10 *The Ford-Fulkerson algorithm solves the maximum flow problem in $O(mnU)$ time, where U is the largest capacity among all the arcs.*

Proof. Step 3 of the algorithm, where we seek a directed path in the residual network, can be performed in $O(m)$ time (see Appendix A). Observe that every iteration of steps 2-6 increases the value of the current flow by at least one. So, the number of augmentations equals $O(m)$ multiplied by an upper bound on the

maximal flow in G. If all the arc capacities are integer and bounded by a finite number U, then the capacity of any cut is at most nU. Hence, the maximum flow is bounded by nU and the worst case running time of the algorithm is $O(mnU)$. ■

The generic Ford-Fulkerson algorithm may run in exponential time. An example is shown in Figure 2.36, where the residual networks for the first three iterations are shown. If we always send flow along the augmenting paths $s - 1 - 2 - t$ or $s - 2 - 1 - t$, then the algorithm will perform $2 * 10^K$ iterations each time increasing the flow by only 1. Note that there is no rule of choosing a directed path in $G(f)$ if there is more than one such path. So, in the worst case, the algorithm will always choose $s - 1 - 2 - t$ or $s - 2 - 1 - t$.

We can improve the running time of the algorithm using the following result:

Theorem 11 (Edmonds - Karp [19]) *If the algorithm always chooses a shortest directed path in $G(f)$ (with respect to the number of arcs), then the Ford-Fulkerson algorithm with this modification runs in $O(nm^2)$ time.*

So using the simple modification of the algorithm, we get an algorithm which runs in polynomial time. Observe that applying this modification to the problem from Figure 2.36, we get the maximum flow in two steps, because we choose path $s - 1 - t$ or $s - 2 - t$ and send 10^K units within one iteration.

2.2.3 Summary

1. In the maximum flow problem we seek a flow of maximum value from a source node s to a sink node t in a given network G with specified arc capacities.

2. The value of the maximum flow is equal to the minimum capacity among all $s - t$ cuts in G.

3. The augmenting path (Ford-Fulkerson) algorithm solves the problem in $O(mnU)$ time, which can be improved to $O(nm^2)$ by using the simple Edmonds-Karp modification. So the problem solvable in polynomial time. The algorithm also returns a minimum cut as a byproduct.

4. If all the arc capacities are integer, then there exists a maximum flow such that the flows along each arcs are integer.

   The implementation details of all the algorithms presented in this section can be found, for example, in [14] and [45]. The Ford-Fulkerson augmenting path algorithm was first described in [21]. The famous max flow - min cut theorem was also first established in this paper. Some additional interesting combinatorial results that are provable using the max flow - min cut theorem can be found in [3] and [22]. The original Ford - Fulkerson algorithm does not run in polynomial time. Two polynomial time implementations of this algorithm were suggested by Edmonds and Karp in [19]. Some other efficient algorithms for the maximum flow problem are also known and a description of them can be found in [3].

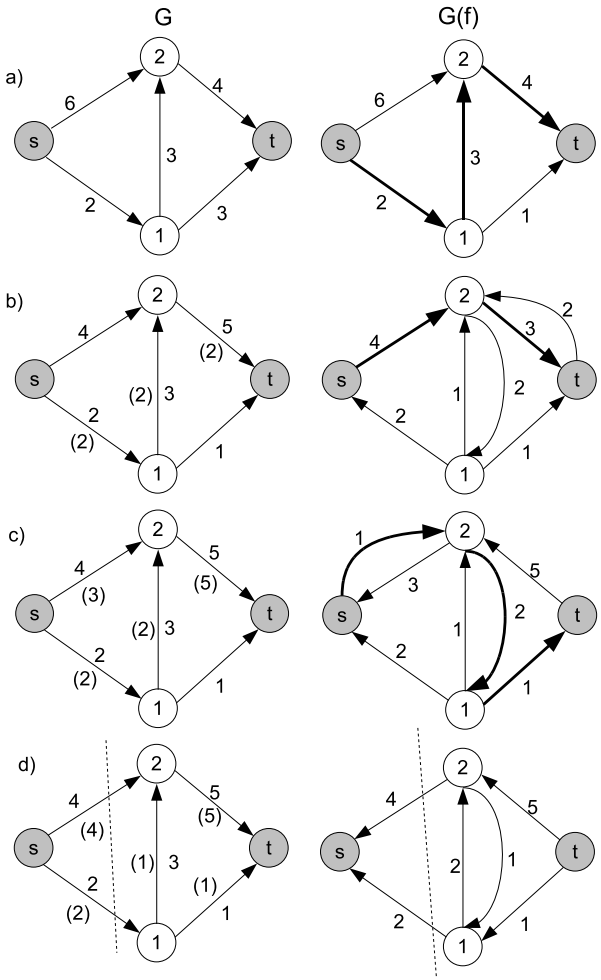


Figure 2.35: Illustration of the Ford-Fulkerson algorithm.

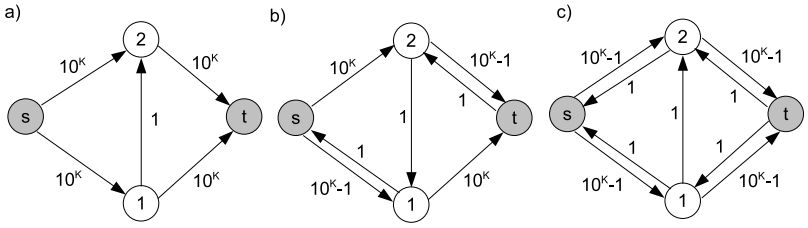


Figure 2.36: A pathological case for the algorithm [3].

2.3 Minimum cost flow

In this section we consider one of the most important and most general network problems, called the *minimum cost flow problem*. In fact, both the shortest path and the maximum flow problems can be viewed as special cases of the minimum cost flow problem. Let $G = (N, A)$ be a directed network, where $|N| = n$ and $|A| = m$. We associate with each node $i \in N$ an integer number $b(i)$ which indicates its *supply* or *demand*, depending on whether $b(i) > 0$ or $b(i) < 0$. If $b(i) > 0$, then i is called a *supply node* and if $b(i) < 0$, then i is called a *demand node*. We will assume that $\sum_{i \in N} b(i) = 0$, which means that the problem is *balanced*, that is to say the total supply is equal to the total demand. Each arc $(i, j) \in A$ has an associated integer cost c_{ij} and integer capacity $u_{ij} \geq 0$. A *flow* $f = (x_{ij})_{(i,j) \in A}$ in G is defined by numbers x_{ij} specified for each arc $(i, j) \in A$, fulfilling the following two conditions:

1. $0 \leq x_{ij} \leq u_{ij}$
2. $\sum_{\{j:(k,j) \in A\}} x_{kj} - \sum_{\{i:(i,k) \in A\}} x_{ik} = b(k)$ for all nodes $k \in N$.

The *cost of the flow* f is defined as $c(f) = \sum_{(i,j) \in A} c_{ij} x_{ij}$. In the *minimum cost flow problem* we seek a flow with the minimum cost.

In order to provide an interpretation of flow, consider the example shown in Figure 2.37.

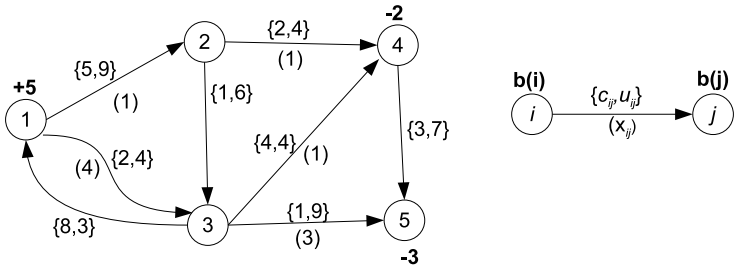


Figure 2.37: A sample minimum cost flow problem.

Node 1 in this network is a supply node and supplies $b(1) = 5$ units. Nodes 4 and 5 are demand nodes and require $b(4) = -2$ and $b(5) = -3$ units respectively. The remaining nodes, 2 and 3, are neither demand nor supply ones (they can be viewed as transshipment nodes) and have $b(2) = b(3) = 0$. Notice that the total supply equals the total demand, which can be expressed as $b(1) + \cdots + b(5) = 0$. Suppose now that the 5 units, available at node 1, must be sent to nodes 4 and 5. Then condition 1 expresses the fact that the number of units sent along arc $(i, j) \in A$ is nonnegative and cannot exceed its capacity u_{ij}. Condition 2 means that for each node $k \in N$ the total outflow from k minus the total inflow to k is equal to $b(k)$. One can check that the flow shown in Figure 2.37 fulfills conditions 1 and 2 and its cost is 22.

In a given network G a flow satisfying conditions 1 and 2 may not exist. Furthermore, even if a flow exists, there may be no flow of minimum cost because the optimal value might be unbounded. The problem of computing a feasible flow in G will be addressed in Section 2.3.2. Observe that the optimal value is unbounded if there is a directed cycle in G of both negative cost and infinite capacity (i.e. all the arcs in this cycle have infinite capacities). In this case, we can augment the flow along this cycle by an arbitrary amount obtaining an arbitrarily small cost. In this section we will make the following assumption:

Assumption. *The input network G does not have a directed cycle of negative cost.*

Under the above assumption we can remove all infinite arc capacities. If $u_{ij} = \infty$, then we can fix $u_{ij} = M$, where M is a sufficiently large constant. Interestingly, we can also transform the problem so that there are no negative arc costs. The approach based on node potentials and reduced arc costs will be presented in Section 2.3.4.

2.3.1 Applications

Application 1 (optimal loading of an airplane [3]). A small company uses a plane with a capacity of p passengers. The plane visits the cities $1, 2, \ldots, n$ in this fixed order. It can pick up passengers at any city and drop them off at any other city. Let b_{ij} be the number of passengers available at city i, who want to go to city j, and let f_{ij} denote the fare per passenger from i to j. How many passengers should the plane carry between various cities to maximize the total profit?

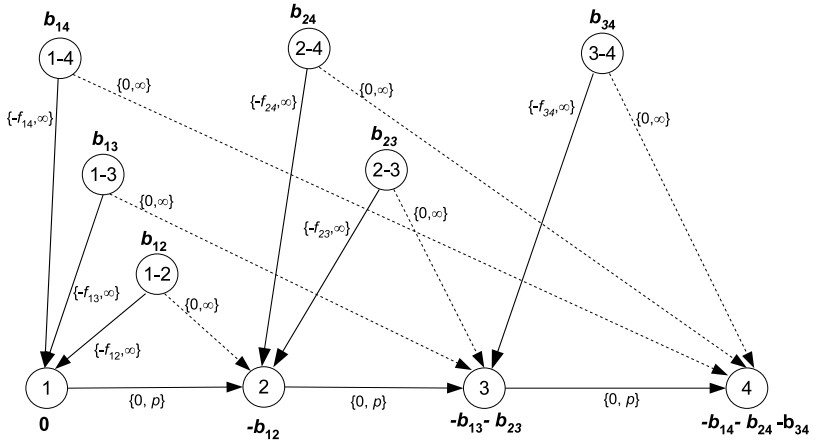


Figure 2.38: The network for 4 cities [3].

The network for 4 cities is shown in Figure 2.38. We get an optimal solution by computing a minimum cost flow for this network. Notice that the network is acyclic, so our assumption from the previous section is satisfied.

Application 2 (dynamic lot sizing). A factory produces some product and it wishes to meet prescribed demand d_j for each of K periods $j = 1, 2, \ldots, K$ by producing up to a_j in period j and/or by drawing upon the inventory I_{j-1} carried forward from the previous period. The unit production cost in the jth period is equal to c_j and the unit inventory cost in the jth period is equal to m_j. We assume that up to I units can be stored between periods. The factory wants to establish an optimal production plan.

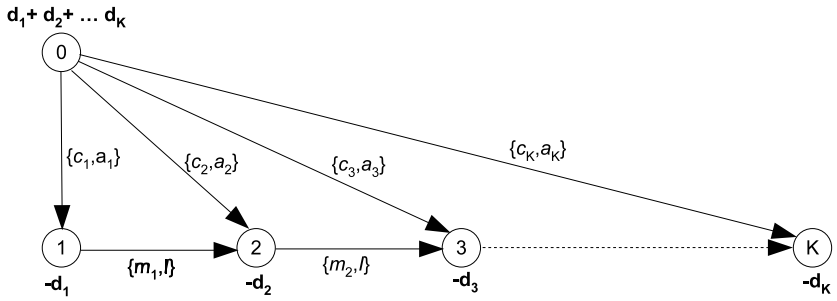


Figure 2.39: The network for the dynamic lot sizing problem.

The network for this problem is shown in Figure 2.39. We get an optimal solution by computing a minimum cost flow in this network.

2.3.2 Establishing a feasible flow

Most algorithms which solve the minimum cost flow problem typically start with some flow and then iteratively decrease the cost of the current flow until some optimality conditions are satisfied. Therefore, the first nontrivial problem to be addressed is how to establish a starting feasible flow (i.e. satisfying the conditions 1 and 2 from Section 2.3) in a given network G. Notice that there may be no feasible flow in G at all. For example, if G consists of only two nodes, 1 and 2, linked by the arc $(1, 2)$ and $b(1) = 2$, $b(2) = -2$, $u_{12} = 1$, then there is no feasible flow in G. So, we must also be able to detect the situation, in which a feasible flow does not exist.

There are several methods of determining a feasible flow in a network G. The first one is illustrated in Figure 2.40. Given a network G, we first add a source node s and a sink node t to G. We then add an additional arc (s, i) for each supply node $i \in N$ with cost 0 and capacity $b(i)$ and an additional arc (i, t) for each demand node $i \in N$ with cost 0 and capacity $-b(i)$. We then solve the maximum flow problem in this modified network. If the maximal flow obtained is less than the total supply (or equivalently the total demand), then there is no

feasible flow in the original network G; otherwise the maximum flow obtained is a feasible flow in G.

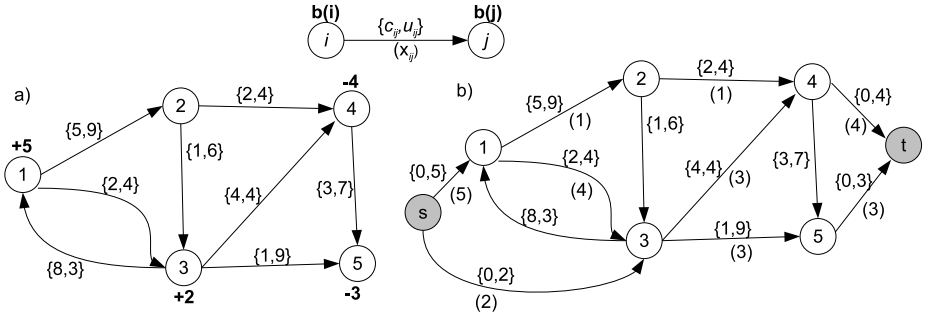


Figure 2.40: Establishing a feasible flow.

The second method is shown in Figure 2.41. We add to network G an additional node s with $b(s) = 0$ and an arc (i, s) for each supply node $i \in N$ and an arc (s, i) for each demand node $i \in N$. All these additional arcs have infinite costs and infinite capacities. We may fix these costs and capacities using a sufficiently large constant M. In this transformed network we can immediately establish a feasible flow without any computational effort. We simply fix $x_{is} = b(i)$ for all $(i, s) \in A$ and $x_{si} = -b(i)$ for all $(s, i) \in A$. The flow along all the remaining arcs is equal to 0 (see Figure 2.41b). We can now compute an optimal flow in this transformed network. If according to the optimal flow obtained, at least one additional arc has a positive flow, then there is no feasible flow in the original network. Otherwise, the optimal flow in the transformed network is also an optimal flow in the original one.

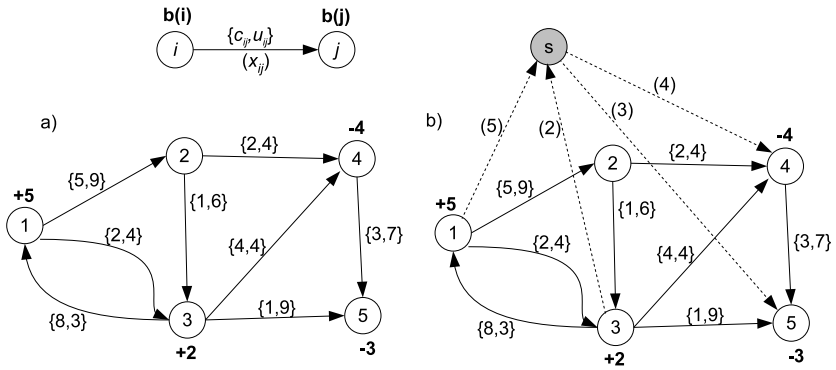


Figure 2.41: Establishing a feasible flow. The additional dashed arcs have costs and capacities equal to a sufficiently large constant M.

The second method of obtaining a feasible flow is less time consuming. We must, however, carefully fix the value of M. Also, the algorithms for computing an optimal flow will typically perform more iterations if the second method is used. On the other hand, using the first method, we can quickly detect whether there is a feasible flow in G.

2.3.3 The cycle canceling algorithm

As for the maximum flow, the concept of a residual network is very useful in the minimum cost flow problem. It is similar to the one defined in Section 2.2.2. We only have to additionally take the arc costs into account. Assume that an arc $(i, j) \in A$ with cost c_{ij} and capacity u_{ij} has a flow equal to x_{ij}. We can increase the flow along this arc by at most $u_{ij} - x_{ij}$ units at a cost of c_{ij} per unit or decrease the flow by at most x_{ij} units at a cost of $-c_{ij}$ per unit. This leads to the following definition. Let $G = (N, A)$ be a network with a flow $f = (x_{ij})_{(i,j)\in A}$. We construct the *residual network* $G(f)$ in the following way:

1. For each arc $(i, j) \in A$ we create two arcs:
   - (i, j) with capacity $r_{ij} = u_{ij} - x_{ij}$ and cost c_{ij}
   - (j, i) with capacity $r_{ji} = x_{ij}$ and cost $-c_{ij}$
2. We remove all the arcs in $G(f)$ with 0 capacity.

An example of a residual network is shown in Figure 2.42.

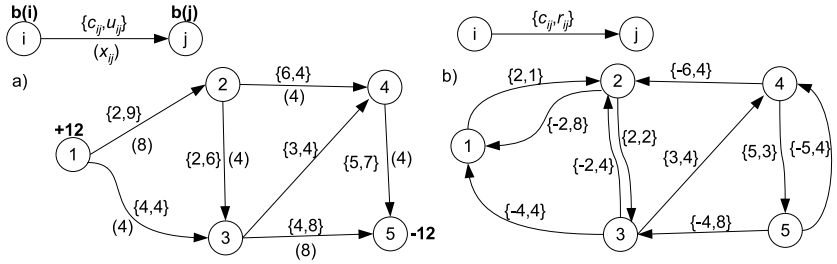


Figure 2.42: a) Network G with a flow f (b) Residual network $G(f)$.

How can we check whether a flow f is optimal? Consider the example shown in Figure 2.43. In Figure 2.43a a network G with a flow f is shown together with its residual network $G(f)$. The residual network $G(f)$ contains directed cycle, 2-3-4-2, of negative cost equal to -1. The original network G also contains cycle 2-3-4-2, which is however not directed. We can now augment the flow $\delta > 0$ around this cycle. Namely, we can increase the flow along arc $(2, 3)$ by δ, increase the flow along arc $(3, 4)$ by δ and decrease the flow along arc $(2, 4)$ by δ (see Figure 2.43b). After this augmentation, we get another flow f_1 whose cost is $c(f_1) = c(f) + 2\delta + 3\delta - 6\delta = c(f) - \delta$. So $c(f_1) < c(f)$ and the flow f_1 is better

than f. The value of δ is the minimum capacity among the arcs of the cycle 2-3-4-2 in $G(f)$. So in our example $\delta = 2$. We can summarize this observation as follows: *if the residual network $G(f)$ contains a directed cycle of negative cost, then it is possible to find a flow f_1 whose cost is less than f by augmenting the flow around this cycle.* This observation also gives us a necessary condition for the optimality of a flow f. Namely, if a flow f is optimal, then the residual network $G(f)$ does not contain any cycle of negative cost.

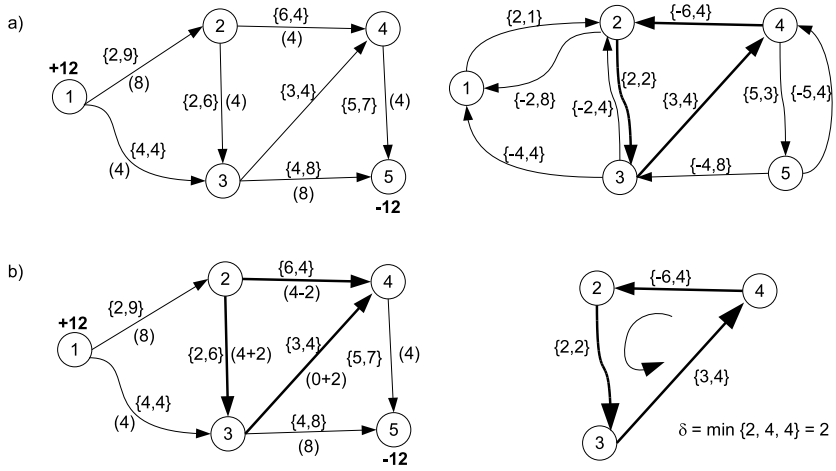


Figure 2.43: (a) Network G with a feasible flow f and $G(f)$. (b) Augmenting a flow around a cycle.

Suppose now that the residual network $G(f)$ does not contain any cycle of negative cost. Can we conclude that the flow f is optimal? The answer to this question is positive and a proof of this fact can be found, for example, in [3]. We thus have the following important result:

Theorem 12 *A flow f is optimal in G if and only if the residual network $G(f)$ contains no negative cost directed cycle.*

Theorem 12 gives us a sufficient and necessary condition of optimality for the flow f. In other words, given a flow f we can easily check whether this flow is optimal and, if not, efficiently compute a flow f_1 with lower cost. In consequence, we can design a simple algorithm for computing an optimal flow, called the *cycle canceling algorithm* (see Figure 2.44). This algorithm starts with an initial feasible flow f, which can be obtained by using the methods presented in the previous section. It then seeks a negative cost directed cycle in $G(f)$ and, if it finds one, then the flow f is improved by augmenting the flow around this cycle. If there is no directed cycle of negative cost in $G(f)$, then the algorithm stops and returns f.

The cycle canceling algorithm is illustrated in Figure 2.45. In Figure 2.45a a network G with an initial feasible flow f and the corresponding residual network

1: Establish a feasible flow f in G
2: **while** $G(f)$ contains a negative cost directed cycle **do**
3: Use some algorithm to identify a negative cost directed cycle W in $G(f)$
4: $\delta := \min\{r_{ij} : (i,j) \in W\}$
5: Augment the flow in the cycle W in G by δ units and update $G(f)$
6: **end while**

Figure 2.44: The cycle canceling algorithm.

$G(f)$ are shown. This residual network contains a directed negative cost cycle 2-3-4-2. So we can decrease the cost of f by augmenting $\delta = 2$ units along this cycle in G. The network with the new flow is shown in Figure 2.45b. The residual network for this flow again contains a directed negative cost cycle 2-1-3-4-2, so we can find a better flow by augmenting the flow by $\delta = 1$ unit around this cycle. The final flow is shown in Figure 2.45c. The residual network for this flow does not contain any directed negative cost cycle and, consequently, this flow is optimal.

Running time of the algorithm

Theorem 13 *The cycle canceling algorithm solves the minimum cost flow problem in $O(nm^2UC)$ time, where U is the largest capacity and C is the largest absolute cost among all the arcs.*

Proof. It is easy to check that the cost of an optimal flow belongs to the interval $[-mUC, mUC]$. In each iteration the algorithm always decreases the cost of the current flow. Therefore, the number of iterations performed is $O(mUC)$. A negative cycle can be detected in $O(n^3)$ time by using the Floyd-Warshall algorithm, but a faster $O(nm)$ time algorithm also exists (see e.g. [3]). So, the running time of the cycle canceling algorithm is $O(nm^2UC)$. ■

The basic cycle canceling algorithm is not polynomial. Its generic version does not specify the order in which negative cycles are selected from the residual network. There are several improved versions of this algorithm. According to one of them we always select a negative cycle of the smallest *mean cost*. With this modification the cycle canceling algorithm runs in polynomial time (see [3] for more details).

2.3.4 Network simplex

In this section we describe one of the most efficient algorithms for the minimum cost flow problem, called the *network simplex*. The name of this algorithm indicates that it is an adaptation of the well known simplex algorithm used for solving linear programming problems (see Appendix B).

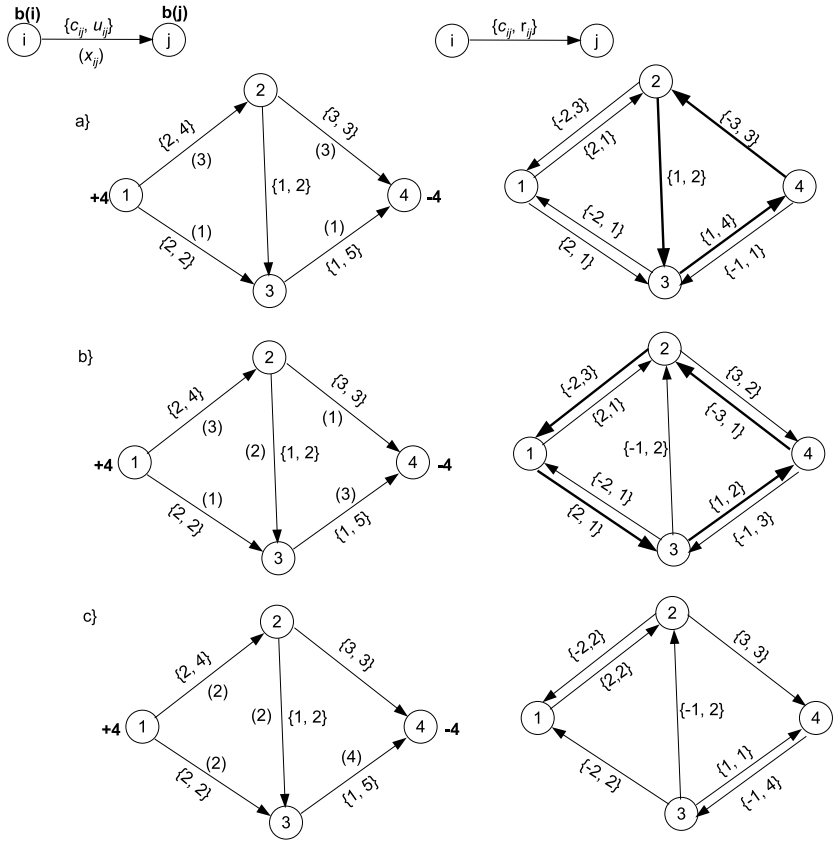


Figure 2.45: Illustration of the cycle canceling algorithm [3].

Spanning tree solution

For any flow $f = (x_{ij})_{(i,j)\in A}$ in $G = (N, A)$, we say that an arc $(i, j) \in A$ is *free* if $0 < x_{ij} < u_{ij}$ and *restricted* if $x_{ij} = 0$ or $x_{ij} = u_{ij}$. For the sample problem shown in Figure 2.37, arcs $(1, 2), (2, 4), (3, 4)$ and $(3, 5)$ are free and all the remaining arcs are restricted. A flow f and an associated spanning tree T of the network G is a *spanning tree solution* if every nontree arc $(i, j) \notin T$ is a restricted arc (alternatively, the network does not contain any cycle composed of free arcs). Note that the tree arcs can be free or restricted. A sample spanning tree solution is shown in Figure 2.46. The spanning tree associated with the flow is composed of arcs $(1, 2), (2, 4), (3, 4), (3, 5)$. Observe that all the nontree arcs are restricted.

Now the crucial observation is that if the problem has an optimal solution, then it also has an optimal spanning tree solution. This follows from the fact that any solution can be transformed into to a spanning tree solution by can-

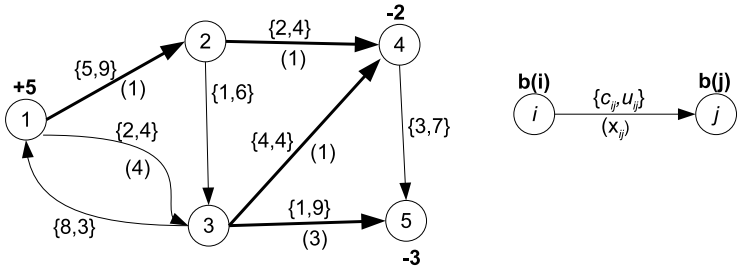


Figure 2.46: A spanning tree solution. All the nontree arcs $(3,1),(1,3),(2,3)$ and $(4,5)$ are restricted.

celing all the cycles composed of free arcs. Furthermore, this transformation does not increase the cost of the flow. An example is shown in Figure 2.47. Suppose that a network contains the cycle 1-2-3-4-5-1 composed of free arcs. Since all arcs are free, we can augment the flow by $\delta > 0$ in either the clockwise or counterclockwise direction around this cycle. If we augment by δ clockwise, then the cost of the resulting new flow will decrease by δ. It is easy to check that the maximum value of δ is 2. After augmenting the flow around the cycle by 2 units we get a new flow of lower cost. Furthermore, the arc $(4,5)$ will become restricted, so a cycle composed of free arcs will be canceled. We can cancel in this way all the cycles composed of free arcs. This procedure is finite, because after canceling a cycle at least one arc becomes restricted and the number of arcs is finite.

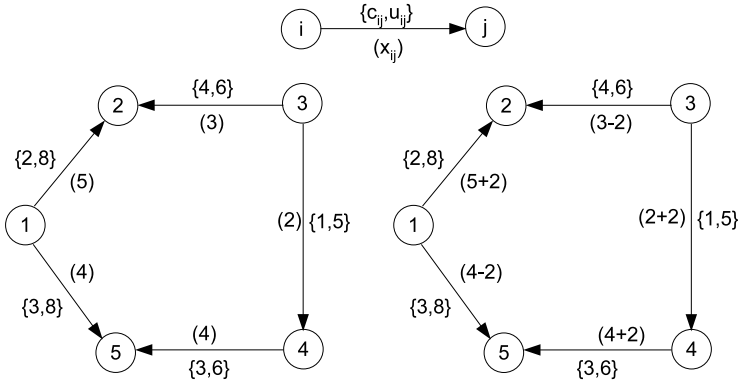


Figure 2.47: Augmenting a flow around a cycle composed of free arcs.

The following observation summarizes our reasoning:

Observation 14 *If the minimum cost flow problem has an optimal solution, then it also has an optimal spanning tree solution.*

Spanning tree structure

Let us partition the set of arcs in network G into three subsets: $\boldsymbol{T}$ - the spanning tree arcs, $\boldsymbol{L}$ - the nontree arcs whose flow is set to 0 and $\boldsymbol{U}$ - the nontree arcs whose flow is set to their capacity u_{ij}. Notice that all the arcs in $\boldsymbol{L} \cup \boldsymbol{U}$ are restricted. The tri-partition $(\boldsymbol{T}, \boldsymbol{L}, \boldsymbol{U})$ is called a *spanning tree structure*. For a given spanning tree structure $(\boldsymbol{T}, \boldsymbol{L}, \boldsymbol{U})$ we can compute a flow in the following way. We first fix $x_{ij} = 0$ for all $(i, j) \in \boldsymbol{L}$ and $x_{ij} = u_{ij}$ for all $(i, j) \in \boldsymbol{U}$. It remains to compute the flow on the tree arcs $(i, j) \in \boldsymbol{T}$. Consider the example shown in Figure 2.48.

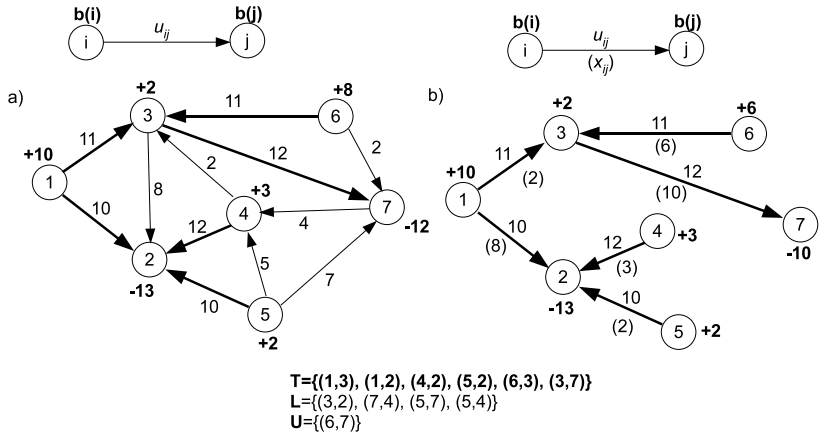


Figure 2.48: Computing a flow for a given spanning tree structure.

In Figure 2.48a a network with spanning tree structure $(\boldsymbol{T}, \boldsymbol{L}, \boldsymbol{U})$ is shown. We first fix the flow on the arcs in $\boldsymbol{L}$ and $\boldsymbol{U}$. Notice that after fixing $x_{67} = 2$ we should modify the supply/demand of nodes 6 and 7 by adding 2 units to $b(7)$ and subtracting 2 units from $b(6)$. The spanning tree with modified node supplies/demands is shown in Figure 2.48b. Now it is easy to see that there is a unique flow on the spanning tree arcs. We start the computation of this flow by considering the leaves of the spanning tree. Node 6 must send 6 units, so the flow along arc $(6, 3)$ must be equal to 6. Similarly, node 7 must receive 10 units, so the flow along arc $(3, 7)$ must be equal to 10. We now modify the supply of node 3, which becomes 2-10+6=-2. Consequently, the flow along arc $(1, 3)$ equals 2. We can proceed in this way and, as a result, we obtain the unique flow along all the tree arcs. We can alternatively obtain the flow by solving the

following system of linear equations:

$$\begin{aligned}
&x_{63} = 6\\
&-x_{37} = -10\\
&x_{42} = 3\\
&x_{52} = 2\\
&x_{13} + x_{63} - x_{37} = 2\\
&-x_{12} - x_{42} - x_{52} = 13\\
&x_{12} + x_{13} = 10
\end{aligned}$$

After obtaining the solution to this system, we must check whether the flow along each arc is nonnegative and does not exceed its capacity, i.e. $0 \leq x_{ij} \leq u_{ij}$ for all $(i,j) \in T$. If so, then we say that the spanning tree structure is *feasible*. Let us summarize the above reasoning.

Observation 15 *Given a spanning tree structure $(\boldsymbol{T}, \boldsymbol{L}, \boldsymbol{U})$, we can efficiently check whether it is feasible and, if so, compute the unique flow corresponding to this structure.*

How can we check whether a given feasible spanning tree structure corresponds to an optimal solution of the minimum cost flow problem? In order to provide an answer to this question, we introduce the concepts of node potentials and reduced costs. Let us associate a real number $\pi(i)$, unrestricted in sign, with each node $i \in N$. This number is called the *potential* of node i. Let us now define the *reduced costs* $c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j)$ of the arcs $(i,j) \in A$ with respect to the node potentials π.

Suppose that we have replaced the original costs c_{ij} with the reduced costs c^{π}_{ij} for all the arcs $(i,j) \in A$. Denote by $c(f)$ the cost of a flow under the original costs and by $c^{\pi}(f)$ the cost of f under the reduced costs. The node potential $\pi(i)$ increases the cost of all the arcs entering i by $\pi(i)$ and decreases the costs of all the arcs leaving i by $\pi(i)$. Thus the total decrease in the cost of the flow equals $\pi(i)$ times the outflow from node i minus the inflow to node i. Since this difference equals the supply/demand of i, the decrease in the cost equals $b(i)\pi(i)$. Repeating this argument for each node we obtain $c^{\pi}(f) = c(f) - \sum_{i \in N} \pi(i) b(i)$. Since $\sum_{i \in N} \pi(i) b(i)$ is a constant which does not depend on f, we conclude that the problems with arc costs c_{ij} and c^{π}_{ij} have the same optimal solutions. Let us formalize this result.

Observation 16 *For any choice of node potentials π, the minimum cost flow problems with arc costs c_{ij} and c^{π}_{ij}, $(i,j) \in A$, have the same optimal solutions.*

We can now use the node potentials to verify the optimality of a given feasible spanning tree structure.

Theorem 17 *A spanning tree structure $(\boldsymbol{T}, \boldsymbol{L}, \boldsymbol{U})$ is optimal if it is feasible and for some choice of node potentials π, the reduced arc costs $c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j)$ satisfy the following conditions:*

- $c_{ij}^{\pi} = 0$ *for all* $(i,j) \in \boldsymbol{T}$
- $c_{ij}^{\pi} \geq 0$ *for all* $(i,j) \in \boldsymbol{L}$
- $c_{ij}^{\pi} \leq 0$ *for all* $(i,j) \in \boldsymbol{U}$

Proof. The theorem easily follows from Theorem 12 and Observation 16. Indeed, suppose that some node potentials π satisfy the conditions from Theorem 17. Let us replace the original costs c_{ij} with the reduced ones c_{ij}^{π} for all $(i,j) \in A$. According to Observation 16, this modification does not change the optimal solutions to the problem. We can now compute the flow f corresponding to the spanning tree structure $(\boldsymbol{T}, \boldsymbol{L}, \boldsymbol{U})$ and construct the residual network $G(f)$ for this flow. It is easy to check that $G(f)$ does not contain any arc with negative cost. So the residual network $G(f)$ cannot contain any directed cycle of negative cost and, by Theorem 12, the flow f is optimal. ■

Consider now the example shown in Figure 2.49. In Figure 2.49a a sample network with a given spanning tree structure is shown. We wish to check whether this spanning tree structure is optimal. In order to satisfy the optimality conditions from Theorem 17, we must find the node potentials π such that $c_{ij}^{\pi} = 0$ for all tree arcs $(i,j) \in \boldsymbol{T}$. So these potentials have to satisfy the following system of equations:

$$\begin{aligned}
5 - \pi(1) + \pi(3) &= 0 \\
2 - \pi(6) + \pi(3) &= 0 \\
3 - \pi(3) + \pi(7) &= 0 \\
4 - \pi(1) + \pi(2) &= 0 \\
1 - \pi(4) + \pi(2) &= 0 \\
2 - \pi(5) + \pi(2) &= 0
\end{aligned}$$

This system has n variables and $n-1$ equations. We may arbitrarily fix one potential, say $\pi(1) = 0$ and then compute the unique solution to the system obtained. The node potentials computed are shown in Figure 2.49b. We can now compute the reduced costs c_{ij}^{π} of all the arcs $(i,j) \in A$ and check that all the reduced costs satisfy the optimality conditions given in Theorem 17. If, additionally, the spanning tree structure is feasible (we leave verification of this as an exercise), then it is also optimal.

The network simplex algorithm

The idea of the network simplex algorithm is the following. We start with an initial feasible spanning tree structure. This structure can be obtained, for example, by using a slight modification of the second method of computing a feasible flow presented in Section 2.3.2. In the example presented in Figure 2.41, we should first add an additional arc, $(s,2)$. The initial feasible spanning tree structure would be then $\boldsymbol{T} = \{(1,s),(3,s),(s,2),(s,5),(s,4)\}$, $\boldsymbol{U} = \emptyset$ and $\boldsymbol{L}$ would contain all the arcs of the original network G. Obtaining a feasible flow for this structure is straightforward. Having this initial spanning tree structure, we check whether it is optimal by computing the node potentials. If it is not

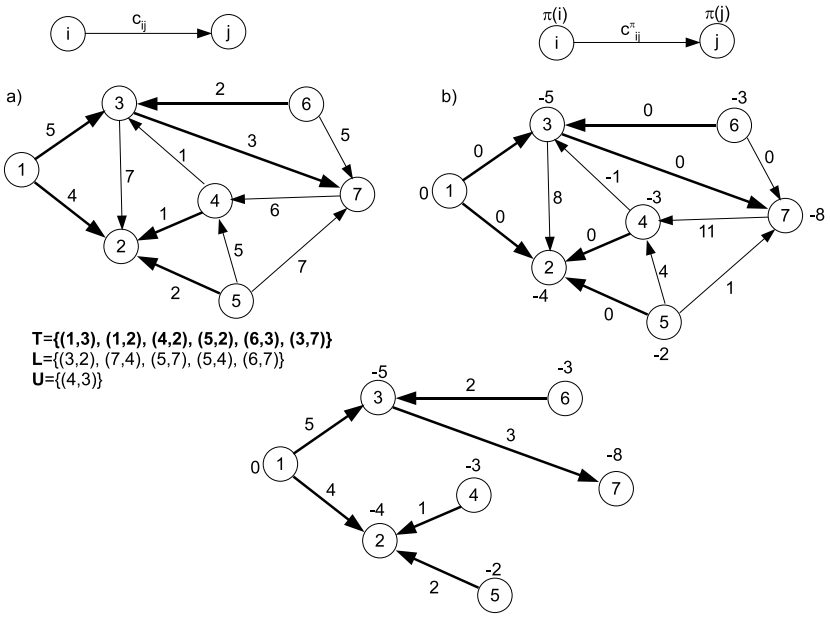


Figure 2.49: Computing node potentials for a given spanning tree structure.

optimal, i.e. some arc violates the optimality conditions, we compute the next spanning tree structure. The arc (k, l) that violates the optimality conditions must be a nontree arc. So, adding (k, l) to $\boldsymbol{T}$ creates a unique cycle. We augment the flow around this cycle so that arc (p, q) in this cycle becomes restricted. Then the spanning tree structure is updated by moving (p, q) to $\boldsymbol{L}$ or $\boldsymbol{U}$, as appropriate. We continue this until some spanning tree structure satisfies the optimality conditions. The flow associated with this structure is then optimal.

1: Determine an initial feasible spanning tree structure
2: Let f and π be the flow and node potentials associated with this structure
3: **while** some arc violates the optimality conditions **do**
4: Select an entering arc (k, l) violating the optimality conditions
5: Add the arc (k, l) to the tree and determine the leaving arc (p, q)
6: Perform the tree update and update f and π
7: **end while**

Figure 2.50: The network simplex algorithm.

We now illustrate the algorithm using the example shown in Figure 2.51.

In Figure 2.51a a sample network is shown together with an initial feasible spanning tree structure and flow f. In Figure 2.51b the computed node potentials and reduced costs are presented. The arc $(1, 3)$ violates the optimality

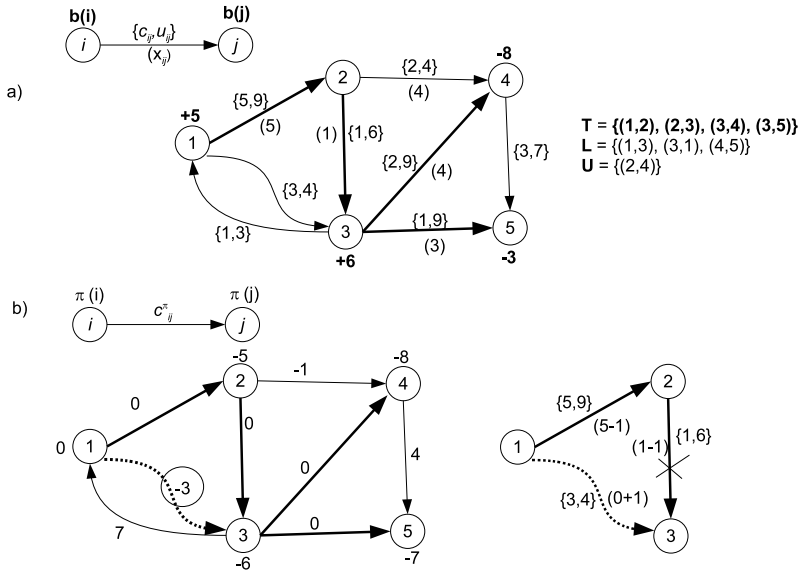


Figure 2.51: (a) A feasible spanning tree structure (b) Reduced arc costs

conditions, because $(1,3) \in \boldsymbol{L}$ and $c_{13}^{\pi} < 0$. So we add $(1,3)$ to $\boldsymbol{T}$, which creates a cycle 1-2-3-1. Since the flow along $(1,3)$ is 0, we wish to increase the flow along $(1,3)$ by δ. If we increase the flow along $(1,3)$ by δ, then we must decrease the flow along $(2,3)$ and $(1,2)$ by δ. So the maximal value of δ is equal to 1. After augmenting $\delta = 1$ unit around the cycle 1-2-3-1, the arc $(2,3)$ becomes restricted (its flow decreases to 0). It is removed from $\boldsymbol{T}$ and added to $\boldsymbol{L}$.

The second spanning tree structure, together with the associated flow, are shown in Figure 2.52a. The node potentials and reduced costs corresponding to this structure are presented in Figure 2.52b. In this case, arc $(2,4)$ violates the optimality conditions because $(2,4) \in \boldsymbol{U}$ and $c_{24}^{\pi} > 0$. We add arc $(2,4)$ to $\boldsymbol{T}$, which creates the cycle 1-2-4-3-1. Since $(2,4) \in \boldsymbol{U}$, we wish to decrease the flow along $(2,4)$ by δ. If we decrease the flow along $(2,4)$ by δ, then we must decrease the flow along $(1,2)$ and increase the flow along $(1,3)$ and $(3,4)$ by δ. So it is easy to verify that the maximal value of δ is 3. After augmenting the flow around this cycle by 3 units, arc $(1,3)$ becomes restricted because $x_{13} = u_{13}$. We remove this arc from $\boldsymbol{T}$ and add it to $\boldsymbol{U}$.

The third spanning tree structure and the associated flow are shown in Figure 2.53a. As we can see in Figure 2.53b, the node potentials and reduced costs satisfy the optimality conditions. Hence, the flow is optimal.

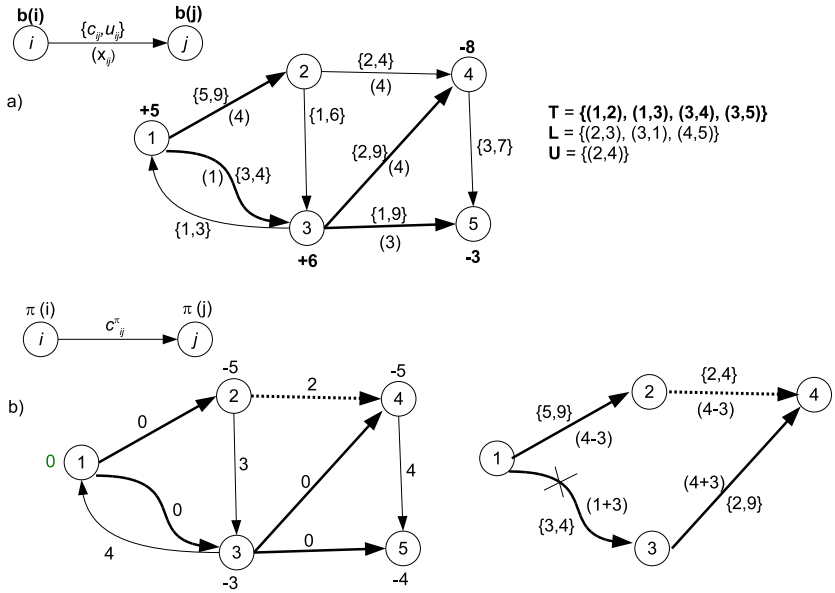


Figure 2.52: (a) A feasible spanning tree structure (b) Reduced arc costs

Correctness and the running time of the algorithm

It is clear that when the algorithm stops, then the resulting flow is optimal. This follows from the fact that it satisfies the optimality conditions from Theorem 17. Furthermore, if the flow around a cycle is augmented in line 5 of the algorithm by a positive amount, then the new flow has lower cost than the previous one. So, if we always augment the flow by a positive integer, then the algorithm reaches an optimal solution in a finite number of steps.

Unfortunately, in some cases $\delta = 0$, and after augmenting by 0 units, the new flow is exactly the same as the previous one - only the spanning tree structure is modified. Such augmentations are called *degenerate*. There are some instances for which the algorithm performs an infinite number of degenerate augmentations. So the algorithm does not necessarily terminate in a finite number of iterations. We can avoid this bad behavior by imposing some additional restrictions on the choice of the entering and leaving arcs. We do not describe the details here and they can be found, for example, in [3].

The network simplex is very efficient in practice. However, the number of iterations performed by the generic version of the algorithm may be exponential. This phenomenon is known as *stalling*. In the literature, several antistalling rules have been proposed and a polynomial time implementation of the network simplex algorithm was described in [39].

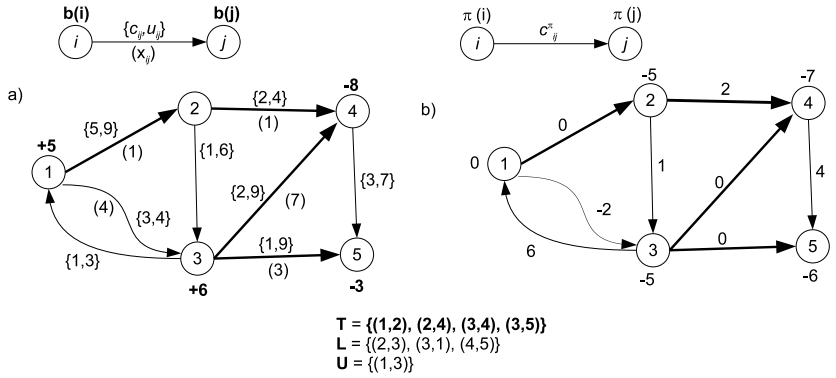


Figure 2.53: (a) A feasible spanning tree structure (b) Reduced arc costs

Sensitivity analysis

The network simplex algorithm has an additional nice property. Namely, it allows us to perform a *sensitivity analysis* of the optimal solution obtained. Suppose that we have computed an optimal flow f. We may now ask the following question: how much can the cost c_{ij} of a given arc $(i,j) \in A$ vary so that the flow f remains optimal?

Consider the example shown in Figure 2.54. In Figure 2.54a an optimal flow is shown. Suppose first that we would like to check how the costs of the original nontree arcs $(1,3)$ and $(4,5)$ can vary so that the flow remains optimal. The cost of arc $(1,3)$ is equal to 3. Assume that its new cost is $3+\lambda$. Similarly, the original cost of $(4,5)$ is 8 and assume that its new cost is $8+\lambda$. Since the arcs considered are nontree ones, their costs do not influence the computations of the node potentials. So the node potentials do not depend on λ and they are the same as for the original costs. The node potentials and the reduced costs are shown in Figure 2.54b. Arc $(1,3)$ belongs to $\boldsymbol{U}$, so its reduced cost must satisfy $c^{\pi}_{13} = -2+\lambda \leq 0$. This yields $\lambda \leq 2$, which means that f remains optimal if $c_{13} \in (-\infty, 5]$ with all the other costs unchanged. A similar reasoning can be carried out for arc $(4,5)$. In this case $(4,5) \in \boldsymbol{L}$, so $c_{45} = 4+\lambda \geq 0$. This gives $\lambda \geq -4$ and $c_{45} \in [-1, \infty)$.

Consider another example shown in Figure 2.55. Now we would like to perform the sensitivity analysis for the tree arc $(2,4)$. We thus assume that the cost of this arc is $2+\lambda$. This situation is a little more complex, because the node potentials depend on the cost of the tree arc $(2,4)$ and, consequently, also on λ. The node potentials and the reduced arc costs are shown in Figure 2.55b. We must now determine all the values of λ for which the optimality conditions are satisfied. For arcs $(2,3) \in \boldsymbol{L}$ and $(3,1) \in \boldsymbol{L}$, we get $1-\lambda \geq 0$, $6+\lambda \geq 0$ respectively and for arc $(1,3) \in \boldsymbol{U}$, we have $-2-\lambda \leq 0$. So, $\lambda \in [-2,1]$ and $c_{24} \in [0,3]$.

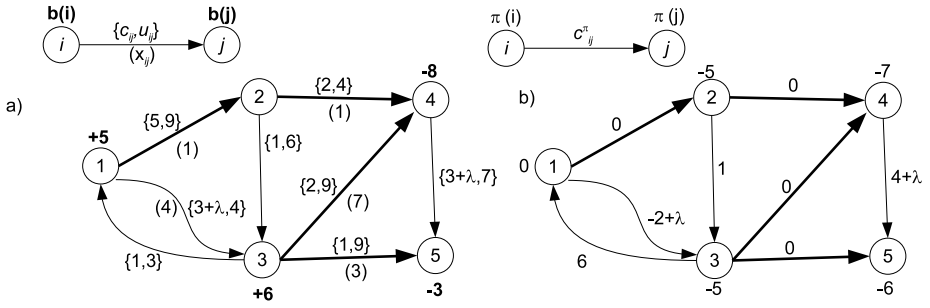


Figure 2.54: Sensitivity analysis for the nontree arcs $(1,3) \in \boldsymbol{U}$ and $(4,5) \in \boldsymbol{L}$.

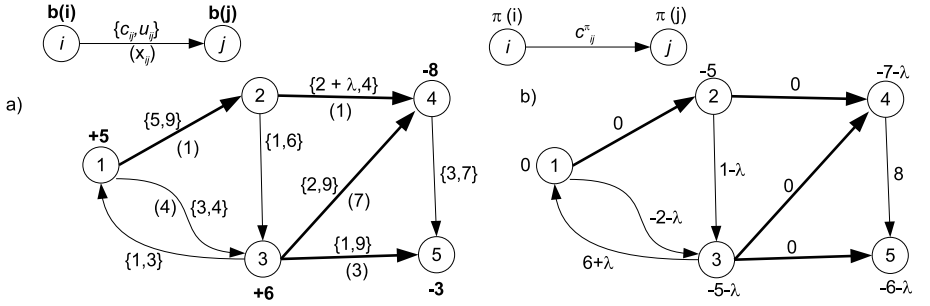


Figure 2.55: Sensitivity analysis for tree arc $(2,4) \in \boldsymbol{T}$.

2.3.5 Summary

1. In the minimum cost flow problem we seek a flow in a given network with the minimum total cost. The shortest path and the maximum flow problems are special cases of the minimum cost flow problem.

2. We assume that the problem is balanced, i.e. the total supply equals the total demand and there is no directed cycle of negative length in the input network.

3. The simplest algorithm that outputs an optimal flow is the cycle canceling algorithm. It iteratively decreases the cost of the current flow by augmenting the flow around directed cycles of negative cost in the residual network. However, this algorithm can be slow in practice.

4. The problem can be solved by using the network simplex algorithm. The network simplex algorithm maintains a feasible spanning tree structure and stops when the optimality conditions are satisfied. If all the arc capacities and node supply/demands are integer, then the algorithm returns an optimal solution with integer flows. The network simplex algorithm is

an adaptation of the simplex algorithm used to solve linear programming problems.

5. The network simplex algorithm is very efficient in practice. However, the basic version of the algorithm may not terminate in a finite time, due to degenerate augmentations. We can guarantee the finiteness of the algorithm by applying some rules for adding and deleting arcs to/from the current spanning tree structure.

6. The network simplex algorithm allows us to perform sensitivity analysis of the optimal solution obtained.

The implementation details of the cycle canceling and network simplex algorithms can be found, for example, in [14] and [45]. The cycle canceling algorithm is credited to Klein [31]. The network simplex algorithm was developed by Dantzig [16] by adapting the simplex algorithm for linear programming (see also Appendix B). The network simplex algorithm became popular in the early 1970s, when some its efficient implementations were discovered. The problem of degeneracy can be resolved by applying the strongly feasible spanning tree technique proposed by Cunningham [15]. This technique is also described in [3]. The basic version of the network simplex algorithm may perform an exponential number of iterations. A polynomial time implementation of this algorithm was developed by Orlin [39]. There exist some other algorithms for the minimum cost flow problem, such as successive shortest paths, primal - dual, out-of-kilter and capacity scaling. Comprehensive descriptions of these algorithms can be found in [3]. However, for most instances the network simplex algorithm is regarded as the fastest one.

2.4 Transportation problem

In this section we consider the following *transportation problem*. We are given a set of *suppliers* $A_1, A_2, \ldots, A_m$ with positive supplies $s_1, s_2, \ldots, s_m$ and a set of *customers* $B_1, B_2, \ldots, B_n$ with positive demands $d_1, \ldots, d_n$. We assume, at this point, that the total supply equals the total demand, i.e. $\sum_{i=1}^{m} s_i = \sum_{i=1}^{n} d_i$. The cost of transporting 1 unit from A_i to B_j equals c_{ij}. The flow $f = (x_{ij})$, $i = 1, \ldots, m$, $j = 1, \ldots, n$, in this problem must satisfy the following system of constraints:

$$\begin{array}{ll} \sum_{j=1}^{n} x_{ij} = s_i & i = 1, \ldots, m \\ \sum_{i=1}^{m} x_{ij} = d_j & j = 1, \ldots, n \\ x_{ij} \geq 0 & i = 1, \ldots, m \; j = 1, \ldots, n \end{array} \tag{2.2}$$

So, each supplier A_i must send s_i units and each customer B_j must receive d_j units. We wish to find a flow f of minimum cost $c(f) = \sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij} c_{ij}$.

The transportation problem can be seen as a special case of the minimum cost flow problem, in which the network has a special bipartite structure and all arc capacities are infinite. Namely, the nodes of the network represent suppliers and customers and the arcs lead only from suppliers to customers. An example is shown in Figure 2.56. There are two suppliers A_1 and A_2 with supplies 10 and 15 and three customers B_1, B_2 and B_3 with demands 5, 10 and 10 respectively. All the possible connections between the suppliers and customers are specified by a bipartite network G, whose arcs represent the connections between the suppliers and customers. Each arc has a cost c_{ij} and infinite capacity. In the sample flow, shown in Figure 2.56, A_1 sends 10 units to B_2, A_2 sends 5 units to B_1 and 10 units to B_3. The cost of this flow is $5 * 1 + 10 * 3 + 10 * 5 = 85$.

It is convenient to represent this problem in the form of a *transportation table*. The rows represent suppliers, the columns represent customers and inside the table we write the costs and a flow (see Figure 2.56).

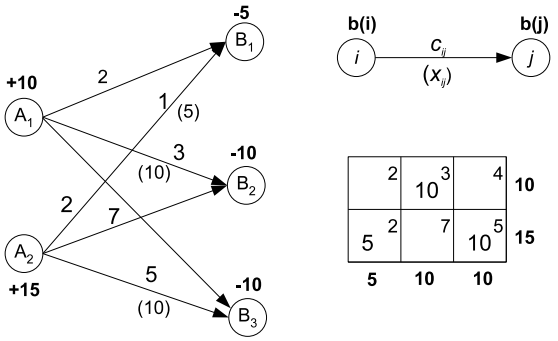


Figure 2.56: A network representation of the transportation problem and the corresponding transportation table.

Up to this point we have assumed that the problem is balanced, i.e. the equality $\sum_{i=1}^{m} s_i = \sum_{i=1}^{n} d_i$ holds. This condition need not be satisfied in some

applications. If $\sum_{i=1}^{m} s_i > \sum_{i=1}^{n} d_i$, then we can make the problem balanced by adding an *artificial customer* with demand equal to $\sum_{i=1}^{m} s_i - \sum_{i=1}^{n} d_i$ and all transportation costs equal to 0. Similarly, if $\sum_{i=1}^{m} s_i < \sum_{i=1}^{n} d_i$, then we can make the problem balanced by adding an *artificial supplier* with supply equal to $\sum_{i=1}^{n} d_i - \sum_{i=1}^{m} s_i$ and all transportation costs equal to 0. Sometimes, there may be no connection between supplier A_i and customer B_j. In this case we assume that the cost $c_{ij} = M$, where M is a sufficiently large constant representing infinity.

Consider the example shown in Figure 2.57. We can see that in the network presented the total supply equals 25, while the total demand equals 20. So we add an additional customer with demand 25-20=5. Also, there are no connections between A_1 and B_2 or A_2 and B_3. Therefore, we fix $c_{12} = M$ and $c_{23} = M$ in the transportation table, where M represents the infinite cost of the prohibited connections.

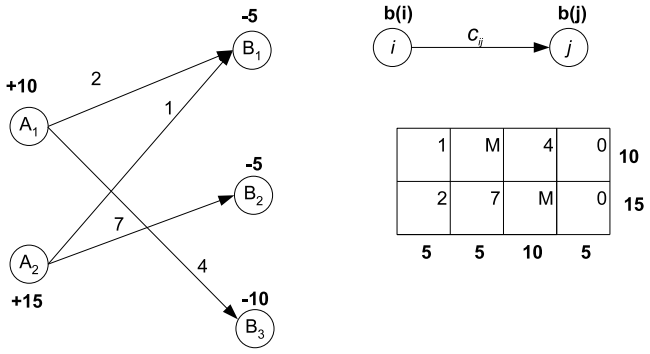


Figure 2.57: A sample problem, which is not balanced and some connections are prohibited.

If we solve this problem and, according to the solution obtained, there is a positive flow in a box with cost M, then the original problem has no solution. This means that there is no feasible flow in the network representing the problem.

2.4.1 Applications

Application 1 (production/transportation problem). A company has m factories. The ith factory produces s_i units of some product and the production cost of 1 unit equals q_i. The product is transported to n cities. The jth city demands d_j units and the price of 1 unit equals p_j. We assume that the total production of all the factories is not less than the total demand of all the cities, i.e. $\sum_{i=1}^{m} s_i \geq \sum_{i=1}^{n} d_i$. The cost of transporting 1 unit from the ith factory to the jth city equals c_{ij}. We would like to establish an optimal production and transportation plan for the company.

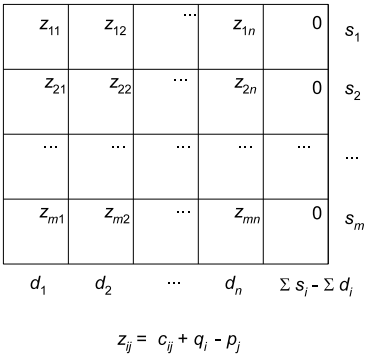


Figure 2.58: Transportation table for the production/transportation problem.

The transportation table for this problem is shown in Figure 2.58. The rows represent factories and the columns represent cities. The additional $n+1$th city is an artificial customer, whose demand is equal to $\sum s_i - \sum d_i$. The production costs and prices can be incorporated into the transportation table by computing the total cost of sending 1 unit from the ith factory to the jth city, which is equal to $z_{ij} = c_{ij} + q_i - p_j$. Notice that z_{ij} may be negative, which means that sending 1 unit yields a positive profit. A flow to the artificial city can be interpreted as an unused part of the production capability of the appropriate factory.

Application 2 (transshipment problem). A company has m factories and the ith factory produces s_i units of some product. The product must be delivered to n customers and the demand of the jth customer equals d_j. We assume that the problem is balanced. The product can be sent to customers by using $l \geq 0$ intermediate stores and the capacity of the kth store equals m_k. All the possible connections, together with their unit transportation costs, are given by network G. We would like to establish an optimal transportation plan of the product from the factories to the customers.

Consider the sample problem shown in Figure 2.59. There are three factories F_1, F_2 and F_3 with supplies 10, 5 and 8, two customers C_1, C_2 with demands 18 and 5, and two intermediate points, 1 and 2. The capacity of intermediate point 1 is equal to 12, which means that at most 12 units can pass through point 1. The capacity of intermediate point 2 is unlimited. The transportation table for this example is shown in Figure 2.59. Observe that the points that have both incoming and outcoming arcs, appear as both rows and columns in the transportation table. Consider, for instance factory F_2, whose supply is 5. It can, however, send more than 5 units, since it can receive something from other points in the network. So we define the supply of F_2 as $5 + s$ and the demand of F_2 as s, which means that F_2 will always send its 5 units (we will fix the value of s later). Now the crucial observation is that F_2 can send $x + 5$ units and receive x units, where $x \in [0, s]$, since the remaining $s - x$ units can

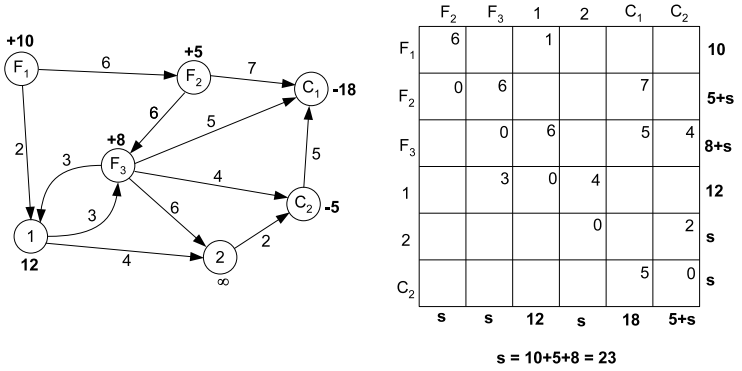


Figure 2.59: A sample transshipment problem. The empty boxes in the transportation table have costs equal to M.

be placed in the box (F_2, F_2), which corresponds to the artificial arc (F_2, F_2) with 0 cost. The optimal value of x can be obtained by solving the transportation problem. The value of s should be large enough and it may be equal, for instance, to the total supply $s = 5 + 8 + 10$. The reasoning for the remaining points in the network is similar, and we leave it as an exercise. Notice that the capacity of point 2 is unlimited so we fix its supply and demand to s.

Application 3 (multi-period production problem.) A company produces some product over K periods. In the ith period the factory can produce s_i units and the demand is equal to d_i (this demand must be fully satisfied). The cost of producing 1 unit in the ith period is equal to c_i. Unsold production can be kept in a store and sold later. The cost of keeping 1 unit in the store is constant and equals m per period. We would like to establish an optimal production plan.

The transportation table for this problem is shown in Figure 2.60. We create a row and a column for each period. Suppose that we produce the product in the ith period and sell it in the jth period. This is not possible if $j < i$, so in this case we fix $c_{ij} = M$. If $j \geq i$, then we must add the cost of keeping the product in the store to the production cost. Hence, $c_{ij} = c_i + (j - i)m$.

2.4.2 Network simplex

Since the transportation problem is a special case of the minimum cost flow problem, the network simplex algorithm, presented in Section 2.3.4, will also solve the transportation problem. As we will see, however, this algorithm can be simplified by using the facts that the network has a special bipartite structure and there are no limits on the arc capacities. In particular, the second fact implies that every spanning tree structure is represented as a partition $(\boldsymbol{T}, \boldsymbol{L})$, where $\boldsymbol{T}$ is the set of spanning tree arcs and $\boldsymbol{L}$ the set of arcs whose flow is equal to 0. There is no set $\boldsymbol{U}$, because the arcs have no upper bounds on

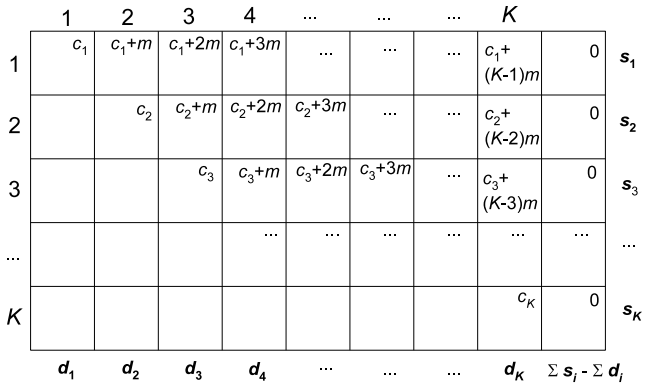


Figure 2.60: The transportation table for the multiperiod production problem. The empty boxes in the transportation table have costs equal to M.

their flow. Furthermore, all the computations can be directly performed on the transportation table and it is not necessary to store the network structure. We will however keep the underlying network G in mind and make the following three assumptions:

1. The problem is balanced.
2. The network G is a complete bipartite network, i.e. it contains an arc between each supplier/customer pair.
3. All supplies and demands are positive.

These assumptions are not restrictive. We can always make the problem balanced and make G complete by using the transformations shown in Section 2.4. Also, if a supply (demand) node has 0 supply (demand), then such a node can be removed from the network without changing the problem.

Computing an initial solution

Recall that the network simplex starts with an initial feasible spanning tree structure and the solution (flow) corresponding to this structure. In the transportation problem, there exists a simple method of establishing a feasible spanning tree structure $(\boldsymbol{T}, \boldsymbol{L})$, which does not require the addition of artificial arcs to the network. This method works as follows. We first represent the problem in the form of a transportation table and set $\boldsymbol{T} = \emptyset$. Let us choose any box (p, q) in the table and put a flow equal to $x_{pq} = \min\{s_p, d_q\}$ inside (p, q). So we make the flow from A_p to B_q as large as possible without exceeding the supply of A_p or the demand of B_q. We then modify the table by subtracting x_{pq} from both s_p and d_q. If $s_p = 0$ and $d_q > 0$, then we delete row p from the table. If $s_p > 0$ and $d_q = 0$, then we delete column q from the table. Finally, if $s_p = 0$

and $d_q = 0$, then we delete either row p or column q, but not both. In this case, if there is only one row in the table, then we delete a column and if there is only one column in the table, then we delete a row. We then select another box and repeat this procedure until the all the rows and columns are deleted.

Why does the presented method work? We must show two things: (a) after deleting all rows and columns we get a flow satisfying conditions (2.2) inside the transportation table, and (b) the obtained set of arcs $\boldsymbol{T}$ is a spanning tree of the underlying network. Point (a) can be proved by induction. If we remove row p from the table, then the flow must satisfy $\sum_{j=1}^{n} x_{pj} = s_p$ and no additional flow is later inserted in row p. A similar condition holds if we remove column q. Furthermore, after removing row p or column q, we get a smaller problem, which is also balanced. So, if the problem is finally reduced to a single box (k, l), then $s_k = d_l$ and the procedure finishes with a flow satisfying (2.2). To prove (b), we need the assumption that all the supplies and demands are positive. Then exactly $m + n - 1$ boxes are chosen and exactly $m + n - 1$ arcs are added to $\boldsymbol{T}$. Furthermore, the set of arcs in $\boldsymbol{T}$ must form a connected subgraph of G. Otherwise, some node would not send anything or some node would not receive anything, which would contradict the assumption that all supplies and demands are positive. A connected subgraph of G with $m+n$ nodes, containing $n+m-1$ arcs is a spanning tree of G.

Choosing the box to be filled is an arbitrary step in our procedure. In the literature several methods of choosing the sequence of boxes have been proposed. If we always choose the box located in the top left corner of the transportation table, then we get the *north-west corner method*, which is illustrated in Figure 2.61. Using this method we first choose box $(1, 1)$. After deleting row 1 we choose box (2,1) and so on. As a result, we get $\boldsymbol{T} = \{(1,1), (2,1), (2,2), (2,3), (3,3), (3,4)\}$ and $\boldsymbol{L}$ contains all the arcs, which do not belong to $\boldsymbol{T}$. The solution associated with $(\boldsymbol{T}, \boldsymbol{L})$ is shown in the last table (see Figure 2.61).

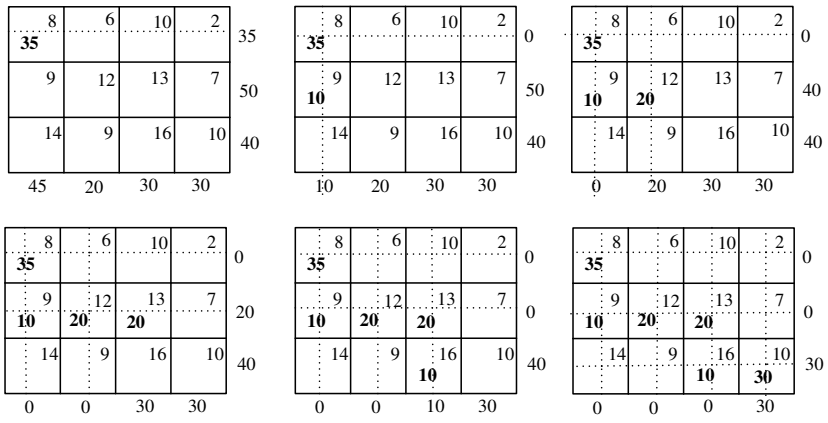


Figure 2.61: An illustration of the north-west corner method.

The north-west corner method is very fast. It, however, does not take into account the arc costs. In consequence, the first solution might be very poor. A better initial solution can be obtained by at each step choosing the box with the minimum cost. This is called the *minimum element cost* method and is illustrated in Figure 2.62. We first choose box $(1,4)$, which has the smallest cost. After removing column 4, we next choose box $(1,2)$ and so on. As a result, we get $\boldsymbol{T} = \{(1,4),(1,2),(2,1),(2,3),(3,3),(3,2)\}$ and $\boldsymbol{L}$ contains all the arcs, which do not belong to $\boldsymbol{T}$. The solution associated with $(\boldsymbol{T},\boldsymbol{L})$ is shown in the last table (see Figure 2.62).

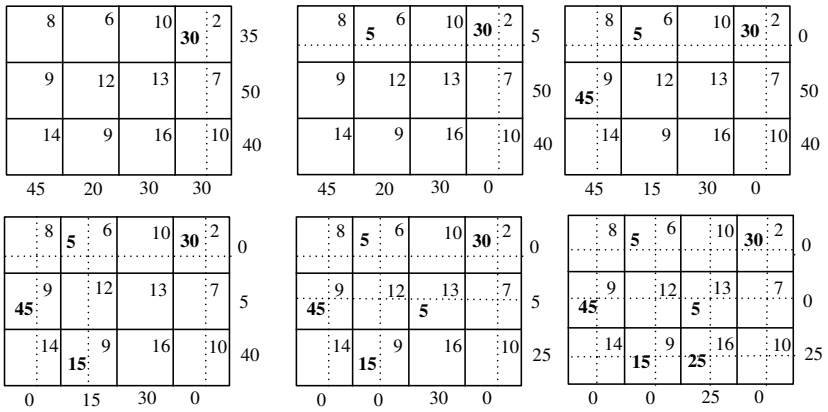


Figure 2.62: An illustration of the minimum element cost method.

The network simplex algorithm

Consider the initial solution obtained by means of the north-west corner method, shown in Figure 2.63a. In this figure, the associated spanning tree arcs $\boldsymbol{T}$ are also shown. The empty boxes correspond to the arcs in $\boldsymbol{L}$ and the flow along these arcs equals 0. Let us recall that in the next step of the network simplex algorithm we have to compute the node potentials. These computations can be performed in the same way as in Section 2.3.4. We only change the notation. We use $\alpha(i)$ to denote the potential of supply node A_i and $\beta(j)$ to denote the potential of demand node B_j. We must set the potentials so that the reduced costs of all the tree arcs are 0. Hence, the potentials must satisfy the following system of equations:

$$
\begin{aligned}
8 - \alpha(1) + \beta(1) &= 0\\
9 - \alpha(2) + \beta(1) &= 0\\
12 - \alpha(2) + \beta(2) &= 0\\
13 - \alpha(2) + \beta(3) &= 0\\
16 - \alpha(3) + \beta(3) &= 0\\
10 - \alpha(3) + \beta(4) &= 0
\end{aligned}
$$

We can easily solve this system by setting $\alpha(1) = 0$, which uniquely determines the values of all the remaining potentials. The computed potentials are shown in Figure 2.63a. In the next step we compute the reduced costs $c_{ij} - \alpha(i) + \beta(j)$ of all the other arcs. These reduced costs are shown in the table presented in Figure 2.63b. Notice that the reduced costs of all tree arcs are equal to 0 as required.

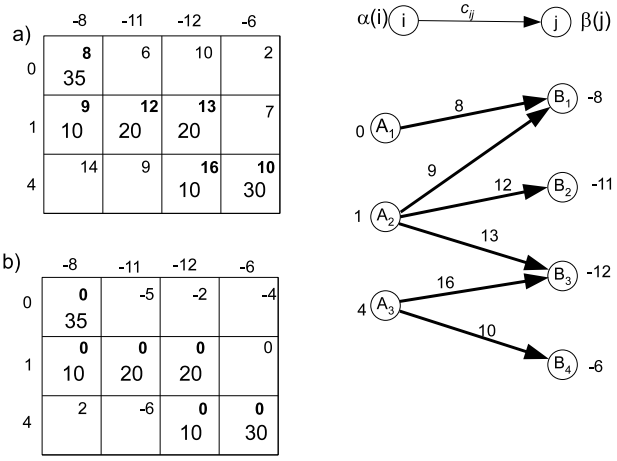


Figure 2.63: (a) The initial spanning tree solution, together with node potentials (b) The reduced arc costs.

We can see that the reduced costs do not satisfy the optimality conditions (Theorem 17), since they take negative values for some $(i, j) \in \boldsymbol{L}$ (i.e there are some negative values in the empty boxes of the transportation table). We can choose, for instance, arc $(A_3, B_2) \in \boldsymbol{L}$, whose reduced cost is -6, and add this arc to $\boldsymbol{T}$. This creates the unique cycle composed of arcs (A_3, B_2), (A_2, B_2), (A_2, B_3) and (A_3, B_3) (see Figure 2.64a). This cycle can be represented as a polygon in the transportation table. The box (A_3, B_2) and some boxes corresponding to the arcs of the spanning tree are located in the corners of this polygon. We can now easily determine the value of δ, the change in the flow around the cycle. If we add δ to box (A_3, B_2), then we must subtract δ from (A_2, B_2), add δ to (A_2, B_3) and subtract δ from (A_3, B_3). So we can mark the boxes of the cycle alternately with $+$ and $-$, starting by marking the box (A_3, B_2) with $+$ (see Figure 2.64a). Now it is easy to see that δ is the minimum among the values contained in the boxes marked with $-$, which in our case is equal to 10. We change the flow around the cycle (increasing or decreasing according to the sign) and the flow in box (A_3, B_3) falls to 0. Hence, arc (A_3, B_3) is removed from $\boldsymbol{T}$ and added to $\boldsymbol{L}$. The new spanning tree solution is shown in Figure 2.64b.

In Figure 2.65 the new spanning tree solution is shown. We again compute the node potentials and reduced costs for this solution, which are shown

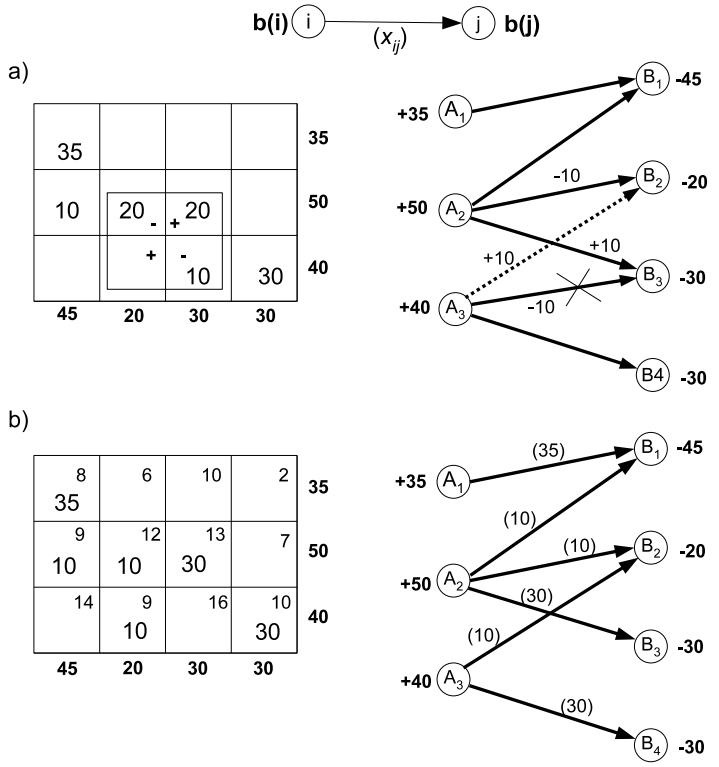


Figure 2.64: (a) Changing the flow around the cycle. (b) The next spanning tree solution.

in Figure 2.65b. We can see that some arcs in $\boldsymbol{L}$ still violate the optimality conditions. We choose such an arc, say (A_1, B_4) and add it to $\boldsymbol{T}$. This creates the unique cycle composed of 6 arcs shown in Figure 2.66a. As previously, we can represent this cycle as a polygon in the transportation table. We mark the box (A_1, B_4) with $+$ and the remaining corners of this polygon alternately with $-$ and $+$. The minimum value of the flow among the boxes marked with $-$ is 10, so we must change the flow around this cycle by 10 units. The flow along arc (A_2, B_2) falls to 0 and this arc is removed from $\boldsymbol{T}$ and added to $\boldsymbol{L}$. As a result, we get the next spanning tree solution shown in Figure 2.66b. We must now again compute the node potentials for this solution and check whether the optimality conditions are satisfied, i.e. the reduced costs of all the arcs in $\boldsymbol{L}$ are nonnegative. We leave this as an exercise.

Correctness of the algorithm

If the algorithm stops, then its correctness follows from the analysis described in Section 2.3.4. Unfortunately, the basic version of the algorithm may not ter-

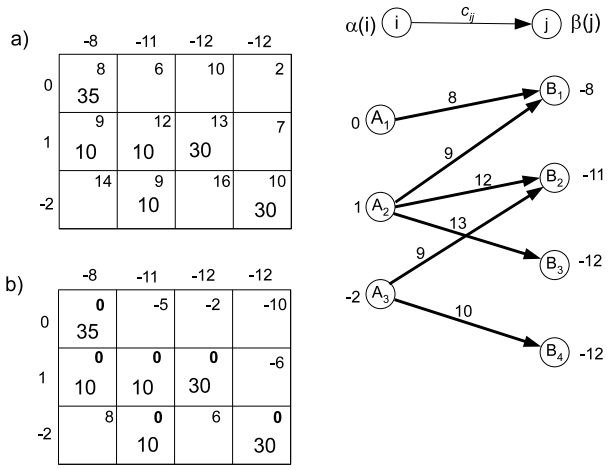


Figure 2.65: (a) The next spanning tree solution with node potentials. (b) The reduced arc costs.

minate in a finite time due to degeneracy. Consider the sample problem shown in Figure 2.67, in the first table the initial solution obtained by means of the north-west corner method is shown. Notice that the flow along some spanning tree arcs is 0 and we must distinguish these arcs from the arcs belonging to $\boldsymbol{L}$. After computing the node potentials, the reduced cost in box $(3, 1)$ is negative, so arc (A_3, B_1) is added to the spanning tree. However, no change can be made to the flow around the cycle obtained. Therefore, only the current spanning tree structure is changed (arc (A_3, B_1) is added to $\boldsymbol{T}$, arc (A_2, B_2) is added to $\boldsymbol{L}$ and the cost of the new flow is the same as the cost of the previous one. In the worst case, the algorithm can perform infinite sequence of such degenerate augmentations. There are some methods described in the literature (see e.g. [3]), which allow us to avoid such bad behavior.

Sensitivity analysis

We can perform a sensitivity analysis of the optimal solution computed, similarly to the sensitivity analysis shown in Section 2.3.4. Consider the optimal solution shown in the table in Figure 2.68a. Assume that we would like to check how much the cost of the nontree arc (A_1, B_4) can vary such that the solution remains optimal. After computing the node potentials, the reduced cost of (A_1, B_4) is $3 + \lambda$. In consequence, the solution remains optimal if $\lambda \geq -3$ or, equivalently, $c_{14} \in [6, \infty)$. Assume now that we would like to check how the cost of the tree arc (A_2, B_2) can vary. Now the computations are more complex, because the node potentials depend on λ (see Figure 2.68b). After computing the node potentials and the reduced costs, we get $1 - \lambda \geq 0$ and $15 - \lambda \geq 0$. Hence $\lambda \leq 1$ and $c_{22} \in (-\infty, 13]$.

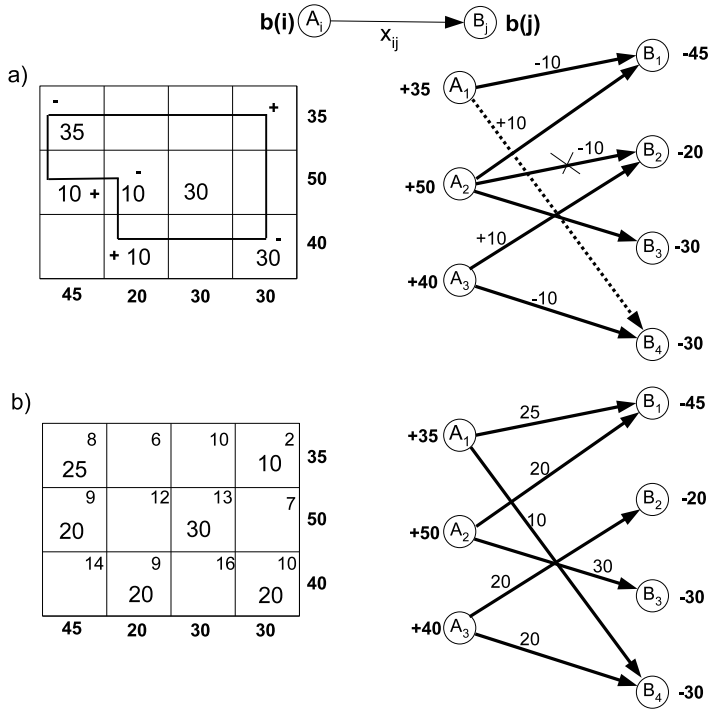


Figure 2.66: (a) Changing the flow around the cycle. (b) The next spanning tree solution.

2.4.3 Summary

1. In the transportation problem we wish to send some product directly from suppliers to customers at the minimal total cost.

2. The problem is a special case of the minimum cost flow problem, where the input network has a bipartite structure and all arc capacities are infinite.

3. The problem can be naturally represented in the form of a transportation table.

4. Only balanced problems have feasible solutions. If a problem is not balanced, then we make it balanced by adding an artificial supplier or artificial customer, as appropriate.

5. We can apply a simplified version of the network simplex algorithm to solve the problem. As in the general case, degeneracy may appear, which can be avoided by imposing some additional restrictions on the way in which the current spanning tree structure is modified.

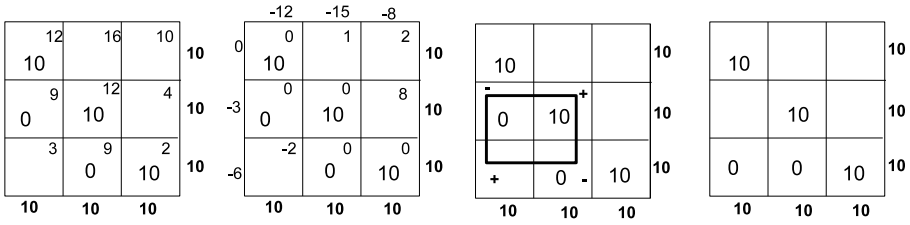


Figure 2.67: A degenerate augmentation.

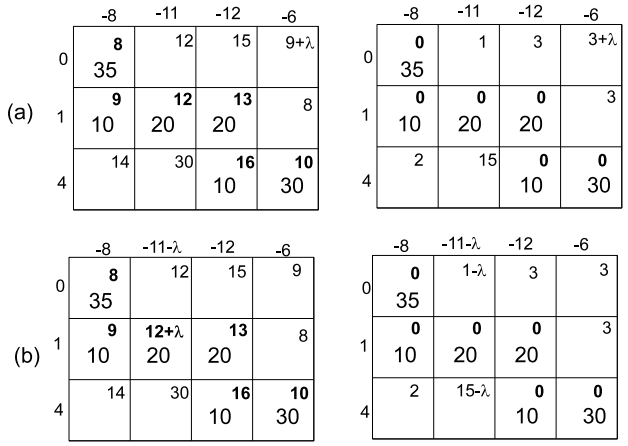


Figure 2.68: (a) Sensitivity analysis for the nontree arc (A_1, B_4). (b) Sensitivity analysis for the tree arc (A_2, B_2).

6. We can easily perform a sensitivity analysis of the obtained optimal solution.

   The transportation problem has a long history. It was first considered in 1941 by Hitchcock [26] and is sometimes known as the Hitchcock problem. Of course, it is a special case of the minimum cost flow problem. In consequence, it can be solved efficiently by using the network simplex algorithm. Interestingly, every minimum cost flow problem can be transformed into the transportation problem. This transformation was first described in [48] and can also be found in [41].

2.5 Minimum cost assignment

We are given a set of objects $A = \{A_1, A_2, \ldots, A_n\}$ and a set of objects $B = \{B_1, B_2, \ldots, B_n\}$, where $|A| = |B| = n$. We wish to pair the objects from A and B, so that each object from A is paired with exactly one object from B and vice versa. Such a pairing is called an *assignment*. The cost of pairing objects A_i to B_j equals c_{ij} and we wish to find an assignment with the minimum total cost. A sample problem is shown in Figure 2.69. Sets A and B contain 3 objects. One possible assignment is (A_1, B_2), (A_2, B_1) and (A_3, B_3). Its total cost is equal to 11. It is easy to see that the minimum cost assignment problem is a very special case of the transportation problem discussed in the previous section. We can treat A as a set of suppliers with supplies equal to 1 and B as a set of customers with demands equal to 1. By solving the transportation problem, we get an optimal assignment.

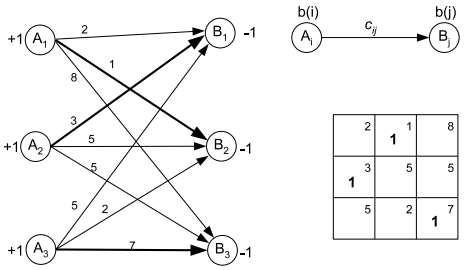


Figure 2.69: A sample assignment problem as a transportation problem.

In this section will use another representation of this problem, which is illustrated in Figure 2.70.

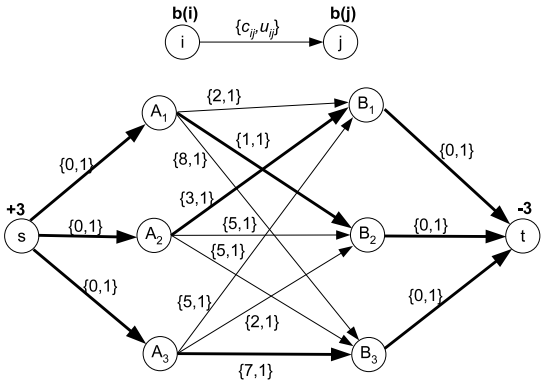


Figure 2.70: A sample assignment problem as a minimum cost flow problem.

We add a source node s with supply n and a sink node t with demand n. We add arcs (s, A_i) and (B_i, t) for $i = 1, \ldots, n$ with costs 0 and capacities 1. The arcs (A_i, B_j) have costs c_{ij} and capacities equal to 1. It is easy to see that an integer flow f with the minimum cost in this network corresponds to an optimal assignment. The flow on each arc can be either 0 or 1 and the arcs between A_i and B_j whose flow is equal to 1 define the minimum cost assignment.

2.5.1 Applications

Application 1 (assigning jobs to machines) A factory has n machines and n jobs to be completed. Each machine must be assigned to one job. The time required to set up the ith machine for the jth job is t_{ij}. The factory wants to minimize the total setup time needed for the n jobs. This is clearly an assignment problem and can be represented in the form of the table shown in Figure 2.71.

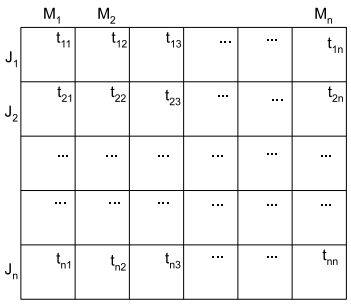


Figure 2.71: The table for Example 1.

2.5.2 Successive shortest path algorithm

Since the minimum cost assignment problem is a special case of the minimum cost flow problem, we could use the network simplex algorithm to obtain an optimal solution (recall that the network simplex always returns an integer solution). However, the special structure of the problem allows us to design a simple and more efficient algorithm. Our strategy will be as follows. We represent the problem as the minimum cost flow problem shown in Figure 2.70. We start by assuming that $b(s) = b(t) = 0$ and iteratively increase $b(s)$ and $b(t)$ by 1 until $b(s) = b(t) = n$. In each iteration we maintain a residual network for the current flow in which all the arc costs are nonnegative. Hence, the last residual network cannot contain any directed cycle of negative length and corresponds to an optimal flow (assignment).

Consider the sample problem shown in Figure 2.72, which is represented as a minimum cost flow problem. Initially, we assume that $b(s) = b(t) = 0$ and, consequently, the flow f equal to 0 on every arc, is optimal. Of course, the residual network $G(f)$ is the same as G. Let $d(i)$ be the shortest distance from

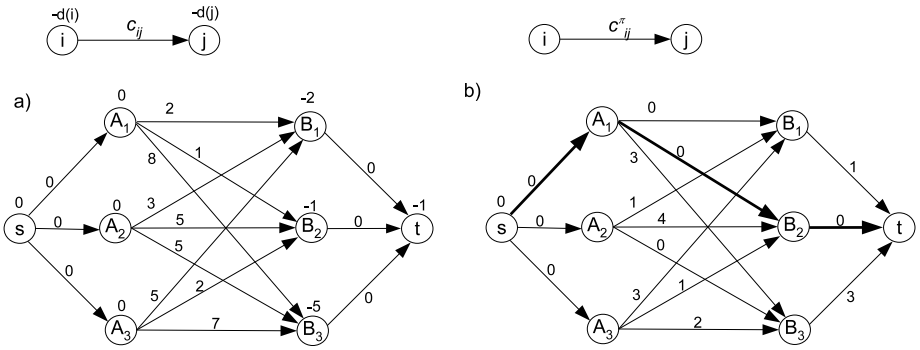


Figure 2.72: (a) Residual network for the initial flow and shortest distances (b) Reduced costs and the shortest augmenting path.

node s to node i in $G(f)$. Let us define the potential of node i as $\pi(i) = -d(i)$ (see Figure 2.72a) and compute the reduced arc costs $c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j) = c_{ij} + d(i) - d(j)$ (see Figure 2.72b). We claim that $c^{\pi}_{ij} \geq 0$ for all arcs (i, j). Indeed, if $c^{\pi}_{ij} < 0$, then $d(j) > c_{ij} + d(i)$, which would contradict the fact that $d(j)$ is the shortest distance from s to j. Furthermore, $c^{\pi}_{ij} = 0$ for all the arcs belonging to the shortest path $p = s - A_1 - B_2 - t$ from s to t. We can now augment the flow from s to t along p by 1 unit and modify the flow f. Observe that all arc costs in the new residual network $G(f)$ are nonnegative (see Figure 2.73a).

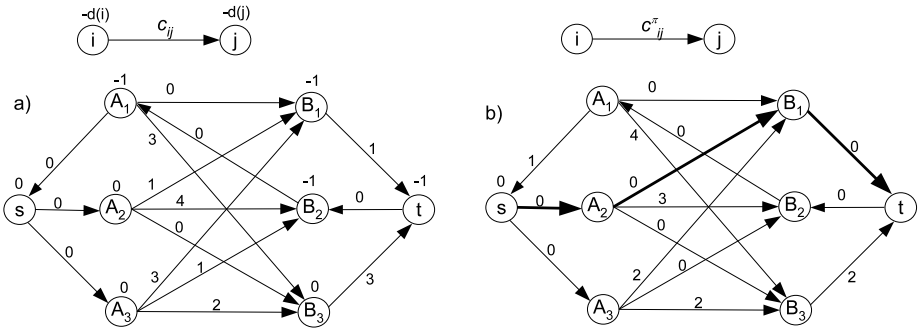


Figure 2.73: (a) Residual network for the second flow and shortest distances (b) Reduced costs and the shortest augmenting path.

We can now repeat the same procedure for the network $G(f)$. So we compute the shortest distance $d(i)$ from s to i in $G(f)$, set $\pi(i) = -d(i)$ (see Figure 2.73a) and compute the reduced costs c^{π}_{ij} for all arcs (i, j) (see Figure 2.73b). Now the shortest path from s to t, composed of arcs with zero reduced costs, is

$s - A_2 - B_1 - t$ and we augment the flow by 1 unit along this path. All arc costs in the new residual network $G(f)$ must be nonnegative (see Figure 2.74a).

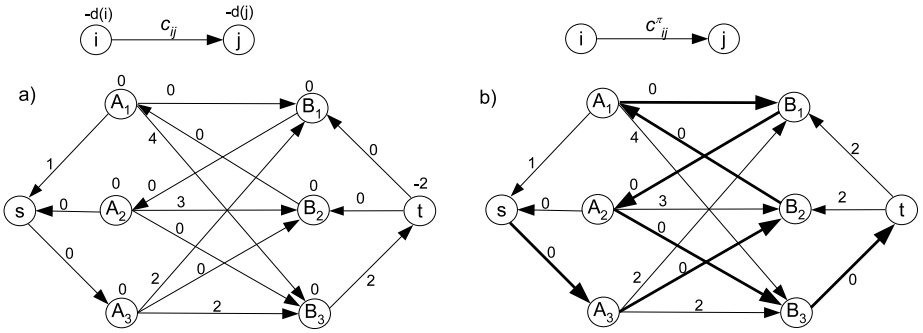


Figure 2.74: (a) Residual network for the third flow and shortest distances (b) Reduced costs and the shortest augmenting path.

The last iteration is shown in Figure 2.74. We augment the flow by 1 unit along the path shown in Figure 2.74b. After this, we get the final flow f and the residual network $G(f)$ is shown in Figure 2.75. The flow has the value of 3 (i.e. 3 units are sent from s to t) and $G(f)$ does not contain any directed cycle of negative cost. Hence, f is optimal. We can easily retrieve the minimum cost assignment by looking at arcs leading from B_j to A_i in $G(f)$. If there is an arc (j, i), then arc (i, j) has a flow equal to 1 and A_i is paired with B_j. So the minimum cost assignment in the example considered is (A_1, B_1), (A_2, B_3), (A_3, B_2).

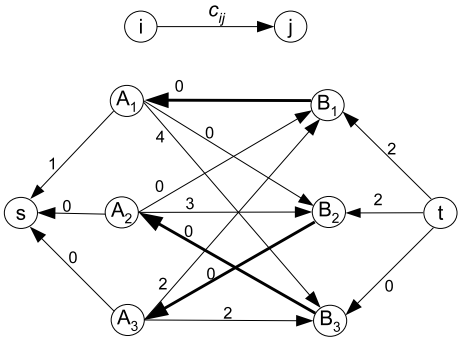


Figure 2.75: Residual network for the last flow.

Correctness and running time of the algorithm

Theorem 18 *The successive shortest path algorithm solves the minimum assignment problem in $O(n^3)$ time.*

Proof. The correctness of the algorithm follows from three observations. (1) As we know from the previous chapters, replacing the original costs with reduced ones does not change the problem. So, any such transformation of the costs does not change the optimal solution of the problem. (2) In each iteration we increase the value of the flow in the network G by 1. Starting with a flow of value 0, we iteratively increase it to n and a flow of value n clearly exists. In consequence, the last network must represent some assignment. (3) Finally, all the residual networks contain only nonnegative arc costs. Therefore, none of them can contain a directed cycle of negative cost. By Theorem 12, the solution obtained must be optimal. The algorithm performs exactly n augmentations, each time increasing the flow by 1. In each iteration we must solve the shortest path problem in a network with nonnegative arc costs. This can be done in $O(n^2)$ time by using Dijkstra's algorithm. So the overall running time of the algorithm is $O(n^3)$. ■

2.5.3 Summary

1. In the assignment problem we wish to pair objects from two sets A and B at the minimum total cost.

2. The minimum cost assignment problem can be seen as a special case of the transportation problem and, consequently, as a special case of the minimum cost flow problem.

3. This very special structure of the problem allows us to design an efficient successive shortest path algorithm for it. This algorithm outputs an optimal solution in $O(n^3)$ time.

The minimum cost assignment problem has been a very popular research topic within the operations research community. The first algorithm for this problem was developed by Kuhn [33]. A survey on assignment algorithms can be found in [9]. The successive shortest path algorithm, which can be applied to solve a general minimum cost flow problem, was developed in [29] and its description can also be found in [3].

2.6 Minimum spanning tree

We are given a connected undirected network $G = (N, A)$, $|N| = n$, $|A| = m$, with a cost c_{ij} associated with each arc $(i, j) \in A$. We wish to find a spanning tree of G that has the smallest total cost. Recall that a *spanning tree* is an acyclic subnetwork of G that connects all the nodes of G. A sample problem is shown in Figure 2.76. The arcs, $(1, 3)$, $(1, 5)$, $(2, 3)$, $(3, 6)$, $(4, 6)$ form a spanning tree of the minimum total cost equal to 9.

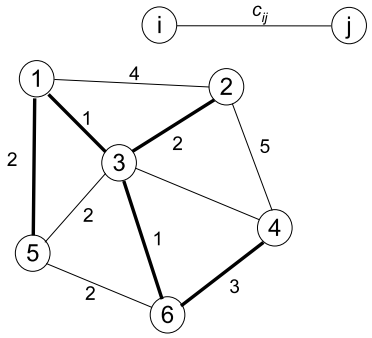


Figure 2.76: A sample network with the minimum spanning tree shown in bold.

We will denote a spanning tree of G by T. We will refer to the arcs contained in T as *tree arcs* and to the arcs not contained in T as *nontree arcs*. The following property of a spanning tree is easy to establish:

Observation 19 *Let T be a spanning tree of a network $G = (N, A)$. Then the following statements are true:*

1. *T has precisely $n - 1$ arcs.*
2. *For every nontree arc (k, l), the spanning tree T contains a unique path from k to l. Furthermore, the arc (k, l) together with this unique path forms a cycle.*
3. *If we delete any tree arc (i, j) from T, the resulting graph partitions the node set into two connected components S and $\overline{S}$. Thus $[S, \overline{S}]$ is a cut in G.*

We can illustrate the above observations using the example shown in Figure 2.77. The sample network has 6 nodes, so every spanning tree has exactly 5 arcs. Consider the nontree arc $(5, 6)$. The spanning tree T contains a unique path $5 - 1 - 3 - 6$ from 5 to 6 and adding $(5, 6)$ to this path creates a cycle $5-1-3-6-5$ (see Figure 2.77a). If we remove arc $(3, 6)$ from T, then the spanning tree splits into two connected components $S = \{1, 2, 3, 5\}$ and $\overline{S} = \{4, 6\}$. The partition $[S, \overline{S}]$ is a cut in G (see Figure 2.77b).

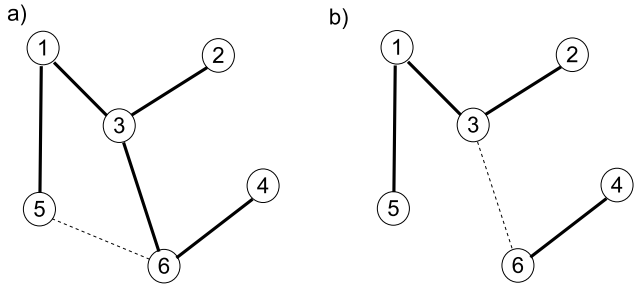


Figure 2.77: An illustration of Observation 19.

2.6.1 Applications

Application 1 (designing physical systems [3]). In all the problems described below we have to construct a minimum spanning tree in some network.

1. We would like to connect terminals in cabling panels of electrical equipment. We should wire the terminals to use the least possible length of wire.

2. We wish to construct a pipeline network to connect a number of towns using the smallest possible total length of pipeline.

3. We would like to link isolated villages in a remote region, which are connected by roads, but not yet by telephone. We wish to determine along which roads we should place telephone lines, using the minimum possible length of lines to link each pair of villages.

4. We have to connect a number of computer sites by high speed lines. We wish to determine a configuration that connects all the sites at the least possible cost.

2.6.2 Kruskal's algorithm

The algorithm presented in this section is based on the following theorem:

Theorem 20 ([3]) *A spanning tree T^* is a minimum spanning tree if and only if for every nontree arc (k,l) of G, $c_{ij} \leq c_{kl}$ for every arc (i,j) contained in the path in T^* connecting k and l.*

Theorem 20 leads to the very simple algorithm presented in Figure 2.78. Before we justify the correctness of this algorithm, we present how it works using the simple example shown in Figure 2.79.

Initially $T = \emptyset$ and E contains all the arcs of the input network G. In the first step we choose arc $(1,4)$ in E, because it has the smallest cost, equal to 1. Since $T \cup \{(1,4)\} = \{(1,4)\}$ is acyclic, we add arc $(1,4)$ to T (see Figure 2.79a).

1: $T := \emptyset$, $E := A$
2: **while** $|T| < |N| - 1$ **do**
3: Select arc (i, j) of the minimum cost from E and remove it from E
4: **if** $T \cup \{(i, j)\}$ is acyclic **then** $T := T \cup \{(i, j)\}$
5: **end while**

Figure 2.78: Kruskal's algorithm.

In the next step we choose arc $(2, 4)$ from E. Again, $T \cup \{(1, 4), (2, 4)\}$ is acyclic so we add $(2, 4)$ to T (see Figure 2.79b). In a similar way, arc $(3, 5)$ is added to T as shown in Figure 2.79c. In the next step we select arc $(1, 5)$. At this point, however, the set $T \cup \{(1, 5)\} = \{(1, 4), (2, 4), (3, 5), (1, 5)\}$ contains a cycle, so we skip arc $(1, 5)$ and select another arc, which is $(2, 3)$ (see Figure 2.79d). Proceeding in this way, we get the spanning tree shown in Figure 2.79e.

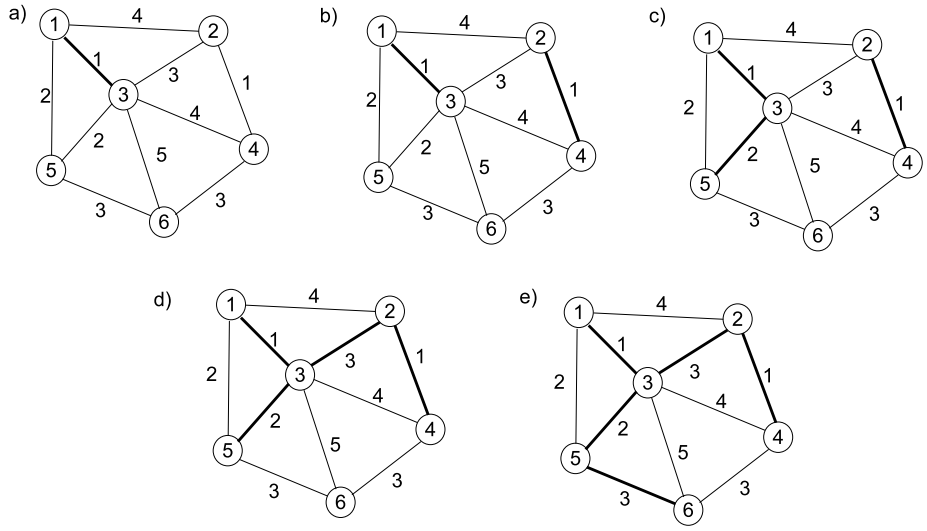


Figure 2.79: Illustration of Kruskal's algorithm.

Correctness and running time of the algorithm

Theorem 21 *Kruskal's algorithm computes a minimum spanning tree of a given network G in $O(m \log n)$ time.*

Proof. Kruskal's algorithm clearly returns a spanning tree of G, because the returned subset of arcs T is an acyclic subgraph of G and $|T| = n - 1$. The algorithm adds arcs to T in order of nondecreasing costs. Consider a nontree arc (k, l). Two cases are possible. (a) The algorithm terminates before the arc (k, l) is chosen from E in line 3. In this case, all the tree arcs have costs not

greater than c_{kl} and (k,l) satisfies the optimality condition from Theorem 20. (b) The arc (k,l) is chosen in line 3 of the algorithm. Since $(k,l) \notin T$, it must form a cycle with the arcs previously added to T. However, the costs of these arcs are not greater than c_{kl} and (k,l) satisfies the optimality condition from Theorem 20. Kruskal's algorithm is very simple. However, its implementation details may be quite involved. We may start by sorting the arcs of G, which requires $O(m \log m) = O(m \log n)$ time. We then examine the arcs one by one, each time checking whether there is a cycle in $T \cup \{(i,j)\}$, where (i,j) is the examined arc. Such a cycle can be detected in $O(n)$ time, so this naive implementation runs in $O(mn)$ time. This running time can be improved by using more sophisticated data structures. One of the simplest implementations runs in $O(m \log n)$ time, but there are implementations whose running time is nearly linear with respect to m (see e.g. [3]). ■

2.6.3 Prim's algorithm

The algorithm presented in this section is based on the following theorem:

Theorem 22 ([3]) *A spanning tree T^* is a minimum spanning tree if and only if for every tree arc $(i,j) \in T$, $c_{ij} \leq c_{kl}$ for every arc (k,l) contained in the cut formed by deleting arc (i,j) from T^*.*

Using Theorem 22, we can construct another simple algorithm for the problem. This algorithm, called Prim's algorithm, is shown in Figure 2.80. We illustrate how it works using the example shown in Figure 2.81.

1: $T := \emptyset$, $S := \{1\}$
2: **while** $|T| < |N| - 1$ **do**
3: Select arc (i,j) of the minimum cost from the cut $[S, \overline{S}]$.
4: $T := T \cup \{(i,j)\}$
5: $S = S \cup \{i,j\}$
6: **end while**

Figure 2.80: Prim's algorithm.

Initially $T = \emptyset$ and $S = \{1\}$ (in fact, we could use any node to initialize S). The initial cut $[S, \overline{S}]$ is shown in Figure 2.81a, this cut contains arcs $(1,2)$, $(1,3)$, $(1,5)$ and arc $(1,3)$ has the smallest cost equal to 1. So, we add $(1,3)$ to T and update $S := S \cup \{1,3\} = \{1,3\}$. The next cut $[S, \overline{S}]$ is shown in Figure 2.81b. It contains arcs $(1,2)$, $(2,3)$, $(3,4)$, $(3,5)$ and $(3,6)$. The arc $(3,6)$ has the smallest cost, so we add it to T and update $S := S \cup \{3,6\} = \{1,3,6\}$. We proceed in this way until $|T| = n-1$ or, equivalently, S contains all the nodes of the network G.

Correctness and running time of the algorithm

Theorem 23 *Prim's algorithm computes a minimum spanning tree of a given network G in $O(m + n \log n)$ time.*

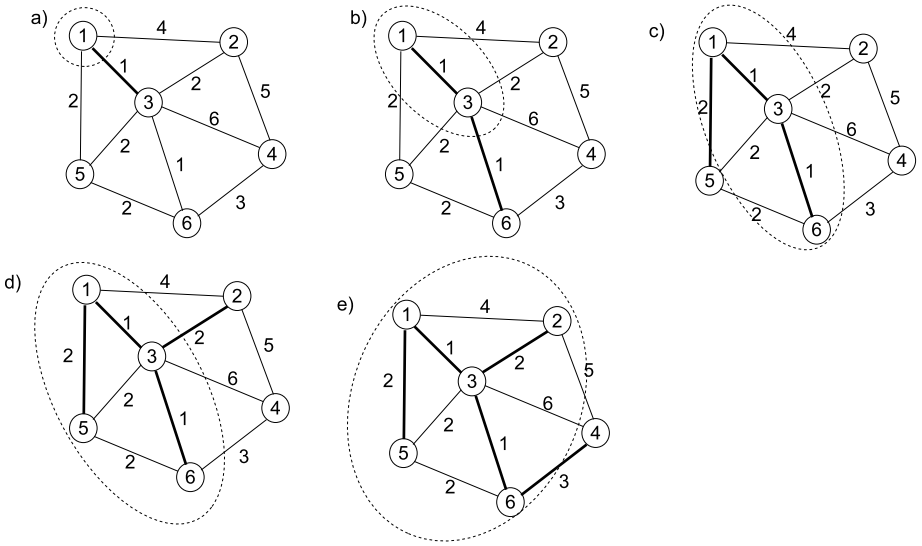


Figure 2.81: Illustration of Prims's algorithm.

Proof. It is easy to see that Prim's algorithm returns a spanning tree of G. It follows from the fact that exactly $n-1$ arcs are added to T and no cycle can be created. We need to show that this tree is the minimum one. Consider a tree arc $(i,j) \in T$. If we remove (i,j) from T, then T splits into two connected components S and $\overline{S}$, which define cut $[S,\overline{S}]$ in G. In line 3 of the algorithm we assume that the arc (i,j) has the smallest cost among all the arcs in the cut $[S,\overline{S}]$. Therefore, the arc (i,j) satisfies the optimality conditions from Theorem 22. Similarly to Kruskal's algorithm, a naive implementation of Prim's algorithm runs in $O(mn)$ time. The algorithm performs $n-1$ steps, each time scanning in $O(m)$ time the list of arcs to find the arc of minimum cost in the cut $[S,\overline{S}]$. Using a more sophisticated implementation, we can achieve a running time of $O(m + n \log n)$ (see e.g. [3]). ■

2.6.4 Summary

1. In the minimum spanning tree problem we seek a spanning tree of a given network G whose total cost is minimal.

2. The problem arises in applications where we wish to connect all the nodes of some network at the minimum possible cost.

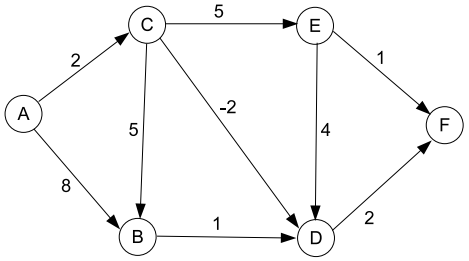
3. The problem can be solved by several efficient algorithms, for example Kruskal's and Prim's algorithms.

The minimum spanning tree is one of the most extensively studied problems in operations research. It is also one of the oldest discrete optimization problems,

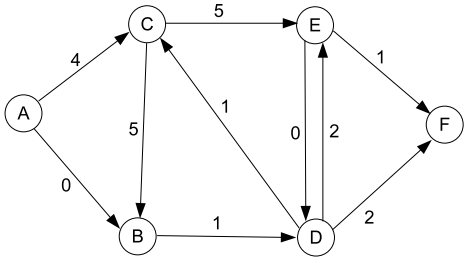
for which an algorithmic solution was proposed. The problem was formulated and solved by Boruvka in 1926 [38]. Later, these algorithms were rediscovered by Kruskal [32] and Prim [42]. Both algorithms belong to the class of greedy algorithms. The fastest algorithm to date was developed by Chazelle [10].It is almost linear with respect to the number of arcs. The implementation details of Kruskal's and Prim's algorithms can be found in [14] and [45]. A survey of the algorithms for the minimum spanning tree problem can be found in [8].

2.7 Exercises

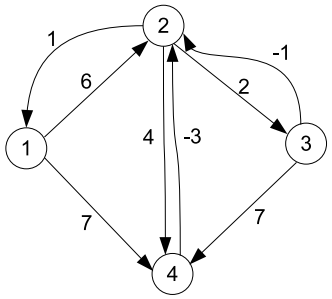
1. Using the dynamic algorithm, compute the tree of shortest paths and the tree of longest paths from node A in the following network (first establish a topological ordering of the nodes of this network):



2. Using Dijkstra's algorithm, compute the tree of shortest paths from node A in the following network:



3. Using the Floyd-Warshall algorithm, compute the shortest paths between each pair of nodes in the following network:

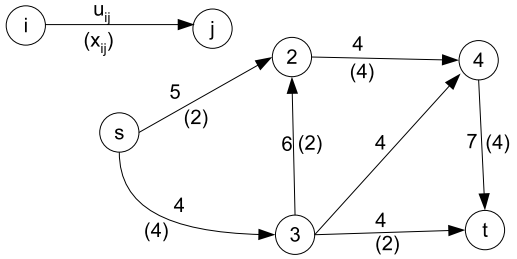


4. For the project shown in the following table, construct the corresponding network, compute the duration time of the project, compute the floats of the activities, find the critical activities and draw the Gantt chart.

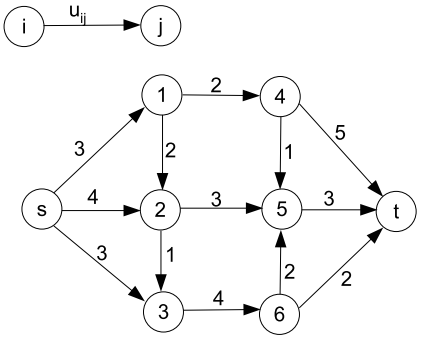
Activity	Direct predecessors	Duration times
A	-	1
B	-	4
C	-	2
D	**B**,**C**	3
E	**A**	2
F	**E**,**D**	3
G	**C**	2

5. Show that the shortest path problem and the maximum flow problem are special cases of the minimum cost flow problem.

6. Consider the following network with a given flow from s to t. Check whether this flow is maximal. If not, find the maximum flow and a minimum cut in this network.



7. Compute the maximum flow and a minimum cut in the following network:

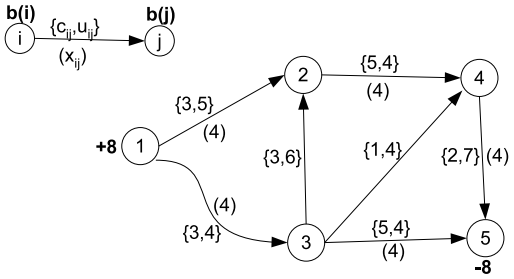


8. Consider the following instance of the minimum cost flow problem, specified by a network with a given feasible flow.

   (a) Show that this flow is not optimal.

(b) Starting with this flow, compute an optimal flow using the cycle canceling algorithm and the network simplex algorithm.

(c) Perform the sensitivity analysis for the optimal solution obtained.

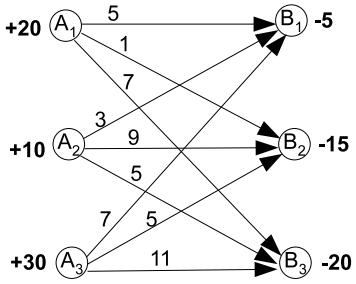


9. Consider the following transportation problem.

(a) Construct the transportation table for this problem and compute the initial solution using the north-west corner or the minimum element cost method.

(b) Starting with the initial solution computed in the previous point, find an optimal solution by using the network simplex algorithm.

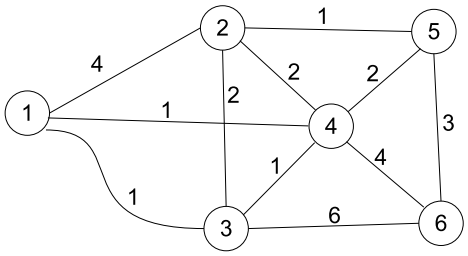
(c) Perform the sensitivity analysis for the optimal solution obtained.



10. Solve the following minimum cost assignment problem.

5	1	2
4	3	7
2	1	6

11. Using Prim's and Kruskal's algorithms, compute a minimum spanning tree in the following network. Having computed a minimum spanning tree, can you find another minimum spanning tree?



Chapter 3

Solving hard problems

In the previous chapter we discussed the class of network flow problems which can be solved by efficient polynomial time algorithms. Unfortunately, there are a lot of important problems, for which no such efficient algorithm is known. In this chapter we present some general techniques for dealing with such problems.

3.1 Mathematical programming formulation

Perhaps the easiest and the most general method of solving a discrete optimization problem is designing a mathematical programming model for it and applying one of many available packages to solve it. Most discrete optimization problems can be formulated as the following *linear integer program*:

$$\begin{aligned} &\min \sum_{j=1}^{n} c_j x_j \\ &\sum_{j=1}^{n} c_{ij} x_j \leq (=) b_i \quad i = 1, \ldots, m \\ &x_j \in D_j \qquad\qquad\quad j = 1, \ldots, n \end{aligned} \tag{3.1}$$

We can distinguish four elements in (3.1). The first is a set of *decision variables* $x_1, \ldots, x_n$. These are unknown quantities whose values are to be computed. Each variable x_j has some domain D_j and in a discrete optimization problem we assume that this domain is a set of integers. One very important, special case is when $x_j \in \{0, 1\}$, i.e. variable x_j may take only two values 0 or 1. We call such variable a *binary variable*. Binary variables are very useful if we wish to describe certain finite objects, such as subsets, permutations, paths, trees or cycles. The second element of any mathematical model is a set of *parameters*. The parameters are the constants c_j, b_i and a_{ij}, which describe an instance of an optimization problem. Typical parameters are costs, traveling times, capacities, demands etc. The parameters may also describe the structure of the instance, for example, a set of connections in a given network. Having

decision variables and parameters we can write an *objective function*, i.e. a linear expression which has to be minimized or maximized. This expression should represent the cost (or value) of a solution, where a solution is described by the decision variables. Finally, we also provide a set of linear expressions called *constraints*, which describe all the requirements to be satisfied.

In theory, there is no limit on the type of the objective function or constraints. They may take the form of any mathematical expression, which can be evaluated. However, in practice we are able to solve model, in which all the expressions are *linear* and the model can be formulated using a *linear program*. In many cases, nonlinear expressions can be linearized by using additional variables (see the literature, e.g. [50]). There are some powerful computer packages for solving linear programming problems such as CPLEX [27] or GLPK [28], which can be used to solve the models constructed. Furthermore, these packages contain modeling languages, which allow us to express the models and input data in a natural way. We will now consider several examples. We will build models for several problems for which no polynomial algorithms are known. We will use the MathProg modeling language which is the part of the freely available GNU GKLPK software [28].

Example 1. (vertex cover) We are given an undirected network $G = (N, A)$. The subset of nodes $W \subseteq N$ *covers* the set of arcs, if for all $(i, j) \in A$ either $i \in W$ or $j \in W$. We wish to find the smallest subset of nodes which covers all the arcs of G. This problem arises in many applications. For example, you can imagine N as a set of rooms in a museum and $(i, j) \in A$ if a door exists between rooms i and j. The problem is where to place cameras so that all the doors are observed. We can solve this problem in the following way. Define a binary variable $x_i \in \{0, 1\}$. Then $x_i = 1$ if and only if a camera is placed in room $i \in N$. The number of cameras installed is thus $\sum_{i \in N} x_i$. Since every door must be observed, the condition $x_i + x_j \geq 1$ must be satisfied for all $(i, j) \in A$. The model in the MathProg language is the following:

```
set N;
set A within N cross N;
var x{N} binary;
maximize cost:  sum{i in N} x[i];
cover{(i,j) in A}:  x[i]+x[j]>=1;
solve;
display x;
data;
#-----provide data
end;
```

Example 2. (knapsack problem) Consider the knapsack problem (Example 2 in Section 1.1). Le us introduce a binary variable $x_i \in \{0, 1\}$ for each item $i = 1, \ldots, n$. So $x_i = 1$ if and only if item i is taken. The value of a given solution

can be expressed as $\sum_{i=1}^{n} p_i x_i$ and this expression should be maximized. There is one constraint in the problem, which ensures that the capacity W cannot be exceeded. So this constraint takes the form $\sum_{i=1}^{n} w_i x_i \leq W$. The model is of the following form:

$$\begin{array}{ll} \max \sum_{i=1}^{n} p_i x_i & \\ \sum_{i=1}^{n} w_i x_i \leq W & \\ x_i \in \{0,1\} & i = 1, \ldots, n \end{array}$$

This basic problem can be extended. Suppose that, for some reasons, there are pairs of packs which cannot be taken together and these pairs form set U. So, if $(i,j) \in U$, then i cannot be taken with j. We can express this requirement by adding constraints $x_i + x_j \leq 1$ for all $(i,j) \in U$. Indeed, these constraints assure that both x_i and x_j never simultaneously take values equal to 1. In the MathProg language, the model is the following:

```
param n;
param p{1..n};
param w{1..n};
param W;
set U within n cross n;
var x{1..n} binary;
profit:  sum{i in 1..n} p[i]*x[i];
constr1:  sum{i in 1..n} w[i]*x[i] <=W;
constr2{(i,j) in U}:  x[i]+x[j]<=1;
solve;
display x;
data;
#-----provide data
end;
```

Example 3. (constrained shortest path) We are given a directed network $G = (N, A)$ with two distinguished nodes s and t. We know the length $d_{ij} > 0$ and travel time $p_{ij} > 0$ of each arc $(i,j) \in A$. We would like to find the shortest directed path from s to t in G with the additional restriction that the total travel time of this path cannot exceed T. This is a version of the shortest path problem with one additional constraint. Let us introduce a binary variable $x_{ij} \in \{0,1\}$ for each arc $(i,j) \in A$. Thus $x_{ij} = 1$ if and only if the path uses arc (i,j). The objective function simply expresses the length of the path and has the form $\sum_{(i,j)\in A} d_{ij} x_{ij}$. The additional constraint can be expressed as $\sum_{(i,j)\in A} p_{ij} x_{ij} \leq T$. The most complex task is to ensure that the solution obtained indeed describes a directed path from s to t in G. Notice first that each path uses exactly one arc leaving s. So we have the constraint, $\sum_{\{j:(s,j)\in A\}} x_{sj} = 1$. Similarly, every path uses exactly one arc to t. Hence, we write $\sum_{\{i:(i,t)\in A\}} x_{it} = 1$. Finally, for every intermediate node $v \in N$, other than s and t, if the path enters v, then it must also leave v. We can express this condition as $\sum_{\{j:(v,j)\in A\}} x_{vj} - \sum_{\{i:(i,v)\in A\}} x_{iv} = 0$. In fact, this condition

expresses something more. It says that the path enters v as many times as it leaves v. In consequence, some nodes may be visited more than once and the solution may contain directed cycles. However, if all the lengths and times are positive, then no such cycle can appear in any optimal solution and the solution obtained indeed describes a directed path from s to t. The model in MathProg takes the following form:

```
set N;
set A within N cross N;
param s in N;
param t in N;
param d{A}>0;
param p{A}>0;
param T;
var x{A} binary;
maximize dist:  sum{(i,j) in A} d[i,j]*x[i,j];
time:  sum {(i,j)in A} p[i,j]*x[i,j]<=T;
c1:  sum{(s,j) in A} x[s,j]=1;
c2:  sum{(i,t) in A} x[i,t]=1;
c3 {v in N diff{s,t}}:  sum{(v,j)in A} x[v,j] - sum{(i,v)in A}
x[i,v]=0;
solve;
display x;
data;
#-----provide data
end;
```

Example 4. (traveling salesperson). Consider the traveling salesperson problem in a directed network $G = (N, A)$, $|N| = n$, with arc costs c_{ij} for each $(i, j) \in A$. Recall that we seek a cheapest directed Hamiltonian cycle in G, i.e. a cheapest directed cycle in G that visits every node of G exactly once. Let us introduce a binary variable $x_{ij} \in \{0, 1\}$ for each arc $(i, j) \in A$. Thus $x_{ij} = 1$ if and only if we move from node i to node j in the tour represented by a solution. The cost of such a tour can be expressed as $\sum_{(i,j)\in A} c_{ij}x_{ij}$. Observe that the tour enters every node exactly once and leaves every node exactly once. We can express this by adding the constraints $\sum_{\{j:(i,j)\in A\}} x_{ij} = 1$ for all $i \in N$ and $\sum_{\{j:(j,i)\in A\}} x_{ji} = 1$ for all $i \in N$. These constraints, however, are not sufficient because we may obtain, as a result, a solution composed of several disjoint cycles. This undesired situation can be prohibited by using the following idea. Suppose that we have $b(1) = n-1$ units of some product at node 1. Each node, other than 1, requires 1 unit of the product, so $b(i) = -1$ for all $i \in N \setminus \{1\}$. We have to deliver the product from node 1 to the remaining nodes and this is a classical network flow problem. Let $y_{ij} \geq 0$ denote the flow along arc $(i, j) \in A$. So the constraints $\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b(i)$ must be satisfied for all $i \in N$. What are the relationships between the variables x_{ij} and y_{ij}? We

can send the product only along arcs such that $x_{ij} = 1$. We can ensure this by adding the constraints $y_{ij} \leq (n-1)x_{ij}$ for each $(i,j) \in A$. The resulting model describes the traveling salesperson problem. Indeed, the variables x_{ij}, $(i,j) \in A$, cannot now describe several disjoint cycles, because in this case the constraints of the network flow problem would not be satisfied. The model written in the MathProg language has the following form:

```
param n;
set A within 1..n cross 1..n;
param c{A};
param b{i in 1..n}:=if i==1 then n-1 else -1;
var x{A} binary;
var y{A} >=0;
minimize cost:  sum{(i,j) in A} c[i,j]*x[i,j];
c1{i in 1..n}:  sum{(i,j) in A} x[i,j]=1;
c2{i in 1..n}:  sum{(j,i) in A} x[j,i]=1;
c3{i in 1..n}:  sum{(i,j) in A} y[i,j]-sum{(j,i) in A}
y[j,i]=b[i];
c4{(i,j) in A}:  y[i,j]<=(n-1)*x[i,j];
solve;
display x;
data;
#-----provide data
end;
```

After writing the models, we can provide some input data and solve them using an available routine. In the case of the MathProg language, we can use the *glpsol* routine [28].

3.2 Branch and bound algorithm

In principle, every discrete optimization problem can be solved by a version of the brute force algorithm. Recall that this algorithm simply evaluates all the feasible solutions and returns the best one. As we have seen in Section 1.2, this approach is very inefficient, because the running time of the brute force becomes very long even for small instances of an optimization problem. In this section we describe a more clever method of evaluating the solutions called the *branch and bound*. In extreme cases, the branch and bound method may behave like the brute force but, in most cases, it allows us to solve much larger instances. In this section we will assume that the problem is a minimization one. It is easy to adapt the reasoning to the maximization problems and we leave this as an exercise.

Let $S_0 = sol(I)$ be the set of all solutions for a given instance I of some optimization problem. As we know, computing an optimal solution in S_0 may be very complex. It may be, however, much easier to estimate the cost of an optimal solution from below. Namely, we could compute a function $B(S_0)$ called

a *lower bound*, which tells us that the cost of an optimal solution in S_0 cannot be less than $B(S_0)$. It is very important that the value of $B(S_0)$ can be efficiently computed. In most cases, we can also efficiently compute a set of solutions (not necessarily optimal) and assume that x_{best} is the best solution among them. We can simply generate a set of random solutions and choose the best of them or use some fast heuristic to obtain x_{best}.

At this point, we know that the cost of an optimal solution belongs to the interval $[B(S_0), f(x_{best})]$. Hence, we have a solution x_{best} whose percentage deviation from the optimum is at most $(f(x_{best}) - B(S_0))/B(S_0)$ provided that $B(S_0) > 0$. If this error is small enough, then we may be satisfied and we accept the solution x_{best}. If not, then we can proceed by performing a *branch*. A branch consists of dividing the solution set S_0 into two or more subsets $S_1, S_2, \ldots, S_k$. We then compute the lower bounds $B(S_1), B(S_2), \ldots, B(S_k)$ for all of these subsets. If $B(S_i) \geq f(x_{best})$ then S_i cannot contain a solution better than x_{best}. In this case, we may reject S_i and do not consider solutions from this set any more. On the other hand, if $B(S_i) < f(x_{best})$, then S_i may contain a better solution and this set must be further explored. We do this by branching S_i and computing the lower bounds for the subsets obtained. The algorithm stops when all the subsets are rejected, i.e. there is nothing to branch. Notice that during the execution of the branch and bound algorithm the lower bounds cannot decrease and the cost of x_{best} cannot increase. So, the percentage deviation of x_{best} from the optimum will decrease and we can terminate the algorithm if we accept the deviation.

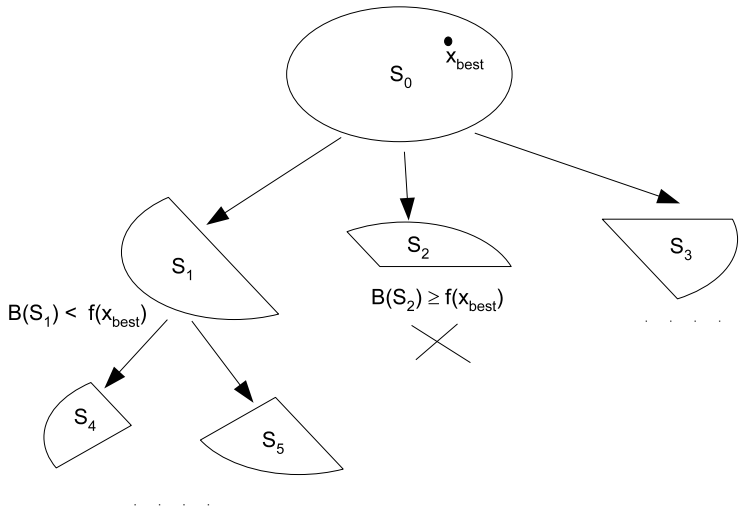


Figure 3.1: Branch and bound algorithm.

There are several parts of the algorithm whose details depend on the problem. In particular, we must establish a method of computing the lower bounds and a method of branching. The execution of the branch and bound algorithm

can be seen as a tree, whose nodes represent parts of the solution space. In Figure 3.1 the current tree contains 6 nodes. The nodes S_0 and S_2 are rejected and the nodes S_3, S_4 and S_5 need to be explored. There are several ways of choosing the next node to be explored. One of the most popular method consists of choosing the node with the smallest lower bound $B(S_i)$. In general, the running time of the branch and bound algorithm can be exponential. The algorithm may also require a large memory to store the information about the search tree. Nevertheless, the idea of branch and bound is regarded as the most powerful technique for solving hard discrete optimization problems.

Example 1. We now design a branch and bound algorithm for the traveling salesperson problem. In order to illustrate the algorithm, we use a small problem with 5 cities, numbered from 1 to 5 and the cost matrix shown in Table 3.1. Observe that the cost of traveling from city i to i is set to ∞. In this way we prohibit traveling from any city to itself.

	1	**2**	**3**	**4**	**5**
1	∞	132	217	164	58
2	132	∞	290	201	79
3	217	290	∞	113	303
4	164	201	113	∞	196
5	58	79	303	196	∞

Table 3.1: Cost matrix for the sample problem.

A good idea for obtaining the first tour is to use the fast and intuitive nearest neighbor method. We start from city 1 and check which city is the nearest one. This is city 5, so we travel from 1 to 5. We are now in city 5 and we again check which city is the nearest one. This is city 1. However, we cannot return at this point to 1, so the second nearest city is 2 and we travel to 2. Proceeding in this way, we finally get the tour 1-5-2-4-3-1 with a total cost of 855. This tour is our x_{best}. We now focus on computing a lower bound. Let us look at Figure 3.2.

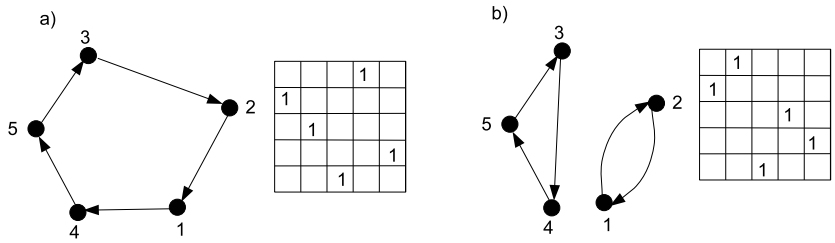


Figure 3.2: (a) Every tour is an assignment, but (b) not every assignment is a tour.

We can treat each tour as an assignment. For example, the tour shown in

Figure 3.2a assigns 3 to 2, 2 to 1, 1 to 4, 4 to 5 and 5 to 3. We can represent this assignment as a matrix with 1's placed in box (i, j) if i is assigned to j. This matrix has exactly one 1 in each column and exactly one 1 in each row. However, not every assignment represents a tour. An example is shown in Figure 3.2b, where the assignment represents two separate cycles.

Now the crucial fact is that, contrary to the optimal tour, the assignment of minimum cost can be computed efficiently (see Section 2.5). Since the set of all assignments contains all the tours, the cost of the optimal tour cannot be less than the cost of the optimal assignment. Consequently, the cost of the optimal assignment is a lower bound on the cost of the optimal tour. The optimal assignment is shown in Figure 3.3. Its cost equals 625, so $B(S_0) = 625$.

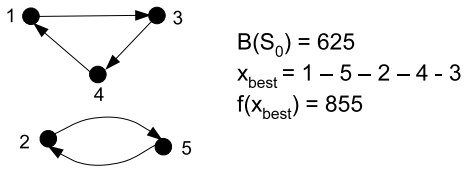


Figure 3.3: The optimal assignment and the tour x_{best}

At this point, we have the following information. We know that the cost of the optimal tour belongs to the interval $[625, 855]$. Furthermore, we know that the tour x_{best} has total cost 855. So the cost of x_{best} is at most 31.1% greater than the optimum. If we are satisfied with this error, then we can terminate the algorithm and return the tour x_{best}. But we are not, so we have to perform a branch (see Figure 3.4).

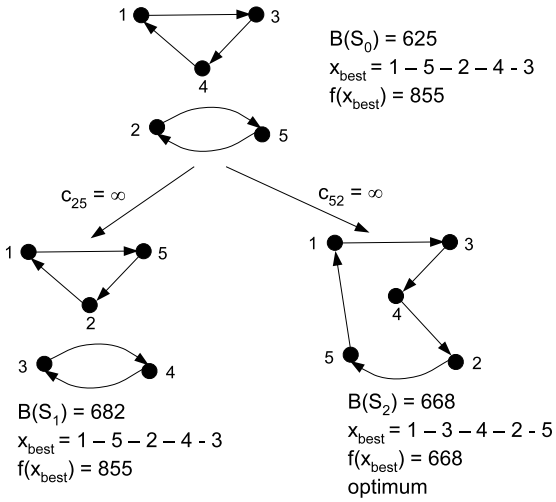


Figure 3.4: The results of branching.

Clearly, no tour can contain the cycle 2-5-2. So we can divide the solution set S_0 into two sets. In S_1 we prohibit the arc $2-5$ by setting the traveling cost from 2 to 5 to be ∞, while in S_2 we prohibit the arc $5-2$ by setting the traveling cost from 5 to 2 to be ∞. We then compute the optimal assignments for S_1 and S_2 and the results obtained are shown in Figure 3.4. The optimal assignment for S_1 has a total cost equal to 682 and it is not a tour. However, we know that the optimal tour in S_1 has a cost of not less than 682. The optimal assignment for S_2 is a tour and has a total cost of 668. At this point, we can update x_{best} to 1-3-4-2-5. This tour is optimal because no better tour can exist in S_1 or S_2. This follows from the inequalities $B(S_1) \geq f(x_{best})$ and $B(S_2) \geq f(x_{best})$.

Example 2. The branch and bound algorithm can be used to solve the general linear integer programming problem problem (3.1) (see Section 3.1). The key observation is that after removing the integrality constraints, we get a linear programming problem which can be solved efficiently. Furthermore, the cost of the optimal solution to the linear program is a lower bound on the optimal cost in the original problem. Consider the following example:

$$
\begin{array}{ll}
z^* = \min -8x_1 - 5x_2 & z_R^* = \min -8x_1 - 5x_2 \\
6x_1 + 10x_2 \leq 45 & 6x_1 + 10x_2 \leq 45 \\
9x_1 + 5x_2 \leq 45 & 9x_1 + 5x_2 \leq 45 \\
x_1, x_2 \geq 0,\ x_1, x_2 \text{ integer} & x_1, x_2 \geq 0,
\end{array}
$$

We get the value of z_R^* by removing the integrality constraints for the variables x_1 and x_2. The resulting problem can be solved efficiently. In this case, we can use the graphical method (see Figure 3.5). When there are more variables, the simplex algorithm can be used.

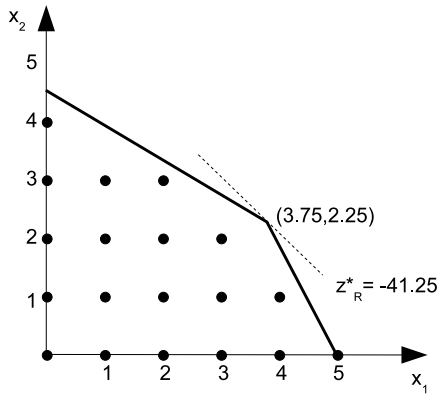


Figure 3.5: Integer feasible solutions (black circles) and the optimal solution after removing the integrality constraints.

After removing the integrality constraints, we get the optimal solution $x_1 = 3.75$ and $x_2 = 2.25$ with the value of the objective function $z_R^* = -41.25$.

Therefore, we know that -41.25 is a lower bound on the value of the objective function in the original problem. However, the solution obtained is not integer and we have to perform a branch. Let us choose variable x_1. Since $x_1 = 3.75$ and it cannot take fractional values, either $x_1 \leq 3$ or $x_1 \geq 4$. Considering these two cases we get two nodes of the branch and bound tree labeled as S_1 and S_2 (see Figure 3.6). In S_1 we add the constraint $x_1 \leq 3$ and in S_2 we add the constraint $x_1 \geq 4$. We solve both problems and again we get fractional solutions. Hence, we branch S_2 by considering two cases $x_2 \leq 1$ and $x_2 \geq 3$. The second case leads to an infeasible problem. The first case (S_3) again leads to a fractional solution, where $x_2 = 4.44$. So we perform another branch by considering two cases $x_2 \leq 4$ and $x_2 \geq 5$. As a result, we get the tree shown in Figure 3.6.

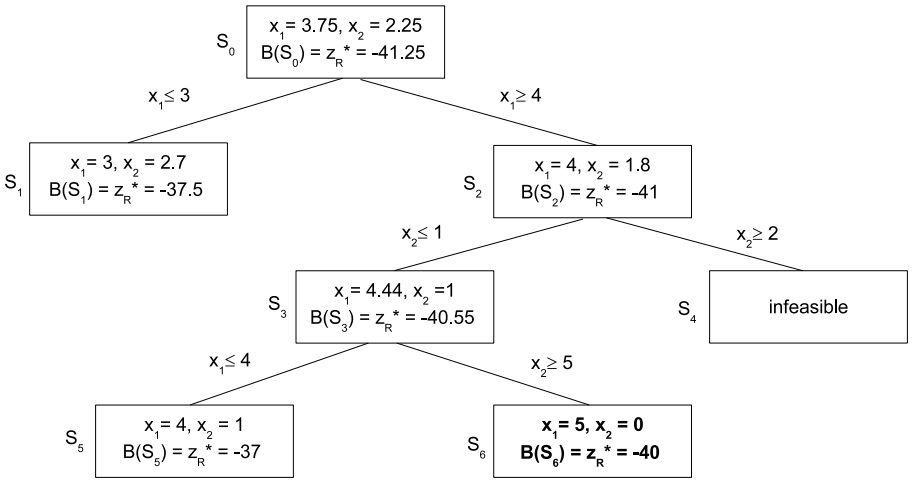


Figure 3.6: The results of branching.

We can terminate the algorithm at this point. At nodes S_5 and S_6 we have obtained integer solutions. The better solution is $x_1 = 5$ and $x_2 = 0$ with the value of the objective function -40. At node S_1 there is still a fractional solution. However, the lower bound at this node is $B(S_1) = -37.5$, which is greater than -40. So no better solution can be obtained by branching S_1. The optimal solution is thus $x_1 = 5$ and $x_2 = 0$.

3.3 Dynamic programming

Dynamic programming is the second general technique for solving discrete optimization problems. It can be applied to problems possessing a special structure. We have described such an algorithm in Section 2.1, where we discussed the shortest path problem in acyclic networks. We now consider the following example:

Example 1. A company wants to invest \$5 000 000 in 3 factories. There are several possibilities for investment in each factory and they are listed in Table 3.2. For example, the company can invest \$2 000 000 in Factory 1, \$3 000 000 in Factory 2 and \$0 in Factory 3, which gives a total profit of \$15 000 000. Which variants should be chosen in each factory to maximize the total profit?

	Factory 1		Factory 2		Factory 3	
Variant i	$c_1(i)$	$p_1(i)$	$c_2(i)$	$p_2(i)$	$c_3(i)$	$p_3(i)$
1	0	0	0	0	0	0
2	1	5	2	8	1	4
3	2	6	3	9	-	-
4	-	-	5	12	-	-

Table 3.2: The costs $c_j(i)$ and the profits $p_j(i)$ in million \$ for investment i in factory j

A naive approach to solving this problem would be to try all the possibilities, i.e. all the possible combinations of the investment variants. This is of course a kind of brute force method, which would work quite well for our small example, but fail for lager instances. If there were n factories with m investment variants, then we would have to check m^n possibilities. This is too many even for small problems, say with $n = 10$ and $m = 10$. This naive approach also performs a lot of unnecessary work by examining infeasible investment variants, i.e. those which exceed the given budget. We now show a much more efficient method of solving this problem, which is based on the idea of dynamic programming.

We divide the problem into three subproblems. In the first subproblem we consider only Factory 1. Let $f_1(x)$ be the maximum profit in Factory 1 under the assumption that we have x dollars. Hence, the value of $f_1(x)$ expresses the best variant in Factory 1 under the assumption that we have x dollars to spend. This quantity can be computed in the following way:

$$f_1(x) = \max_{\{i:c_1(i)\leq x\}} \{p_1(i)\}.$$

In the second subproblem we consider Factory 1 and Factory 2. Now $f_2(x)$ is the maximum profit from both factories assuming that we can spend x dollars. If we have x dollars, then we can choose only those variants i in Factory 2, for which $c_2(i) \leq x$. If we choose the ith variant, then our profit will be $p_2(i)+f_1(x-c_2(i))$. The first term $p_2(i)$ results from choosing the ith variant in Factory 2 and the second term $f_1(x - c_2(i))$ expresses the value of the best variant in Factory 1, when we have $x - c_2(i)$ dollars remaining. Hence, $f_2(x)$ can be computed as follows:

$$f_2(x) = \max_{\{i:c_2(i)\leq x\}} \{p_2(i) + f_1(x - c_2(i))\}.$$

Finally, we consider the third subproblem with all three factories. The function $f_3(x)$ expresses the maximum profit from all three factories under the assumption that we can spend x dollars. We can repeat the same argument as used

previously and compute $f_3(x)$ in the following way:

$$f_3(x) = \max_{\{i:c_3(i)\leq x\}}\{p_3(i) + f_2(x - c_3(i))\}.$$

We have now some recursive relationships involving $f_i(x)$ for $i = 1, 2, 3$. We get an optimal solution by computing $f_3(5)$, i.e. the value of an optimal investment for all three factories assuming that we have \$5 000 000. The computation of $f_3(5)$ is shown in Figure 3.7.

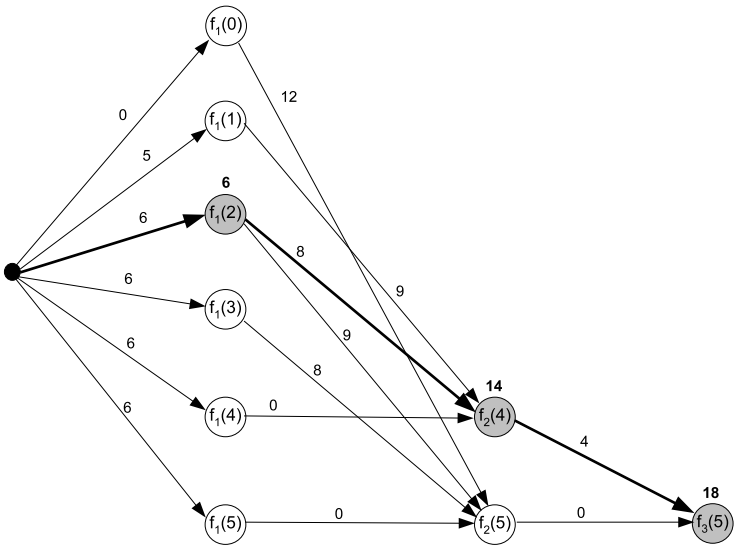


Figure 3.7: Solving the problem by dynamic programming.

We first construct a network, whose nodes represent all the possible states $f_i(x)$ in the problem. Thus we start with $f_3(5)$ and check that this state can be reached from $f_2(5)$ (we can choose variant 1 in Factory 3) and $f_2(4)$ (we can choose variant 2 in Factory 3). We represent these transitions by arrows $(f_2(5), f_3(5))$ and $(f_2(4), f_3(5))$ with profits 0 and 4, respectively. We consider nodes $f_2(5)$ and $f_2(4)$ in a similar way. As a result, we get a directed and acyclic network. The optimal solution can be obtained by computing a longest path in this network. This path is shown in Figure 3.7. Its value is equal to 18 and we should choose variant 2 in Factory 3, variant 2 in Factory 2 and variant 3 in Factory 1.

Example 2 (knapsack problem) Consider the knapsack problem, which was defined in Section 1.1 (Example 2). In principle, we could evaluate all the solutions to this problem and choose the best one. We would have to, however, evaluate up to 2^n solutions, which is prohibitive even for small n. We now show a much faster method of solving this problem. We divide the problem into $n + 1$ subproblems numbered from 0 to n. In the ith subproblem, we compute the

optimal solution for the subset of items $\{1, \ldots, i\}$. If $i = 0$, then the set of items is empty. Let $f_i(x)$ be the value of the optimal solution to the ith subproblem, under the assumption that we can take items whose total weight does not exceed x. Of course $f_0(x) = 0$ for all x since there are no items to take. How can we compute $f_i(x)$ for $i > 0$? In the ith subproblem we have to decide whether to take item i. We can do this only if $x \geq w_i$. So, if $x < w_i$, then $f_i(x) = f_{i-1}(x)$. If $x \geq w_i$, then we have two possibilities: we do not take i and $f_i(x) = f_{i-1}(x)$, or we take i and $f_i(x) = f_{i-1}(x - w_i) + p_i$. Since we wish to maximize the value of the items taken, we should choose the possibility which gives us a greater value of $f_i(x)$. Summarizing, we get the following recursive formula:

$$f_0(x) = 0$$
$$f_i(x) = \begin{cases} f_{i-1}(x) & x < w_i \\ \max\{f_{i-1}(x), f_{i-1}(x - w_i) + p_i\} & x \geq w_i \end{cases} \quad i = 1, \ldots, n$$

We obtain an optimal solution by computing $f_n(W)$. The computations for a sample problem with 4 items are shown in Figure 3.8. The nodes of the network constructed represent all the possible states $f_i(x)$ and the arcs represent transitions between states. We compute the longest path in this network, which corresponds to the optimal solution $\{2, 3, 4\}$.

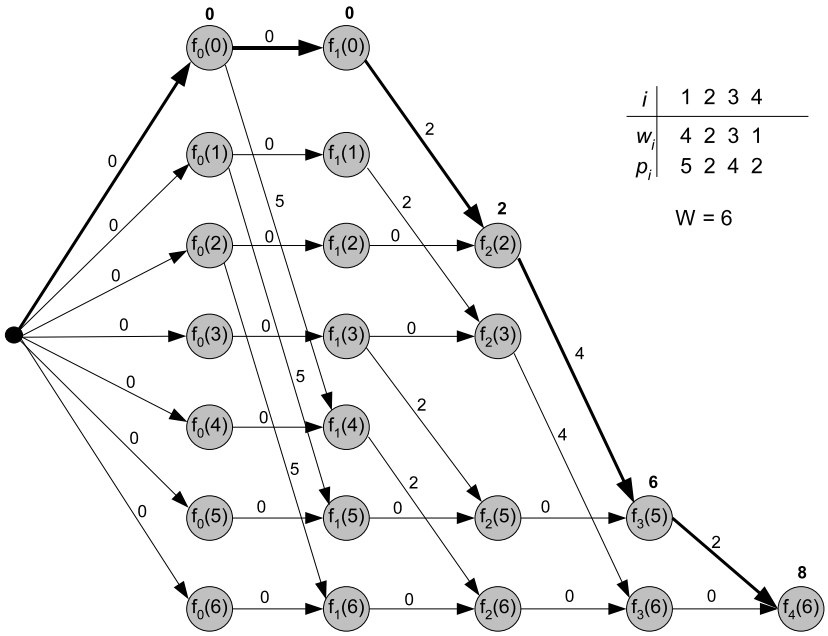


Figure 3.8: Solving the knapsack problem by dynamic programming.

We now estimate the running time of the dynamic algorithm. The network contains $O(nW)$ nodes. The special structure of this network allows us to com-

pute the longest path in $O(nW)$ time. So we can get an optimal solution to the knapsack problem in $O(nW)$ time. Notice that this is much better than the running time of the brute force algorithm, which is $O(n2^n)$. If W is not very large, then we can efficiently solve problems with thousands of items.

After analyzing these two examples, we can informally describe the general idea of dynamic programming. We first split the problem into a sequence of n subproblems, where the last, nth, subproblem represents the whole problem. We then define a function $f_i(x)$, with the argument x representing the state in subproblem i. In the first example x is the amount of money possessed and in the second example x is the available capacity. Finally, there are some recursive relationships, which allow us to compute $f_i(x)$ having computed $f_1(x), \ldots, f_{i-1}(x)$. We obtain an optimal solution by computing $f_n(x_0)$, where x_0 is the state in the original problem. The computations can be represented as a directed network, whose nodes are all the possible values of $f_i(x)$. We first build this network by working backwards using the recursive relationships. We then compute the longest (or the shortest if we are minimizing the objective function) path in the network obtained, which represents an optimal solution to the original problem. The algorithms based on this approach are much faster than the brute force algorithm.

3.4 Approximation algorithms and heuristics

If an instance of a hard optimization problem is large, then it may not be possible to compute an optimal solution in reasonable time. For example, computing a shortest traveling salesperson tour for thousands of cities may be a hopeless task. In this case, we might be satisfied with a solution which is *close* to optimal, if such a solution can be computed quickly.

Example 1 (vertex cover) In Section 3.1 (Example 3) we described the vertex cover problem. This problem is known to be computationally hard. Consider the algorithm shown in Figure 3.9.

```
1: W := ∅
2: while G contains some arc do
3:     Choose any arc (i, j) of G
4:     W := W ∪ {i, j}
5:     Remove all the arcs incident to i and j from G.
6: end while
7: return W
```

Figure 3.9: An approximation algorithm for the vertex cover problem.

The algorithm chooses any arc (i, j) of G, adds nodes i and j to the cover constructed and removes from G all the arcs incident to i and j. The algorithm

stops when G has no arcs. It is easy to observe that W is a cover in G, i.e. all the arcs of G are covered by some node in W. Furthermore, the algorithm is very fast and can easily be implemented to run in $O(m)$ time. We can thus suspect that W need not be an optimal solution, since otherwise the vertex cover problem would be easy to solve. Let W^* be a minimum cover. Observe that if an arc (i, j) is chosen in line 3, then either i or j must belong to W^* (otherwise the arc (i, j) would not be covered). This implies that $|W^*| \geq |W|/2$ or, equivalently, $|W| \leq 2|W^*|$. This inequality means that the size of the cover constructed is at most two times larger than the size of the minimum cover.

Let us now introduce the following definition. An algorithm A is said to be a *k-approximation algorithm* for a minimization problem Π if:

1. A runs in polynomial time.
2. For every instance $I \in D_\Pi$, the algorithm A returns a solution $x \in sol(I)$ such that $f(x) \leq kf(x^*)$. The constant k is called the *worst case ratio* for A.

The corresponding definition for a maximization problem is very similar. We must only replace $f(x) \leq kf(x^*)$ with $f(x) \geq \frac{1}{k}f(x^*)$ in condition 2. Condition 1 means that algorithm A is fast. Condition 2 means that, for every instance of the problem, A returns a solution whose cost is at most k times greater (smaller) than the optimum. Observe that a 1-approximation algorithm is an exact algorithm for Π, so if Π is computationally hard, then we should expect that $k > 1$. Of course, the value of k should be as small as possible.

Example 2 (metric traveling salesperson) Consider the traveling salesperson problem defined in Section 1.1 (Example 1). We discuss here a special *metric* version of this problem, where the arc costs satisfy the so called *triangle inequality*, that is for every three nodes i, j, k the inequality $c_{ik} \leq c_{ij} + c_{jk}$ must hold. We also assume that the problem is symmetric, i.e. $c_{ij} = c_{ji}$ for all nodes i, j. A special case is *Euclidean* problem, where the nodes are located in a plane and $c_{ij} = c_{ji}$ is the Euclidean distance between i and j. Consider the problem with 6 nodes shown in Figure 3.10.

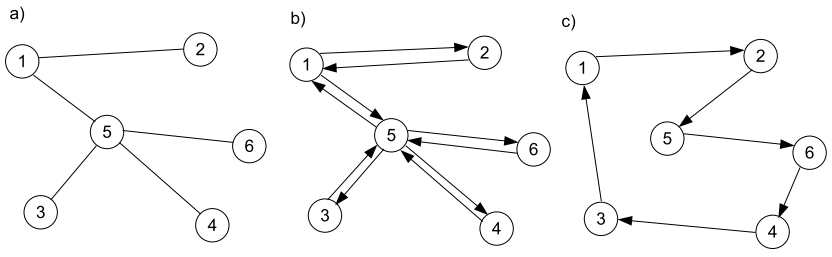


Figure 3.10: An approximation algorithm for the metric traveling salesperson problem.

In the first step, we compute a minimum spanning tree T for G, as shown in Figure 3.10a. Let us denote the optimal tour by π^*. It follows that $f(T) \leq f(\pi^*)$, so the total cost of T is not greater than the total cost of an optimal tour. The proof of this observation results easily by contradiction. Suppose that $f(T) > f(\pi^*)$. If we remove any arc from π^*, then we get a spanning tree whose cost is less than T, which contradicts the fact that T is a minimum spanning tree. In the next step, we transform T by replacing every undirected arc of T by two directed arcs as shown in Figure 3.10b. Note that the degree of every node in the network obtained is even, so this network has an Eulerian cycle, i.e. a walk that visits every arc exactly once. The Eulerian cycle of the network from Figure 3.10b is $L = 1-2-1-5-6-5-4-5-3-5-1$ and, clearly, $f(L) = 2f(T) \leq 2f(\pi^*)$. We can now transform L into a traveling salesperson's tour by removing the repeated nodes from L. Hence we get the tour $\pi = 1-2-5-6-4-3-1$ shown in Figure 3.10c. Removing a node from L cannot increase the cost of the walk obtained, which follows from the triangle inequality. For example, replacing $2-1-5$ with $2-5$ does not increase the cost because $c_{25} \leq c_{21}+c_{15}$. Hence, we get $f(\pi) \leq f(L) \leq 2f(\pi^*)$ and the algorithm presented is a 2-approximation for the metric traveling salesperson problem.

The algorithm can be additionally improved and the idea is shown in Figure 3.11.

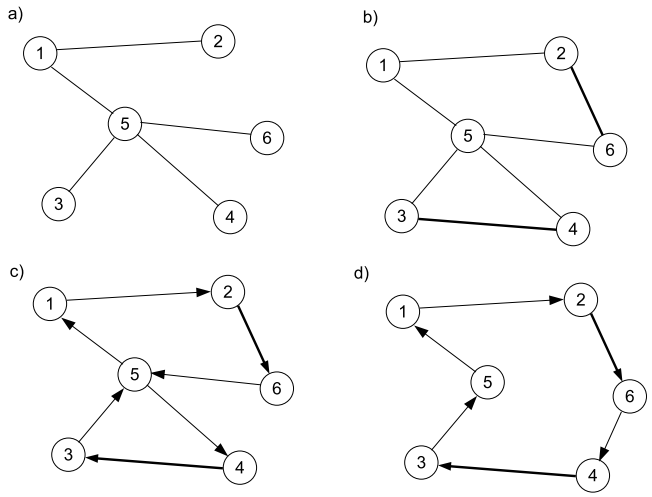


Figure 3.11: Christofides algorithm for the metric traveling salesperson.

The first step is the same as previously. We construct a minimum spanning tree T for the set of nodes. The next step, however, is different. We consider the nodes of the spanning tree T obtained which have an odd degree (nodes 2,3,4 and 6). The number of such nodes must be even, so we construct a matching M with the minimum cost for them (see Figure 3.11b). The minimum cost matching can be computed in polynomial time. In this example, the minimum cost matching

is $(2, 6)$ and $(3, 4)$. We add this matching to T obtaining a network, where all the nodes have an even degree. The last steps are the same as in the previous algorithm. We compute an Eulerian cycle L and remove all the repeated nodes from L obtaining a tour π. As previously, we have $f(T) \leq f(\pi^*)$. It also follows that $f(M) \leq f(\pi^*)/2$. In consequence, $f(\pi) \leq f(T) + f(M) \leq 3/2 f(\pi^*)$. The algorithm, called *Christofides algorithm*, has an approximation ratio equal to $3/2$.

For some optimization problems it is believed that no k-approximation algorithms exist (see Appendix C for a deeper discussion on this topic). A polynomial algorithm, which returns a solution to an optimization problem is called a *heuristic*. The solution returned by a heuristic may be arbitrarily far from the optimum in the worst case. We can expect, however, that this solution will be of good quality for most typical instances. Consider the following examples:

Example 3 (graph coloring). We are given an undirected network $G = (N, A)$. A *coloring* is an assignment of a color to each node $i \in N$, so that every two nodes i, j linked by arc $(i, j) \in A$ have distinct colors. We wish to find a coloring that uses a minimum number of colors. The graph coloring is a very famous problem having a long history. It is also one of the hardest problems in discrete optimization. In order to find a reasonable coloring, the following simple method can be used. Assume that the colors are numbered $1, 2, \ldots$. We order the nodes of G with respect to nonincreasing degrees (the number of incident arcs) and successively assign the smallest possible color to each node in this order. This algorithm works well for many instances. However, it is not a k-approximation algorithm for any finite k and this fact is illustrated in Figure 3.12. All the nodes in the sample network have the same degree equal to n. So the algorithm may assign colors to the nodes in any order. If this order is 1,2,... $2n$, then it uses n colors. But it is easy to see that two colors are enough to color this network. So the worst case ratio for this algorithm is $n/2$, which can be arbitrarily large.

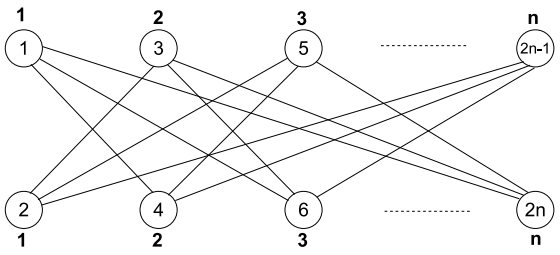


Figure 3.12: A sample graph coloring problem.

Example 4 (general traveling salesperson). Again, consider the traveling salesperson problem, but now we do not impose any restrictions on the arc costs. We will assume, however, that the network is complete, i.e. there is an arc

between every pair of nodes i and j. The following algorithm for constructing a tour seems to be natural. Starting from node 1, we always move to the closest unvisited node. This natural algorithm is only a heuristic, because the solution obtained may be arbitrarily bad. Consider the sample problem shown in Figure 3.13. It is easy to check that, starting from node 1, we visit the nodes in order 1-2-3-4-1 obtaining a tour with cost $3 + K$. The value of K can be arbitrarily large so the worst case ratio for this algorithm is $(3 + K)/6$, which can also be arbitrarily large. Nevertheless, the nearest neighbor heuristic is very fast and may be useful if some tour should be constructed quickly.

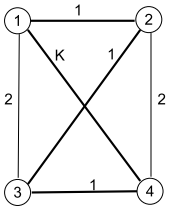


Figure 3.13: A sample traveling salesperson problem.

3.5 Local search

One of the most general methods of solving hard discrete optimization problems is *local search*. The idea of local search is as follows. We start with an initial solution x_0. We then move to another solution x_1 in some neighborhood of x_0. The next solution x_2 is obtained by moving to some solution in the neighborhood of x_1. We continue this until some stopping criterion is satisfied. A *neighborhood function* $\mathcal{N}$ is a function which specifies for each solution $x \in sol(I)$ a set $\mathcal{N}(x) \subseteq sol(I)$, which is called the *neighborhood* of x. Consider the following example:

Example 1 (traveling salesperson). Let us denote by π a subset of arcs of G which forms a tour. Then $\mathcal{N}_k(\pi) = \{\pi' : |\pi \setminus \pi'| \leq k\}$ is called a *k-opt* neighborhood. Hence, $\mathcal{N}_k(\pi)$ contains all tours, which differ from π in at most k arcs. The 2-opt neighborhood is illustrated in Figure 3.14. A sample tour π is shown in Figure 3.14a. We get a tour $\pi' \in \mathcal{N}_2(\pi)$ by removing two arcs in π and adding two new arcs to the resulting subset of arcs. Observe that $|\pi \setminus \pi'| = 2$.

A solution $\hat{x} \in sol(I)$ is called a *local minimum* if $f(\hat{x}) \leq f(x)$ for all $x \in \mathcal{N}(\hat{x})$. Hence, $\hat{x}$ is a local minimum if there is no better solution in its neighborhood. In a similar way, we can define a *local maximum*. Observe that the definition of a local minimum (maximum) depends on the neighborhood function $\mathcal{N}$. In general, a local minimum (maximum) can be different from the global one. Having defined a neighborhood function, we can construct the simplest local search algorithm called the *iterative improvement* (see Figure 3.15).

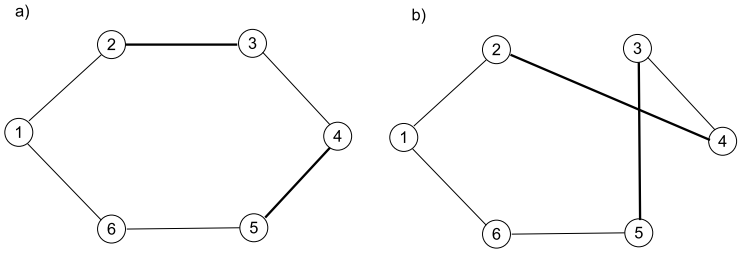


Figure 3.14: a) A sample tour π. b) A sample tour $\pi' \in N_2(\pi)$.

1: Generate a starting solution $x \in sol(I)$
2: **repeat**
3: Choose $x' \in \mathcal{N}(x)$ with the minimal cost $f(x')$
4: **if** $f(x') < f(x)$ **then** $x := x'$
5: **until** $f(x') \geq f(x)$ for all $x' \in \mathcal{N}(x)$
6: **return** x

Figure 3.15: The iterative improvement algorithm for a minimization problem [37].

The idea of the iterative improvement algorithm is very simple. We start with an initial solution x, which may be generated randomly. We then scan the neighborhood of x and seek a best solution x' in this neighborhood. If the cost of x' is less than the cost of x, then we move to x' and repeat the procedure. The algorithm stops if the current solution cannot be improved, i.e. if x is a local minimum. The iterative improvement algorithm is simple and also fast in many cases. Furthermore, it often returns quite good solutions. In particular, it performs well for the traveling salesperson problem with the 3-opt neighborhood function.

The iterative improvement algorithm has two drawbacks. It always stops at a local minimum and scans the whole neighborhood of the current solution, which may be inefficient if the neighborhood size is large. Instead of enumerating all the solutions in $\mathcal{N}(x)$, we may select $x' \in \mathcal{N}(x)$ uniformly at random. Hence, x' is chosen randomly from $\mathcal{N}(x)$ with probability $1/|\mathcal{N}(x)|$. We move to the solution x' if $f(x') - f(x) < t$, i.e. we move to x' if the increase in cost is smaller than a given threshold $t > 0$. The threshold t is a parameter which allows us to control the search process. Large values of t enable us to explore large parts of the solution space, while small values of t allow us to explore a part of the solution space more exhaustively. A good idea is to fix a large threshold at the beginning of the algorithm and then decrease its value during its execution. The algorithm stops after performing a given number of iterations. The threshold acceptance algorithm is shown in Figure 3.16

The threshold acceptance algorithm can be additionally refined by modifying the rule of moving to worse solutions. We move to the solution $x' \in \mathcal{N}(x)$

1: Generate a starting solution $x \in sol(I)$
2: $x_{best} := x$
3: **repeat**
4: Choose $x' \in \mathcal{N}(x)$ uniformly at random.
5: **if** $f(x') < f(x_{best})$ **then** $x_{best} := x'$
6: **if** $f(x') - f(x) < t$ **then** $x := x'$
7: **until** stop criterion
8: **return** x_{best}

Figure 3.16: Threshold acceptance for a minimization problem [37].

with probability 1 if $f(x') < f(x)$ and with probability $\exp((f(x) - f(x'))/t_k)$ if $f(x') \geq f(x)$. Observe that the probability of moving to a worse solution is positive, but less than 1. Furthermore, the smaller is the difference $f(x) - f(x')$ the smaller is the probability of moving to x'. The parameter t_k is called the *temperature* and it allows us to control the search process. The larger t_k the larger is the probability of moving to worse solutions. At the beginning, we should choose a large value for t_k and decrease it during the execution of the algorithm. The resulting algorithm is called *simulated annealing* (see Figure 3.17) in analogy to the physical process for obtaining low energy states of a solid in a heat bath.

1: Generate a starting solution $x \in sol(I)$
2: $x_{best} := x$
3: $k := 1$
4: **repeat**
5: Choose $x' \in \mathcal{N}(x)$ uniformly at random.
6: **if** $f(x') < f(x_{best})$ **then** $x_{best} := x'$
7: **if** $f(x') \leq f(x)$ **then** $x := x'$
8: **else**
9: **if** $\exp((f(x) - f(x'))/t_k) >$rand[0,1) **then** $x := x'$
10: $k := k + 1$
11: **until** stop criterion
12: **return** x_{best}

Figure 3.17: Simulated annealing for a minimization problem [37].

The last local search algorithm is a modification of the iterative improvement algorithm. We also scan the whole neighborhood of the current solution x and choose the best solution $x' \in \mathcal{N}(x)$. However, we do not terminate if x' is local minimum and we always move to x'. So we always move to the neighbor that returns the smallest increase in cost. The straightforward application of this idea may lead to cycling, i.e. we may repeatedly return to the same local minimum, either immediately or within a few iterations. In order to avoid such undesired behavior, we introduce the so-called *tabu list*. The tabu list contains descriptions of the moves which are forbidden for a number of iterations.

Consider, for example, the traveling salesperson problem with the 2-opt neighborhood function. Suppose that we have performed the move from tour π to π' shown in Figure 3.14. In order to forbid going back to π, we can add to the tabu list the two elements $\{(2,3),5\}$ and $\{(4,5),5\}$, which means that we forbid moves in which we add edges $(2,3)$ or $(4,5)$ for the next 5 iterations. Observe that the tabu list contains just some attributes of the forbidden moves. In consequence, moving to some unvisited solutions may be prohibited. To prevent us from missing attractive solutions, we equip the algorithm with the so-called *aspiration criterion*, which may work as follows. If moving to a solution x is prohibited by the tabu list, then we nevertheless accept x if it is better than the best solution found so far. We will say that the solution $x' \in \mathcal{N}(x)$ is *admissible* if the move to x' is not prohibited by the tabu list or is allowed because the aspiration criterion is satisfied. The resulting algorithm, called a *tabu search* is shown in Figure 3.18.

1: Generate a starting solution $x \in sol(I)$
2: $T := \emptyset$
3: $x_{best} := x$
4: **repeat**
5: Choose the best admissible solution $x' \in \mathcal{N}(x)$
6: **if** $f(x') < f(x_{best})$ **then** $x_{best} := x$
7: $x := x'$
8: Update the tabu list T
9: **until** stop criterion
10: **return** x_{best}

Figure 3.18: Tabu search for a minimization problem [37].

Local search algorithms are very general and, in most cases, easy to implement. They have been successfully applied to many hard discrete optimization problems, including the traveling salesperson problem.

3.6 Summary

1. A lot of important discrete optimization problems are computationally hard, which means that no efficient polynomial time algorithms are known for them.

2. Perhaps the easiest method of solving a hard optimization problem is to formulate a linear programming model for it and apply one of several available packages to get an optimal solution. In many cases, the packages are able to provide an optimal solution in reasonable time.

3. For a hard problem, we can design a branch and bound algorithm or an algorithm based on the dynamic programming technique. The branch and bound algorithm is often the best exact method of solving a hard problem.

4. If the instance of a hard problem is large and we have to quickly compute a good solution, then we can apply a k-approximation algorithm. This algorithm outputs a solution, which is not necessarily optimal, but is of some guaranteed quality. However, not every problem has a k-approximation algorithm (see Appendix C). In this case, we can use a heuristic which, however, may return an arbitrarily bad solution in the worst case. Nevertheless, we can expect that the heuristic will give reasonable solutions for typical instances.

5. Local search algorithms can be seen as sophisticated heuristics. They are easy to implement and return good solutions in many cases. They are a good choice if a problem is hard and the instance is very large.

   There is an extensive literature on various methods of solving hard problems. Building mathematical models for practical problems is an important area of operations research (see. e.g. [50]). A description of the branch and bound method, together with some applications, can be found, for example, in [35]. An application of the branch and bound method to the traveling salesperson problem can be found, for example, in [6]. The branch and bound algorithm for integer linear programming can be found in [24]. Dynamic programming is described in [46]. Approximation is an important field of computer science. Many results and algorithms can be found in [47] and [5]. The Christofides algorithm for the metric traveling salesperson problem was first described in [12]. A good description of various methods of solving hard problems can be found in [36]. A lot of results on local search can be found in [37]. Some applications of local search to large scale problems can be found in [2]. The simulated annealing algorithm was developed in 1983 [30] and the tabu seach technique was proposed in 1986. More information on tabu seach can be found in [25].

3.7 Exercises

1. A factory wants to open m stores. The factory considers n places, so that $n > m$. In place i a store with capacity C_i can be built. The distance between places i and j is equal to d_{ij}. The factory wants to open m stores to maximize their total capacity. However, the stores should be located so that the distance between any pair of open stores is not greater than K. Build a model for this problem.

2. An area of some factory is divided into mn squares. Some squares are occupied by valuable packs (we assume that every square may be occupied by at most one pack). We must place some cameras in the area to observe the packs. Every camera can observe r squares to the left, r squares to the right, r squares up and r squares down. It is prohibited to place cameras on the squares occupied by packs. Where should we place cameras so that every pack is observed and the total number of cameras is minimized? Build a model for this problem.

3. In some area there is a set $\boldsymbol{C}$ of cities with a symmetric matrix D containing travel times (in minutes) between each pair of cities. For each city $i \in \boldsymbol{C}$ there is a number u_i denoting the number of citizens in i. The government wants to build k fire stations and each fire station must be placed in a different city. The fire stations should be placed so that the traveling time from each city to the nearest fire station is not greater than 10 minutes. Because the budget is limited, it may be hard to satisfy this demand. Where should the fire stations be placed so that the number of citizens for whom no fire station is reachable within 10 minutes is minimal? Build a model for this problem.

4. We are given an undirected graph $G = (V, E)$. A *cut* $[V_1, V_2]$ in G is a partition $V_1 \cup V_2 = V$ such that $V_1 \cap V_2 = \emptyset$. Let $u(V_1, V_2)$ denote the number of arcs with one endpoint in V_1 and one endpoint in V_2. We wish to determine a cut, for which the value of $u(V_1, V_2)$ is maximized. Build a model for this problem.

5. We are given an undirected graph $G = (V, E)$. We wish to assign a color to each node of the graph so that no two adjacent nodes share the same color. We would like to find such a coloring using the minimum number of distinct colors. Build a model for this problem.

6. Build linear programming models for all the network flow problems considered in Chapter 2.

7. Use the branch and bound algorithm to solve the following discrete optimization problem:

$$\begin{array}{ll} \min & -5x_1 - 4x_2 \\ & x_1 + 2x_2 \leq 6 \\ & -2x_1 + x_2 \leq 4 \\ & 5x_1 + 3x_2 \leq 15 \\ & x_1, x_2 \geq 0,\ x_1, x_2 \text{ integer} \end{array}$$

8. Design a branch and bound algorithm for the knapsack problem and, using it, solve the following problem:

$$\begin{array}{l} \max 5x_1 + 3x_2 + 6x_3 + 6x_4 + 2x_5 \\ 5x_1 + 4x_2 + 7x_3 + 6x_4 + 2x_5 \leq 15, \\ x_1, \ldots, x_5 \in \{0, 1\} \end{array}$$

9. Show that the cost of a minimum spanning tree of G is a lower bound on the cost of the optimal traveling salesperson's tour in G.

10. Apply dynamic programming to solving the traveling salesperson problem.

11. Using the dynamic programming, solve the knapsack problem with 4 items whose weights are 4,2,3,1 and values are 50, 15, 20, 12, respectively and the capacity $W = 6$.

12. Consider the maximum cut problem from Exercise 4. Let us define the following neighborhood function for this problem: the cut $[V_1', V_2']$ is in a neighbor of $[V_1, V_2]$ if $|V_1' \setminus V_1| \leq 1$ and $|V_1' \setminus V_1| \leq 1$ (i.e. we obtain $[V_1', V_2']$ by moving one node between V_1 and V_2). Show that the iterative improvement algorithm, starting with any cut, is a 2-approximation algorithm for this problem.

13. Consider the following *minimum Steiner tree* problem. Given an undirected network $G = (N, A)$ with nonnegative arc costs and whose nodes are partitioned into two sets, required and Steiner, find a minimum cost tree in G (i.e. an acyclic and connected subnetwork of G) that contains all the required nodes and any subset of the Steiner nodes.

    (a) Show that the shortest path and minimum spanning tree problems are special cases of the minimum Steiner tree problem.

    (b) Construct a 2-approximation algorithm for this problem.

    (c) Propose a neighborhood function for this problem.

14. Consider the following simple heuristic for the knapsack problem: sort the items by decreasing ratio of of value to weight, and then greedily pick the items in this order until the capacity W is not violated.

    (a) Show that this algorithm may return a solution which is arbitrarily far from the optimum. Hence, it is only a heuristic.

    (b) Can you improve this algorithm to guarantee the worst-case ratio of 2?

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, 1983.

[2] R. Ahuja, Ö. Ergun, J. Orlin, and A. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows. Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[4] S. Arora and B. Barak. *Computational Complexity. A Modern Approach*. Cambridge University Press, 2009.

[5] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation. Combinatorial Optimization Problems and Their Approximability Properties*. Springer, 1999.

[6] E. Balas and P. Toth. Branch and bound methods. In E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, New York, 1985.

[7] M. S. Bazaraa, J. J. Jarvis, and H. F. Sherali. *Linear Programming and Network Flows*. John Wiley & Sons, New York, second edition, 1990.

[8] C. F. Bazlamaçci and K . S. Hindi. Minimum-weight spanning tree algorithms A survey and empirical study. *Computers & OR*, 28(8):767–785, 2001.

[9] R. E. Burkard. Selected topics on assignment problems. *Discrete Applied Mathematics*, 123(1-3):257–302, 2002.

[10] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM*, 47:1028–1047, 2000.

[11] D.S. Chen, R.G. Batson, and Y. Dang. *Applied Integer Programming. Modeling and Solution*. John Wiley and Sons, 2 edition, 2010.

[12] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, GSIA, Carnegie-Mellon University, Pittsburgh, 1976.

[13] S. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing.*, pages 151–158, 1971.

[14] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[15] W. H. Cunningham. A network simplex method. *Mathematical Programming*, 11:105–116, 1976.

[16] G. B. Dantzig. Application of the simplex method to a transportation problem. In T. C. Koopmans, editor, *Activity Analysis and Production and Allocation*. Wiley, 1951.

[17] R. Diestel. *Graph theory*. Springer-Verlag, 2 edition, 2000.

[18] E. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[19] Edmonds and Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19, 1972.

[20] R. W. Floyd. Shortest path. *Commun. ACM*, 5(6):345, 1962.

[21] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[22] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, 1962.

[23] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, San Francisco, 1979.

[24] R. Garfinkel and G. Nemhauser. *Integer Programming*. John Wiley and Sons Inc., 1972.

[25] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.

[26] F. L. Hitchcock. The distribution of a product from several sources to numerous localities. *J. Math. Phys.*, 20:224–230, 1941.

[27] http://www-01.ibm.com/software/integration/optimization/cplex optimizer/.

[28] http://www.gnu.org/software/glpk/.

[29] W. S. Jewell. Optimal flow through networks. *Operations Research*, 10:476–499, 1962.

[30] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[31] M. Klein. A primal method for minimal cost flows. *Management Science*, 14:205–220, 1967.

[32] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proc. Am. Math. Soc.*, pages 48–50, 1956.

[33] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.

[34] E. Lawler. *Combinatorial Optimization. Networks and Matroids*. Saunders College Publishing, 1976.

[35] E. L. Lawler and D. E. Wood. Branch and bound methods: a survey. *Operations Research*, 14:699–719, 1984.

[36] Z. Michalewicz and B. F. Fogel. *How to Solve It. Modern Heuristics*. Springer-Verlag, 2000.

[37] W. Michiels, E. Aarts, and J. Korst. *Theoretical Aspect of Local Search*. Springer - Verlag, 2007.

[38] J. Nesetril, E. Milkova, and H. Nesetrilova. Otokar łboruvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233:3–36, 2001.

[39] J. B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 77:109–129, 1997.

[40] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[41] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization. Algorithms and Complexity*. Prentice-Hall, 1982.

[42] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[43] K. A. Ross and C. R. B. Wright. *Discrete Mathematics*. Prentice-Hall, 1985.

[44] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley and Sons, 1986.

[45] R. Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

[46] M. Sniedovich. *Dynamic Programming: Foundations and Principles*. CRC Press, 2 edition, 2010.

[47] V. V. Vazirani. *Approximation Algorithms*. Springer, 2004.

[48] H. M. Wagner. On a class of capacitated transportation problems. *Management Science*, 5:304–318, 1959.

[49] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[50] H.P. Williams. *Model Building in Mathematical Programming*. John Wiley and Sons, 1994.

A. Networks

In this chapter we describe some graph theoretic definitions, which are necessary to understand the material presented in this book. We also present some basic network algorithms. The material presented in this section is based on [3].

Networks, path and cycles

A *network* (*directed graph*) $G = (N, A)$ consists of a set N of *nodes* and a set A of *arcs*, whose elements are ordered pairs of nodes. A sample network is shown in Figure 19.

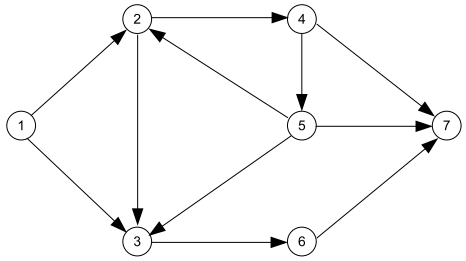


Figure 19: A sample network, where $N = \{1, 2, \ldots, 7\}$ and $A = \{(1,2), (1,3), (2,3), (2,4), (3,6), (4,5), (4,7), (5,7), (6,7)\}$.

A *walk* in $G = (V, A)$ is a sequence of nodes $i_1 - i_2 - \cdots - i_r$ such that either $a_k = (i_k, i_{k+1}) \in A$ or $a_k = (i_{k+1}, i_k) \in A$ for all $1 \leq k \leq r - 1$.

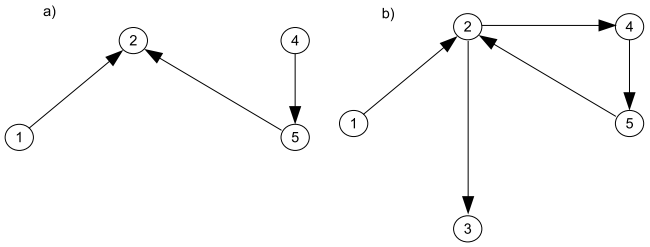


Figure 20: Two walks: $1 - 2 - 5 - 4$ and $1 - 2 - 5 - 4 - 2 - 3$.

A *path* is a walk without any repetition of nodes. A *directed path* is a path in which for any two consecutive nodes i_k and i_{k+1} on the path $(i_k, i_{k+1}) \in A$.

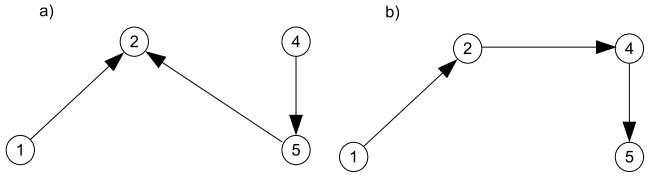


Figure 21: A path $1-2-5-4$ and a directed path $1-2-4-5$.

A *cycle* is a path $i_1 - i_2 - \cdots - i_r$ together with the arc (i_r, i_1) or (i_1, i_r). We will also use the notation $i_1 - i_2 - \cdots - i_r - i_1$. A *directed cycle* is a directed path $i_1 - i_2 - \cdots - i_r$ together with arc (i_r, i_1). A *Hamiltonian cycle* in G is a directed cycle containing all the nodes of G. A network is *acyclic* if it contains no directed cycle. Observe that the network in Figure 19 is not acyclic.

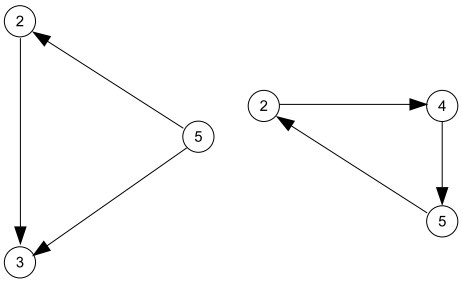


Figure 22: A cycle $2-5-3-2$ and a directed cycle $2-4-5-2$.

A network is *connected* if it contains a path between each pair of its nodes. A network is *strongly connected* if it contains at least one directed path between each pair of its nodes. The network in Figure 19 is connected, but it is not strongly connected. A *tree* is a connected network that contains no cycle. A *spanning tree* of $G = (N, A)$ is a tree that contains all the nodes of G.

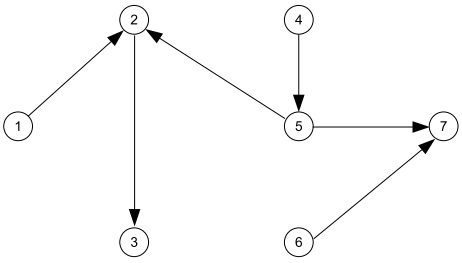


Figure 23: A spanning tree of G

It is not difficult to show that a tree on n nodes contains exactly $n-1$ arcs. Furthermore, each pair of nodes of in a tree is connected by a unique path and adding a new arc to a spanning tree creates a unique cycle.

A network is *bipartite* if we can partition its node set into two subsets N_1 and N_2, so that for each arc $(i,j) \in A$ either $i \in N_1$ and $j \in N_2$ or $i \in N_2$ and $j \in N_1$. It can be shown that a network is bipartite if and only if it does not contain a cycle having an odd number of arcs.

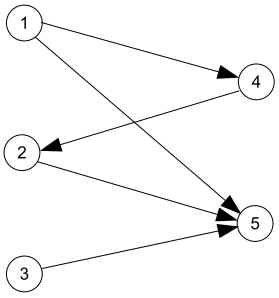


Figure 24: A bibaprtite network, where $N_1 = \{1,2,3\}$, $N_2 = \{4,5\}$.

A *cut* is a partition of the node set N into two parts S and $\overline{S} = N \setminus S$. Each cut defines a set of arcs $[S,\overline{S}]$ that have one endpoint in S and another endpoint in $\overline{S}$. A cut is an $s-t$-*cut* if $s \in S$ and $t \in \overline{S}$. Removing all the arcs from $[S,\overline{S}]$ splits the network into two components.

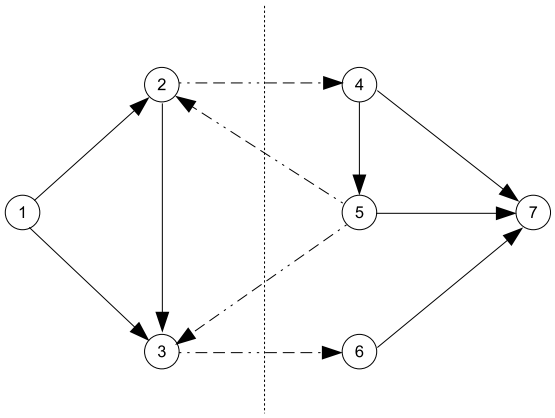


Figure 25: The $1-7$-cut $S = \{1,2,3\}$, $\overline{S} = \{4,5,6,7\}$, $[S,\overline{S}] = \{(2,4),(5,2),(5,3),(3,6)\}$.

A network is *undirected* when the arcs are unordered pairs of distinct nodes. An undirected arc (i,j) can be regarded as a two-way connection between i and j, or as a pair of arcs, (i,j) and (j,i). When drawing an undirected network, we omit all arrows. Walks, paths, trees, cycles and cuts have the same definitions

as for directed graphs, except that we do not distinguish between directed and undirected cases.

Network representations

An instance of a network problem is a network $G = (N, A)$, together with arc costs c_{ij} and arc capacities u_{ij} for all $(i, j) \in A$. The size of such an instance is specified by four numbers: the number of nodes $n = |N|$, the number of arcs $m = |A|$, the quantity $\log C$, where C is the largest arc cost and the quantity $\log U$, where U is the largest arc capacity. A sample instance is shown in Figure 26.

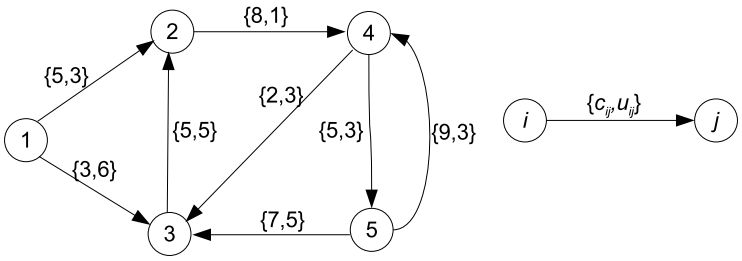


Figure 26: An instance of a network problem.

The first method of representing a network is the *node - arc incidence matrix*. The rows of this matrix correspond to nodes and the columns correspond to arcs. For arc (i, j) we write 1 in the ith row and -1 in the jth row. The following node - arc incidence matrix represents the network shown in Figure 26. The arc costs and capacities can be stored in two additional matrices.

	$(1,2)$	$(1,3)$	$(2,4)$	$(3,2)$	$(4,3)$	$(4,5)$	$(5,3)$	$(5,4)$
1	1	1	0	0	0	0	0	0
2	-1	0	1	-1	0	0	0	0
3	0	-1	0	1	-1	0	-1	0
4	0	0	-1	0	1	1	0	-1
5	0	0	0	0	0	-1	1	1

The second method is the *node - node adjacency matrix*. Both the rows and columns of this matrix correspond to nodes and we write 1 in ith row and jth column if $(i, j) \in A$ and 0 otherwise. The following node - node adjacency matrix represents the network shown in Figure 26. As previously, the arc costs and capacities can be stored in two additional matrices.

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	1	0
3	0	1	0	0	0
4	0	0	1	0	1
5	0	0	1	1	0

The third method of representing a network is the *adjacency list*. In this representation, a data structure called a linked list is used. We first create a list linking all the nodes of the network. We then link to each node an additional list containing all the adjacent nodes. The adjacency list for the network in Figure 26 is shown in Figure 27. Notice that we can store additional information for each arc, in particular the arc costs and capacities.

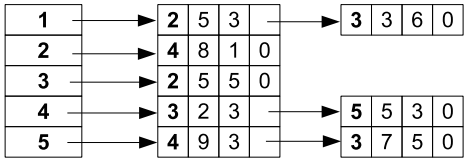


Figure 27: The adjacency list for the network in Figure 26

A comparison of the network representations is shown in Table 3.

Representation	**Input size**	**Features**
Node-arc inc. matr.	knm	1. Space inefficient 2. Too expensive to manipulate 3. Important for theoretical reasons
Node-node adj. matr.	kn^2	1. Suited to dense networks 2. Easy to implement
Adjacency list	$k_1 n + k_2 m$	1. Space efficient 2. Efficient to manipulate 3. Suited to dense and sparse netw. 4. More difficult to implement.

Table 3: Comparison of different network representations [3]. Input size is the number of bits required to store the network, k, k_1, k_2 are constants.

Basic network algorithms

Consider the following *search problem*: given a directed network $G = (N, A)$ find all the nodes that are reachable by directed paths from a given node $s \in N$. An algorithm for solving this problem is shown in Figure 28. Its idea is very simple. It iteratively marks all the nodes which are reachable from s and adds the marked nodes to $LIST$. A node i is removed from $LIST$ if all the nodes j such that $(i, j) \in A$ have been marked. The algorithm additionally keeps the direct predecessor $pred(i)$ of every node $i \in N$ along a directed path from node s. Therefore, when the algorithm terminates we can easily retrieve a directed path from s to $i \in N$ (if such a path exists).

This search algorithm is fast. It runs in $O(m)$ time. It also has the following property: if in line 5 we always select nodes from the front of the $LIST$ (i.e.

in FIFO order), then we get paths from s to all the reachable nodes with the minimum number of arcs.

```
1: Unmark all nodes in N
2: Mark node s
3: pred(i) := 0 for all i ∈ N, LIST := {s}
4: while LIST ≠ ∅ do
5:    Select a node i in LIST
6:    if node i is incident to an admissible* arc (i, j) then
7:       Mark node j
8:       pred(j) := i
9:       Add node j to LIST.
10:   else
11:      Delete node i from LIST
12:   end if
13: end while
```

Figure 28: Search algorithm [3]. (*) Arc (i, j) is admissible if j is unmarked.

Consider the following *reverse search problem*: given a directed network $G = (N, A)$. Find all the nodes in the network from which we can reach a given node t along a directed path. To solve this problem, it is enough to reverse the arc directions in G and run the algorithm for the search problem with $s' = t$ and $t' = s$. Obviously, this also requires $O(m)$ time.

Suppose now that we would like to verify whether a given directed network G is strongly connected. A simple approach to solving this problem would be to run the search algorithm for each node in G in turn as a starting node. This would require $O(nm)$ time. There is, however, a much faster method. Let us choose any node s of G. It is easy to verify that the network G is strongly connected if and only every node is reachable from s and node s is reachable from every other node of G. The implication $\Rightarrow$ is obvious. To see why the implication $\Leftarrow$ is true, consider any two nodes i and j in G such that $i \neq j$. Since s is reachable from i and j is reachable from s, there must be a directed path from i to j. Using this simple observation, we can determine the strong connectivity of G by executing the search algorithm twice, solving the search and reverse search problems. This requires $O(m)$ time.

Let us label the nodes of the network $G = (N, A)$ by distinct numbers from 1 through $n = |N|$. We say that this labeling is a *topological ordering* of nodes if $(i, j) \in A$ implies $i < j$. It can be shown that G has a topological ordering of nodes if and only if it is acyclic. So by establishing a topological ordering, we can ensure that G contains no directed cycle. Establishing a topological ordering is not difficult. We present the idea of the algorithm using the example shown in Figure 29. We first seek a node, which has no incoming arcs and we give it the label 1 (see Figure 29a). Such a node must exist in every acyclic network. We then delete this node together with all the arcs emanating from it. We again seek a node with no incoming arcs and give it the label 2 (see

Figure 29b). We repeat this procedure until all the nodes are labeled (the input graph is acyclic) or there is no node with no incoming arcs (the graph is not acyclic). In the former case, a topological ordering has been established.

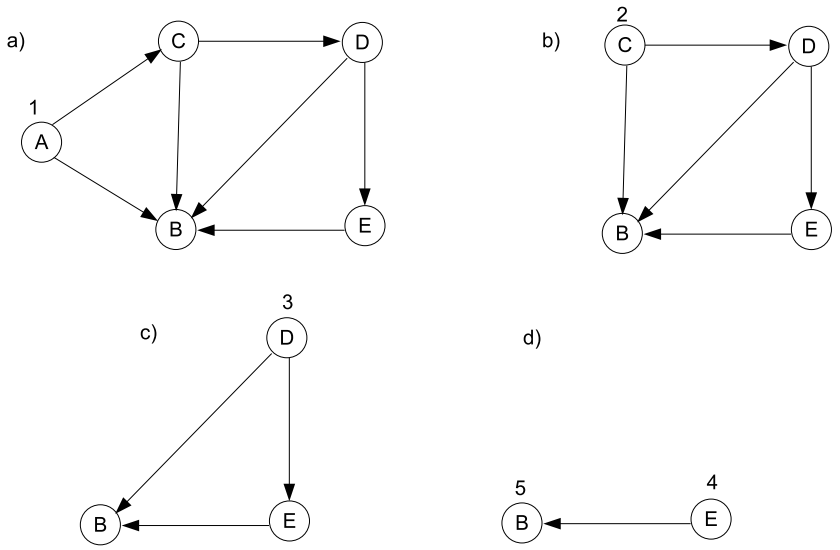


Figure 29: Establishing a topological ordering.

B. Linear programming

In this section we briefly discuss one of the most important class of optimization problems, namely linear programming problems. We present some properties of this class and show its connections with the minimum cost flow problem. A *linear programming problem* is of the following form:

$$\begin{array}{ll} \min & c_1x_1 + c_2x_2 + \cdots + c_qx_q \\ & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1q}x_q = b_1 \\ & \cdots \\ & a_{p1}x_1 + a_{p2}x_2 + \cdots + a_{pq}x_q = b_p \\ & 0 \leq x_i \leq u_i \qquad\qquad i = 1, \ldots, q \end{array} \tag{2}$$

The problem consists of an objective function, which should be minimized, and a set of linear constraints. The first p constraints have the form of linear equalities and the last q constraints have the special form of *boundary constraints*, which bound the values of all the variables from above and below. It can be shown that (2) is the most general form of the linear programming problem. If we wish to maximize the objective function, then we multiply it by -1 and minimize. If a constraint has the form of a $\leq$ or $\geq$ inequality, then we can convert it to an equality by adding or subtracting an additional slack variable from the left hand side of this constraint. If some variable x_i is unrestricted in sign, then we replace x_i with $x_i^+ - x_i^-$, $x_i^+ \geq 0$, $x_i^- \geq 0$. If some variable x_i is not bounded from above, then $u_i = \infty$ and we simply write $x_i \geq 0$. We will also assume that $b_i \geq 0$ and b_i is integer for all $i = 1, \ldots, p$. If $b_i < 0$ then we multiply the ith constraint by -1. If b_i is rational, then we multiply the ith constraint by a suitable constant.

The class of linear programming problems is perhaps the most important class of optimization problems, which can be solved efficiently. A polynomial algorithm for this problem was discovered by Leonid Khachiyan in 1979. In practice the simpler *simplex algorithm*, discovered by George Dantzig in 1947, is often used. The simplex algorithm does not run in polynomial time. It is, however, fast for most problems arising in practice. In this section we describe the main idea of the simplex method and we show how it can be applied to the minimum cost flow problem.

Basis structure

Let us partition the set of q variables into three sets $\boldsymbol{B}$, $\boldsymbol{L}$ and $\boldsymbol{U}$, where $\boldsymbol{B}$ contains exactly p variables, for which the vectors $[a_{1i}, a_{2i}, \ldots, a_{pi}]$, $i \in \boldsymbol{B}$, are linearly independent, $\boldsymbol{L}$ is the set of variables whose values are equal to 0, and $\boldsymbol{U}$ is the set of variables whose values are equal to their upper bounds u_i. The set $\boldsymbol{B}$ is called a *basis*, the variables in $\boldsymbol{B}$ are called *basic variables* and the partition $(\boldsymbol{B}, \boldsymbol{L}, \boldsymbol{U})$ is called a *basis structure*. For the simplicity of the presentation, we renumber the variables so that $\boldsymbol{B} = \{x_1, \ldots, x_p\}$. The values of the variables in $\boldsymbol{L}$ and $\boldsymbol{U}$ are determined, so if we fix $x_i = 0$ for all $x_i \in \boldsymbol{L}$ and $x_i = u_i$ for all $x_i \in \boldsymbol{U}$, then the constraints take the following form:

$$\begin{array}{ll} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1p}x_p = b'_1 & \\ \cdots & \\ a_{p1}x_1 + a_{p2}x_2 + \cdots + a_{pp}x_p = b'_p & \\ 0 \le x_i \le u_i & i = 1, \ldots, p \end{array} \tag{3}$$

Now observe, that there is a unique solution to the first p constraints, since they form a system of p linear equations with p variables and these constraints are linearly independent. We can solve this system and check whether the obtained solution satisfies the boundary constraints. If this is the case, then we say that the basis structure $(\boldsymbol{B}, \boldsymbol{L}, \boldsymbol{U})$ is *feasible* and is associated with the unique solution.

Optimality conditions

Consider a feasible spanning tree structure $(\boldsymbol{B}, \boldsymbol{L}, \boldsymbol{U})$. As previously, we renumber the variables so that $\boldsymbol{B} = \{x_1, \ldots, x_p\}$. We know that this structure corresponds to a unique feasible solution to the linear programming problem. Now our aim is to check whether this solution is optimal. Let us assign numbers $\pi(1), \ldots, \pi(p)$, called *potentials*, to the first p constraints of (2). We can transform the objective function in the following way:

$$\sum_{j=1}^{q} c_j x_j + \sum_{i=1}^{p} [\pi(i)(\sum_{j=1}^{q} a_{ij}x_{ij} - b_i)]. \tag{4}$$

This transformation does not change the value of the objective function, because the second term in the sum equals 0. After easy algebraic manipulations, we can rewrite (4) as follows:

$$\sum_{j=1}^{p} [c_j - \sum_{i=1}^{p} \pi(i)a_{ij}]x_j + \sum_{j=p+1}^{q} [c_j - \sum_{i=1}^{p} \pi(i)a_{ij}]x_j + \sum_{i=1}^{p} \pi(i)b(i) \tag{5}$$

The quantity $c_j^\pi = c_j - \sum_{i=1}^{p} \pi(i)a_{ij}$ is the *reduced cost* of variable x_j with respect to the potentials π. We can now choose potentials π, so that the reduced costs of all the basic variables are 0. Namely, we compute the potentials by solving the following system of linear equations $c_j - \sum_{i=1}^{p} \pi(i)a_{ij} = 0, \quad j =$

$1, \ldots, p$. This is a system of p linear equations with p variables. Furthermore, these equations are linearly independent, so the system has a unique solution. So finally the objective can be expressed as:

$$\sum_{j=p+1}^{q} c_j^{\pi} x_j + \sum_{i=1}^{p} \pi(i) b(i) = \sum_{j \in \boldsymbol{U}} c_j^{\pi} u_j + \sum_{i=1}^{p} \pi(i) b(i). \tag{6}$$

Now observe that if $c_j^{\pi} \geq 0$ for all $j \in \boldsymbol{L}$ and $c_j^{\pi} \leq 0$ for all $j \in \boldsymbol{U}$, then it is not possible to decrease the value of the objective function (6). It follows from the fact that all the decision variables x_j are nonnegative. Let us summarize this observation:

Theorem 24 *A solution corresponding to a feasible basis structure* $(\boldsymbol{B}, \boldsymbol{L}, \boldsymbol{U})$ *is optimal if for some potentials* $\pi(1), \ldots, \pi(p)$ *the reduced costs* c_j^{π}, $j = 1, \ldots, q$, *satisfy the following optimality conditions:*

1. $c_j^{\pi} = 0$ *for all* $j \in \boldsymbol{B}$
2. $c_j^{\pi} \geq 0$ *for all* $j \in \boldsymbol{L}$
3. $c_j^{\pi} \leq 0$ *for all* $j \in \boldsymbol{U}$

Moving to another feasible basis structure

Suppose now that a feasible basis structure $(\boldsymbol{B}, \boldsymbol{L}, \boldsymbol{U})$ does not satisfy the optimality conditions from Theorem 24. We can transform the constraints of (2) and represent them in terms of the basic variables $x_1, \ldots, x_p$ in the following way:

$$\begin{array}{llll} x_1 + & & & + a'_{1p+1} x_{p+1} + \cdots + a'_{1q} x_q = b'_1 \\ & x_2 + & & + a'_{2p+1} x_{p+1} + \cdots + a'_{2q} x_q = b'_2 \\ \ldots & & & \\ & & x_p + & + a'_{pp+1} x_{p+1} + \cdots + a'_{pq} x_q = b'_p \\ 0 \leq x_i \leq u_i & i = 1, \ldots, q & & \end{array} \tag{7}$$

We then get the solution $\hat{x}_j = b'_j - \sum_{i \in \boldsymbol{U}} a'_{ij} u_i$ for $j \in \boldsymbol{B} = \{1, \ldots, p\}$, $x_j = 0$ for $j \in \boldsymbol{L}$ and $x_j = u_j$ for $j \in \boldsymbol{U}$.

Assume that $c_j^{\pi} < 0$ for some $x_j \in \boldsymbol{L}$. Then by increasing the value of x_j, we can decrease the value of the objective function. We would like to compute the maximum value x_j can be increased to. Consider the entires a'_{ij} for $i = 1, \ldots, p$. If $a'_{ij} > 0$, then we can increase x_j by at most $\delta_i = \hat{x}_i / a'_{ij}$ units and keep the value of the basic variables $\hat{x}_i$ nonnegative. If $a'_{ij} < 0$, then we can increase x_j by at most $\delta_i = |(u_i - \hat{x}_i)/a'_{ij}|$ units and keep the value of $\hat{x}_i$ not greater than u_i. Finally x_j cannot be increased by more than u_j. So we take the minimum of u_j and all the δ_i and increase x_j by this amount. It may, however, happen that no upper bound on the increase in x_j can be established (for example, if all $a'_{ij} < 0$ and $u_i = \infty$ for all $i = 1, \ldots, q$). In this case, the problem is unbounded and has no optimal solution. If, after an increase in x_j, $x_j < u_j$, then at least

one variable from $\boldsymbol{B}$, say x_k, satisfies $x_k = 0$ or $x_k = u_k$. In the former case, we move x_k to $\boldsymbol{L}$ and in the latter one, we move x_k to $\boldsymbol{U}$. We also move the variable x_j to $\boldsymbol{B}$ in this way obtaining another feasible basic structure. If $x_j = u_j$, then the new feasible basic structure is obtained by moving x_j from $\boldsymbol{L}$ to $\boldsymbol{U}$. Notice that we also get a feasible solution corresponding to this new basis structure. The reasoning is exactly the same if $c_j^\pi > 0$ for some variable $x_j \in \boldsymbol{U}$. In this case, we have to decrease the value of x_j, so we must compute the minimum possible new value for x_j. We leave the details as an exercise.

Simplex algorithm

The implementation of the reasoning presented in the previous points is known as the *simplex algorithm*. There are many technical details involving this implementation and we refer to the existing literature for a description of them. The general idea is as follows. We start with an initial feasible basis structure (some methods of obtaining such a structure can be found in the literature [7, 44]). We then check whether it is optimal by computing the potentials and reduced costs. If the optimality conditions are violated, then we compute another feasible basis structure by moving some variables between the sets $\boldsymbol{B}$, $\boldsymbol{L}$ and $\boldsymbol{U}$. The algorithm terminates if the current structure satisfies the optimality conditions. However, the algorithm may not terminate in a finite time, because some moves may be *degenerate*, i.e. they do not improve the value of the objective function. This happens, when the upper bound on the increase (or decrease) in the value of the variable entering $\boldsymbol{B}$ is 0. In this case, only the basis structure is changed, while the values of all the variables are unchanged. If we are not careful, then the algorithm may fall into an infinite loop, repeating a sequence of degenerate moves. There are, however, some simple methods of dealing with degeneracy, which ensure that the simplex algorithm runs in finite time (see the literature).

Example 1. Consider the following problem:

$$\begin{aligned} \min \quad & x_1 + x_2 - 2x_3 + 5x_4 \\ & 2x_1 - x_2 - x_3 + x_4 = 13 \\ & x_1 + x_2 + 2x_3 - x_4 = 3 \\ & 0 \le x_1 \le 7, 0 \le x_2 \le 6, 0 \le x_3 \le 5, 0 \le x_4 \le 3 \end{aligned}$$

Let us choose the initial basis structure $\boldsymbol{B} = \{x_1, x_2\}$, $\boldsymbol{L} = \{x_3\}$ and $\boldsymbol{U} = \{x_4\}$. So we set $x_3 = 0$, $x_4 = 3$ and solve the following system of linear equations:

$$\begin{aligned} 2x_1 - x_2 &= 10 \\ x_1 + x_2 &= 6 \end{aligned}$$

We obtain the feasible solution $x_1 = 16/3$, $x_2 = 2/3$, $x_3 = 0$, $x_4 = 3$ corresponding to the basis structure. Let us now associate potentials $\pi(1)$ and $\pi(2)$ with the two equality constraints. We compute the potentials by solving the following system of linear equations:

$$\begin{aligned} 2\pi(1) + \pi(2) &= 1 \\ -\pi(1) + \pi(2) &= 1 \end{aligned}$$

Hence, $\pi(1) = 0$ and $\pi(2) = 1$ and the reduced costs are $c_1^\pi = 0$, $c_2^\pi = 0$, $c_3^\pi = -3$ and $c_4^\pi = 6$. Observe that both c_3^π and c_4^π violate the optimality conditions. So we have to move to another basis structure by either increasing the value of x_3 or by decreasing the value of x_4. Let us represent the linear constraints of the sample problem in terms of the basic variables x_1 and x_2:

$$\begin{array}{rrrrr} x_1+ & & +1/3x_3 & & = 16/3 \\ & x_2 & +1/2x_3 & -x_4 & = -7/3 \end{array}$$

Assume first that we increase the value of x_3. Since $x_4 = 3$ we can transform the problem into:

$$\begin{array}{rrrr} x_1+ & & +1/3x_3 & = 16/3 \\ & x_2 & +1/2x_3 & = 2/3 \end{array}$$

Since all the variables are nonnegative and $x_3 \leq 4$, we can increase x_3 up to $\min\{16/3 : 1/3, 2/3 : 1/2, 4\} = 4/3$. After this, the value of x_2 falls to 0. So we get another basic structure, where $\boldsymbol{B} = \{x_1, x_3\}$, $\boldsymbol{L} = \{x_2\}$, $\boldsymbol{U} = \{x_4\}$. The solution corresponding to this structure is $x_1 = 44/9$, $x_2 = 0$, $x_3 = 4/2$ and $x_4 = 3$.

Assume now that we wish to decrease the value of x_4. The value of $x_3 = 0$, so we can represent the problem as:

$$\begin{array}{rrr} x_1 & & = 16/3 \\ & x_2 & = -7/3 + x_4 \end{array}$$

Now it is easily seen that we can decrease the value of x_4 to $7/3$ and keep x_2 nonnegative. After this, the value of x_2 falls to 0. So, we get another basic structure, where $\boldsymbol{B} = \{x_1, x_4\}$, $\boldsymbol{L} = \{x_2, x_3\}$, $\boldsymbol{U} = \emptyset$. The solution corresponding to this structure is $x_1 = 16/3$, $x_2 = 0$, $x_3 = 0$ and $x_4 = 7/3$.

Integer optimal solutions

Of course, an optimal solution to the linear programming problem need not be integer, even if all the input data are integer. Let $(\boldsymbol{B}, \boldsymbol{L}, \boldsymbol{U})$ be a basis structure corresponding to an optimal solution. We first fix $x_i = 0$ for all $i \in \boldsymbol{L}$ and $x_i = u_i$ for all $i \in \boldsymbol{U}$. The values of the basic variables can be then determined by solving the system of linear equations with p variables and p constraints, which is of the form (3). Let us denote by $\mathcal{B}$ the matrix $[a_{ij}]_{p \times p}$ in (3). Since $\mathcal{B}$ is nonsingular, the inverse matrix $\mathcal{B}^{-1}$ exists and the basic variables take the values $\mathcal{B}^{-1}\boldsymbol{b}'^T$, where T denotes the transposition. Observe now that if all the entries of the matrix $\mathcal{B}^{-1}$ are integer, then all the basic variables take integer values.

Let $\mathcal{A}$ be a $p \times q$ matrix with integer elements. We say that $\mathcal{A}$ is *totally unimodular* if each square submatrix of $\mathcal{A}$ has determinant 0, +1 or −1. Assume now that $\mathcal{A} = [a_{ij}]_{p \times q}$ is the matrix of integer coefficients a_{ij} given in (2). If $\mathcal{A}$ is totally unimodular, then any its square submatrix $\mathcal{B}$ corresponding to a basis $\boldsymbol{B}$ has determinant 1 or −1. In this case, we know from linear algebra that all the elements of the inverse matrix $\mathcal{B}^{-1}$ are integer and, consequently,

the problem (2) has an integer optimal solution. Furthermore, this solution can be found by applying the simplex algorithm. Let us summarize this important fact.

Theorem 25 *If the matrix $\mathcal{A} = [a_{ij}]_{p\times q}$ of coefficients given in (2) is totally unimodular, then the values of the variables in the optimal solution returned by the simplex algorithm are integer.*

Relations to the network simplex

The minimum cost flow problem can be formulated as the following linear programming problem:

$$\begin{aligned} \min \quad & \sum_{(i,j)\in A} c_{ij}x_{ij} \\ & \sum_{\{j:(k,j)\in A\}} x_{kj} - \sum_{\{j:(j,k)\in A\}} x_{jk} = b(k) \quad k \in N \\ & 0 \le x_{ij} \le u_{ij} \qquad\qquad\qquad\qquad\qquad (i,j) \in A \end{aligned} \tag{8}$$

Of course, the problem is of the same form as (2). Let us denote by $\mathcal{N}$ the matrix of coefficients a_{ij} in (8). Observe that $\mathcal{N}$ is the node - arc incidence matrix of the network G and all its elements are 0, +1 or -1 (see Appendix A). A sample node - arc incidence matrix, which describes the network from Figure 26, is shown below:

	x_{12}	x_{13}	x_{24}	x_{32}	x_{43}	x_{45}	x_{53}	x_{54}
1	1	1	0	0	0	0	0	0
2	−1	0	1	−1	0	0	0	0
3	0	−1	0	1	−1	0	−1	0
4	0	0	−1	0	1	1	0	−1
5	0	0	0	0	0	−1	1	1

The following theorem characterizes the matrix $\mathcal{N}$.

Theorem 26 ([3]) *The matrix $\mathcal{N}$ is totally unimodular.*

Proof. We need to show that every square submatrix $\mathcal{B}$ of $\mathcal{N}$ of size k has determinant 0, 1 or -1. This result can be established by performing induction on k. It is obviously true for $k = 1$, because each entry of $\mathcal{N}$ is 0, 1 or -1. Suppose now that every square submatrix of $\mathcal{N}$ of size k has determinant 0, 1 or -1. Consider a square submatrix $\mathcal{B}$ of size $k + 1$. Consider three cases. (1) $\mathcal{B}$ contains a column with no nonzero element. Then $\det \mathcal{B} = 0$ and the theorem follows. (2) Every column of $\mathcal{B}$ has exactly two nonzero elements, in which case, one of these elements must be 1 and the other -1. Then summing all the rows of $\mathcal{B}$ yields the zero vector, implying that the rows in $\mathcal{B}$ are linearly dependent and, consequently, $\det \mathcal{B} = 0$. (3) Some column, say the lth column, of $\mathcal{B}$ has exactly one nonzero element in the ith row. This element must be 1 or -1. Using the Laplace expansion, we obtain $\det \mathcal{B} = \pm \det \mathcal{B}'$, where $\mathcal{B}'$ is the

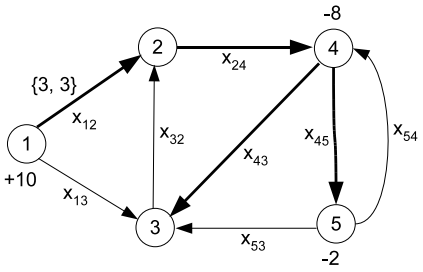
square matrix obtained by deleting the lth column and the ith row. From the induction assumption, $\det \mathcal{B}'$ is 0, 1 or -1 and the theorem follows. ■

Theorem 26 is of great importance. It implies that if all the input data of the minimum cost flow problem are integer, then the problem has an integer optimal solution. Furthermore, this solution can be computed by applying the simplex algorithm. In fact, the network simplex described in Section 2.3.4 is a simplex algorithm, in which the special structure of the minimum cost flow problem is used. Let us analyze this step by step.

Observe first that the rows of $\mathcal{N}$ are linearly dependent. It follows from the fact that adding all these rows together we get the zero vector. We can thus remove any row from $\mathcal{N}$ and there are at most $n-1$ basic variables in the problem. Each variable x_{ij} corresponds to an arc $(i,j) \in A$. Consider a basis structure $(\boldsymbol{B},\boldsymbol{L},\boldsymbol{U})$ for (8). The set $\boldsymbol{B}$ contains at most $n-1$ basic variables. Let us recall that a spanning tree of the network $G=(N,A)$ contains $|N|-1$ arcs. This coincidence is not accidental.

Theorem 27 *A set of arcs T is a spanning tree of G if and only if the variables corresponding to these arcs form a basis for the minimum cost flow problem.*

We omit the proof of the above theorem. It can be found, for example, in [3]. An illustration is shown in Figure 30. We first delete row 1. The spanning tree T of G (shown in bold) corresponds to the basis $\boldsymbol{B}=\{x_{12},x_{24},x_{43},x_{45}\}$.



	x12	x13	x24	x32	x43	x45	x53	x54
1	1	1	0	0	0	0	0	0
2	**-1**	0	**1**	-1	**0**	**0**	0	0
3	**0**	-1	**0**	1	**-1**	**0**	-1	0
4	**0**	0	**-1**	0	**1**	**1**	0	-1
5	**0**	0	**0**	0	**0**	**-1**	1	1

Figure 30: A spanning tree T of G corresponds to a basis of the linear programming problem.

So, in the minimum cost flow problem, a basis structure is equivalent to a spanning tree structure $(\boldsymbol{T},\boldsymbol{L},\boldsymbol{U})$, where $\boldsymbol{T}=\boldsymbol{B}$. It is now easy to check whether $(\boldsymbol{T},\boldsymbol{L},\boldsymbol{U})$ is feasible. We do not need to solve the system of linear equations (7). We can instead perform the computations directly on the corresponding spanning tree (as shown in Section 2.3). Observe that while computing the values of the basic (spanning tree) variables only additions and subtractions are performed. So, if all the input data are integers, then the resulting flows must also be integer.

Each equality constraint in (8) corresponds to some node $i \in N$. Therefore, we can refer to potentials as the node potentials. The reduced costs are $c^{\pi}_{ij} = c_{ij} - \sum_{i=1}^{p} \pi(i)a_{ij} = c_{ij} - \pi(i) + \pi(j)$ and we compute them by solving

the system of equations $c_{ij} - \pi(i) + \pi(j) = 0$ for all $(i, j) \in \boldsymbol{T}$. Again, these computations can be performed directly on the corresponding spanning tree (see Section 2.3). Theorem 17 is now equivalent to Theorem 24. If some variable (arc) violates the optimality condition, then we move to another feasible spanning tree (basis) structure, as shown in Section 2.3. We again perform only additions and subtractions working directly on the corresponding spanning tree. In consequence, the next flow will also be integer. We thus can see that the network simplex algorithm is an adaptation of the simplex algorithm in which the special structure of the minimum cost flow problem is taken into account.

C. NP-completeness

In this chapter we describe some notions of the theory of NP-completeness. More information about this large and important field of computer science can be found in the literature [5, 4, 40, 47, 23]. The theory of NP-completeness gives an evidence that some natural and important discrete optimization problems are computationally intractable. We start by considering the class of *decision problems*. A decision problem Π consists of a set of input data (instances) D_Π and a question such that for each instance $I \in D_\mathcal{D}$ the answer is either **yes** or **no**. Consider the following examples of decision problems:

SAT: We are given a boolean formula $\phi(x_1, \ldots, x_n)$. Is it possible to assign boolean values to the variables $x_1, \ldots, x_n$ so that the value of ϕ is 1? For example, for the formula $(x_1 \wedge x_2) \rightarrow \sim x_2$, the answer is **yes** because we can assign $x_1 = 0$ and $x_2 = 1$, which makes the value of ϕ equal to 1. On the other hand, the answer for the formula $x_1 \wedge \sim x_1 \wedge x_2$ is **no**, because no assignment makes the value of this formula equal to 1.

3-SAT: This problem is similar to the previous one with the exception that the input formula is in the conjunctive normal form, in which each clause contains at most 3 literals. For example, the following formula is a valid instance of this problem: $(x_1 \vee \sim x_2 \vee x_3) \wedge (x_1 \vee x_4) \wedge (\sim x_1 \vee x_3 \vee x_4)$. One can check that the answer to this particular formula is **yes**.

HAMILTONIAN CYCLE: We are given an undirected network $G = (N, A)$. Is there a Hamiltonian cycle in G?

HAMILTONIAN PATH: We are given a directed network $G = (N, A)$ with two distinguished nodes s and t. Is there a directed path from s to t that visits every node of G exactly once?

REACHABILITY: We are given a directed network $G = (N, A)$. Is there a directed path between two given nodes s and t in G?

PARTITION: We are given a collection of integers $C = (a_1, a_2, \ldots, a_n)$. Is it possible to find a subset $I \subseteq \{1, \ldots, n\}$, such that $\sum_{i \in I} a_i = \frac{1}{2} \sum_{i=1}^{n} a_i$?

We would like to solve decision problems using algorithms (see Section 1.2). An algorithm solves a decision problem Π if it takes an instance $I \in D_\Pi$ as the input and returns the correct answer **yes** or **no** for I. We can now define the class of tractable problems:

Definition 28 *A decision problem Π belongs to the class $\mathcal{P}$ if it can be solved by a polynomial time algorithm.*

Recall that polynomial time algorithms are efficient in the sense that they can solve computational problems in reasonable time even for large instances. Therefore, the class $\mathcal{P}$ contains decision problems which are easy from a computational point of view. Among the problems described at the beginning of this chapter, the problem REACHABILITY belongs to $\mathcal{P}$. In order to check whether there is a directed path between two given nodes of G, a simple search algorithm which runs in $O(m)$ time can be applied (see Appendix A).

Do the remaining problems belong to $\mathcal{P}$? Before we discuss this question, let us make the following observation. All the sample decision problems described at the beginning of this chapter share a common property. Namely, if I is an instance for which the answer is **yes**, then it is possible to give a short proof of this fact. For SAT and 3-SAT, the proof simply involves finding a boolean assignment for which the formula has value 1. For HAMILTONIAN CYCLE (PATH), the proof constructs a subset of arcs, which forms a Hamiltonian cycle (path) in G. Finally, for the PARTITION problem, the proof involves constructing a subset of C whose elements sum to $\frac{1}{2}\sum_{i=1}^{n} a_i$. Note that, in each of these cases, the proof y satisfies the following three properties: (1) y is short, formally its size is polynomially bounded by the size of the instance I, $|y| = O(|I|^k)$ for some fixed k; (2) given instance I and proof y, we can decide in polynomial time whether y is a proof of I; (3) a proof of I exists if and only if the answer to I is **yes**. If a decision problem Π possesses such a property, then we say that it has a *short proof*. We can now introduce the following class of problems:

Definition 29 *A decision problem Π belongs to the class $\mathcal{NP}$ if it has a short proof.*

It is easy to observe that $\mathcal{P} \subseteq \mathcal{NP}$, because for problems in $\mathcal{P}$ we can verify the answer for I in polynomial time without any help from a proof. Observe next that all decision problems in $\mathcal{NP}$ have a trivial brute force algorithm which works as follows: given an instance I, check all possible proofs; if you find a valid proof y for I then the answer is **yes**; otherwise the answer is **no**. Consider, for example, the SAT problem. The brute force algorithm simply checks all binary sequences of length n. If at least one such sequence satisfies the formula, then the answer is **yes** and, otherwise, the answer is **no**. Similarly, we can list all the possible subsets of arcs in G and thus find one which forms a Hamiltonian cycle or path in G. Of course, the brute force algorithm will not run in polynomial time, because the number of potential proofs is exponential in the size of I. For example, there are 2^n possible boolean assignments for n variables and the brute force must check all of them in the worst case, i.e. to be sure that the answer is **no**.

One of the most important questions in computer science is whether all problems in $\mathcal{NP}$ have polynomial time algorithms. So far, no-one has been able to provide an answer. But it is widely accepted that

$$\mathcal{P} \neq \mathcal{NP},$$

which means that the class $\mathcal{NP}$ contains the problems which cannot be solved by fast polynomial algorithms. But we should keep in mind that $\mathcal{P} \neq \mathcal{NP}$ is only a conjecture and no-one has been able to provide a proof to date.

We now try to identify problems in $\mathcal{NP}$ which may be computationally hard (recall that $\mathcal{NP}$ also contains easy problems, such as REACHABILITY). First consider the following example:

Example 1 (dancing problem) We have a set of n boys and a set of n girls. For boy/girl pair (i, j) we know whether i would like to dance with j. We organize a dancing party and the problem is to check whether it is possible to compose n dancing pairs. We can solve this problem using the method shown in Figure 31. We construct a network G whose nodes represent boys and girls. We add arc (i, j) if i wants to dance with j. We add a source node s and link it to nodes representing boys and a sink node t linked to nodes representing girls. We fix the capacities of all the arcs in G to 1. Now it is easy to check that it is possible to compose n pairs if and only if there is a flow of value n from s to t in G.

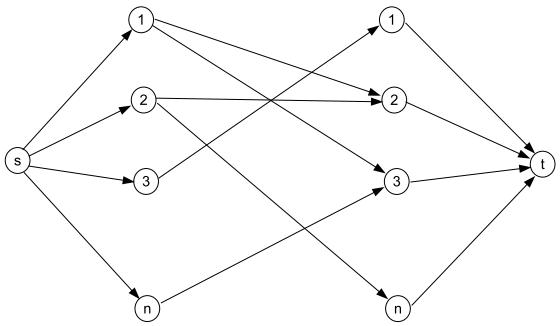


Figure 31: Solving the dancing problem

Example 1 illustrates the idea of the *reduction* of one problem to another. In this particular case, we have transformed the dancing problem into the maximum flow one. We can now use any algorithm for computing the maximum flow and, as a result, we get the solution of the dancing problem. This reduction has an additional important property, namely it transforms the dancing problem into the maximum flow problem in polynomial time. Indeed, the resulting network has $2n^2$ nodes and up to $n^2 + 2n$ arcs. We can thus conclude that the dancing problem is not more difficult than the maximum flow problem or, equivalently, that the maximum flow problem is at least as difficult as the dancing problem.

Definition 30 *A decision problem* Π_1 *is polynomially reducible to a decision problem* Π_2, $\Pi_1 \prec \Pi_2$, *if there is a polynomial time algorithm* A, *which for a given instance* $I \in D_{\Pi_1}$ *returns instance* $A(I) \in D_{\Pi_2}$ *such that the answer to* I *is* ***yes*** *if and only if the answer to* $A(I)$ *is* ***yes***

We can now define the hardest problems in $\mathcal{NP}$:

Definition 31 *A decision problem* Π *is* $\mathcal{NP}$*-complete if*

- $\Pi \in \mathcal{NP}$
- $\Pi' \prec \Pi$ *for all* $\Pi' \in \mathcal{NP}$

Notice that the second condition in the above definition is very strong. It requires that every problem in $\mathcal{NP}$ is polynomially reducible to Π. This means that having a polynomial algorithm for an $\mathcal{NP}$-complete problem Π, we would be able to solve all the problems in $\mathcal{NP}$ in polynomial time. It is not obvious that an $\mathcal{NP}$-complete problem exists. However, this is the case due to the following famous theorem, proven by Cook in 1971:

Theorem 32 (Cook [13]) *The* 3-SAT *problem is* $\mathcal{NP}$*-complete.*

Hence, the 3-SAT problem is a hardest problem in $\mathcal{NP}$. If one could solve this problem in polynomial time, then all problems in $\mathcal{NP}$ would be solvable in polynomial time. Of course, this would imply $\mathcal{P} = \mathcal{NP}$. We can express this in the following equivalent way: *if* $\mathcal{P} \neq \mathcal{NP}$, *then there is no polynomial time algorithm for* 3-SAT.

It turns out that there are a lot of $\mathcal{NP}$-complete problems. In fact, we know thousands of them. For example, we can immediately conclude that SAT is $\mathcal{NP}$-complete. Clearly this problem belongs to $\mathcal{NP}$ and it cannot be easier than 3-SAT. We can prove the $\mathcal{NP}$-completeness of many problems by using the following theorem:

Theorem 33 *If* $\Pi' \in \mathcal{NP}$ *and* $\Pi \prec \Pi'$ *for some* $\mathcal{NP}$*-complete problem* Π, *then* Π' *is* $\mathcal{NP}$*-complete.*

The above theorem says that in order to prove the $\mathcal{NP}$-completeness of some problem $\Pi' \in \mathcal{NP}$ we must find an $\mathcal{NP}$-complete problem Π and reduce Π to Π'. Using this theorem, the $\mathcal{NP}$-completeness of the HAMILTONIAN CYCLE, HAMILTONIAN PATH and PARTITION problems has been proved. The corresponding reductions from 3-SAT are not trivial and can be found in the literature (for example in [40]).

Let us summarize what we have learned so far. If one proves that a decision problem is $\mathcal{NP}$-complete, then this is very strong evidence that this problem is computationally hard. This means that no fast polynomial algorithm exists for this problem if the conjecture $\mathcal{P} \neq \mathcal{NP}$ is true. We now apply the framework presented to optimization problems. There is a natural link between optimization and decision problems. Let Π be an optimization problem. We assume that Π is a minimization problem and the reasoning for maximization problems will

be the same. Consider the following decision version of Π. Given an instance $I \in D_\Pi$ decide whether there is a solution $x \in sol(I)$ such that $f(x) \leq K$, where K is a given number. First notice that this decision problem is not harder than the optimization one. If we can find an optimal solution x^* in polynomial time, then providing an answer to the decision problem is trivial. It is enough to check whether $f(x^*) \leq K$. On the other hand, if the decision problem is hard, say $\mathcal{NP}$-complete, then the optimization problem can not be easier.

Definition 34 *An optimization problem is $\mathcal{NP}$-hard if its decision version is $\mathcal{NP}$-complete.*

We immediately get the following theorem:

Theorem 35 *If $\mathcal{P} \neq \mathcal{NP}$, then no polynomial algorithm exists for any $\mathcal{NP}$ hard problem.*

We now show several examples.

Theorem 36 *The traveling salesperson problem is $\mathcal{NP}$-hard.*

Proof. Consider an instance of the HAMILTONIAN CYCLE problem. So we are given an undirected network $G = (N, A)$, $|N| = n$, and we ask whether G has a Hamiltonian cycle. We construct an instance of the traveling salesperson problem in the following way. The network $G' = (N', A')$ is a complete graph with the same set of nodes as G. If $(i, j) \in A$, then we fix the cost of $(i, j) \in A'$ to 1 and otherwise we fix the cost of $(i, j) \in A'$ to 2. Now it is easy to check that G has a Hamiltonian cycle if and only if there is a salesperson's tour in G' with total cost not greater than n. ■

Observe that the proof shows something more. If all the costs in the traveling salesperson problem are 1 or 2, then the triangle inequality is automatically satisfied. So the special metric version of the problem is also $\mathcal{NP}$-hard.

Theorem 37 *The knapsack problem is $\mathcal{NP}$-hard.*

Proof. Consider an instance of the PARTITION problem. So we have a collection $C = (a_1, a_2, \ldots, a_n)$ of integers and we ask whether there is a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = S$, where $S = \frac{1}{2} \sum_{i=1}^{n} a_i$. Let $\{1, \ldots, n\}$ be the set of items. We fix the values $p_i = a_i$ and weights $w_i = a_i$ for all $i = 1, \ldots, n$. We also fix $W = S$. We now claim that there is a subset of items $X \subseteq \{1, \ldots, n\}$ of total value $f(X) \geq S$ if and only if the answer to the PARTITION problem is **yes**. Suppose that the answer to the PARTITION problem is **yes** and let I be the corresponding subset. Then $X = I$ is a feasible solution such that $f(X) = S$. On the other hand, if X is a feasible solution such that $f(X) \geq S$, then $\sum_{i \in X} p_i = \sum_{i \in X} a_i \geq S$. But X is feasible so $\sum_{i \in X} w_i = \sum_{i \in X} a_i \leq S$. In consequence, $\sum_{i \in X} a_i = S$, $I = X$ and the answer to the PARTITION problem is **yes**. ■

Theorem 38 *The integer (0-1) linear programming problem is $\mathcal{NP}$-hard.*

Proof. This theorem follows directly from the fact that the knapsack problem is $\mathcal{NP}$-hard. Recall that the knapsack problem can be represented as a 0-1 linear programming problem with only one constraint (see Example 5 in Section 1.1). Since this problem is $\mathcal{NP}$-hard, the more general class of 0-1 (integer) linear programming problems is also $\mathcal{NP}$-hard. ■

From the above theorem, it follows that linear integer programming problems are generally much harder to solve than linear programming problems with continuous variables. This does not mean, however, that all discrete optimization problems are $\mathcal{NP}$-hard. An important class of problems, which are solvable in polynomial time (and thus not $\mathcal{NP}$-hard), has been discussed in Chapter 2.

In Section 2.1 we discussed the shortest path problem. We assumed that the problem has no solution when there is a directed cycle of negative cost in G. We can, however, omit the problem of negative cycles by assuming that it is prohibited to visit any node of G more than once. Unfortunately, this assumption makes the shortest path problem NP-hard.

Theorem 39 *If we assume that no node on a path from s to t can be repeated, then the shortest path problem becomes NP-hard.*

Proof. Consider an instance of the HAMILTONIAN PATH problem. So we are given a directed network $G = (N, A)$, $|N| = n$ and we ask whether G has a Hamiltonian path from a given node s to a given node t. Recall that a Hamiltonian path is a directed path that visits every node of G exactly once. Let us now define the cost of each arc in G to be equal to -1. We can now solve the shortest path problem from s to t in G with the additional assumption imposed in the theorem. It is easy to see that G has a Hamiltonian path from s to t if and only if there is a path from s to t of total cost not greater than $-(n-1)$. If G has a Hamiltonian path, then this path has exactly $n-1$ arcs and its cost is $-(n-1)$. On the other hand, if there is no Hamiltonian path in G, then every path from s to t with no repeated nodes, has less than $n-1$ arcs. So its cost is greater than $-(n-1)$. Since the HAMILTONIAN PATH problem is $\mathcal{NP}$-complete, the modified shortest path problem is $\mathcal{NP}$-hard. ■

We now present another interesting result. Consider again the shortest path problem with the following modification. Suppose that the distance d_{ij} and travel time p_{ij} of each arc $(i, j) \in A$ are specified. So we have two parameters associated with each arc of the network. Now the problem consists of computing a shortest path in G under the additional assumption that the travel time for this path is not greater than a given time limit T. We discussed this problem in Section 3.1 (Example 3), where we have called it the constrained shortest path problem. We now prove the following result:

Theorem 40 *The constrained shortest path problem is $\mathcal{NP}$-hard.*

Proof. Consider an instance of the PARTITION problem. So we have a collection of integer numbers $C = (a_1, a_2, \ldots, a_n)$ and ask whether it is possible to find a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \frac{1}{2} \sum_{i=1}^{n} a_i$. The reduction to the constrained shortest path problem is shown in Figure 32.

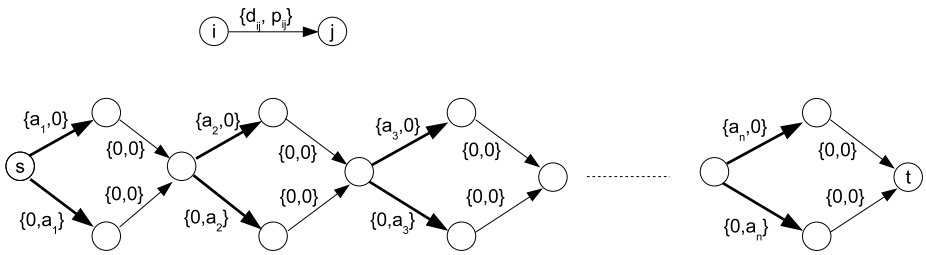


Figure 32: The reduction from the partition problem to the constrained shortest path problem.

For each number a_i we form a network consisting of four nodes and four arcs, whose distances and travel times are shown in Figure 32. We fix $T = \frac{1}{2}\sum_{i=1}^{n} a_i$. It is not difficult to verify that the answer to the PARTITION problem is **yes** if and only if there is a feasible path from s to t in the network whose length is not greater than $\frac{1}{2}\sum_{i=1}^{n} a_i$. If the answer is **yes**, then there is a subset I such that $\sum_{i\in I} a_i = \frac{1}{2}\sum_{i=1}^{n} a_i$. We can build a path from s to t by choosing the upper node for all $i \in I$ and the lower node for all $i \notin I$. It is easy to see that the length and the traveling time of this path is equal to $\frac{1}{2}\sum_{i=1}^{n} a_i$. On the other hand, if the answer to the PARTITION problem is **no**, then for every path p in G, either its length or travel time will be greater than $\frac{1}{2}\sum_{i=1}^{n} a_i$. ■

In Section 3.4 we introduced the concept of a k-approximation algorithm. Recall that such an algorithm runs in polynomial time and returns a solution x such that $f(x) \leq kOPT$, where OPT is the cost of an optimal solution. Even if an optimization problem is $\mathcal{NP}$-hard, there may exist a k-approximation algorithm for it. We have seen several examples in Section 3.4. We now show that for the general traveling salesperson problem it is believed that no k-approximation algorithm exists.

Theorem 41 *If $\mathcal{P} \neq \mathcal{NP}$, then there is no k-approximation algorithm for the traveling salesperson problem for any fixed $k \geq 1$.*

Proof. The proof is similar to that of Theorem 36. Consider an instance $G = (N, A)$, $|N| = n$, of the HAMILTONIAN CYCLE problem. The corresponding instance of the traveling salesperson problem is formed by the complete network $G' = (N', A')$ with the same set of nodes as G. If $(i, j) \in A$, then we fix the cost of $(i, j) \in A'$ to 1 and otherwise we fix the cost of $(i, j) \in A'$ to $nk + 1$. If G has a Hamiltonian cycle, then there is a tour in G' of cost equal to n. On the other hand, if there is no Hamiltonian cycle in G, then all tours have a cost not less than $nk + 1$. Suppose now that we have a polynomial k-approximation algorithm for the traveling salesperson problem. Applying this algorithm to the network G', we obtain a tour π such that $f(\pi) \leq kn$ if G has a Hamiltonian cycle and a tour π such that $f(\pi) \geq kn + 1$ otherwise. In consequence, the k-approximation algorithm would be able to solve the HAMILTONIAN CYCLE problem in polynomial time and $\mathcal{P} = \mathcal{NP}$. ■

Theorem 41 is much stronger than Theorem 36. Not only is the traveling salesperson $\mathcal{NP}$-hard, but also obtaining a suboptimal tour with guaranteed performance is hard. Any polynomial time algorithm for the traveling salesperson problem can return an arbitrarily bad tour for some instances. Notice, however, that the costs in the network G' need not satisfy the triangle inequality and, indeed, the metric version of the problem, where the triangle inequality is satisfied, has a 3/2-approximation algorithm (the Christofides algorithm).

Index