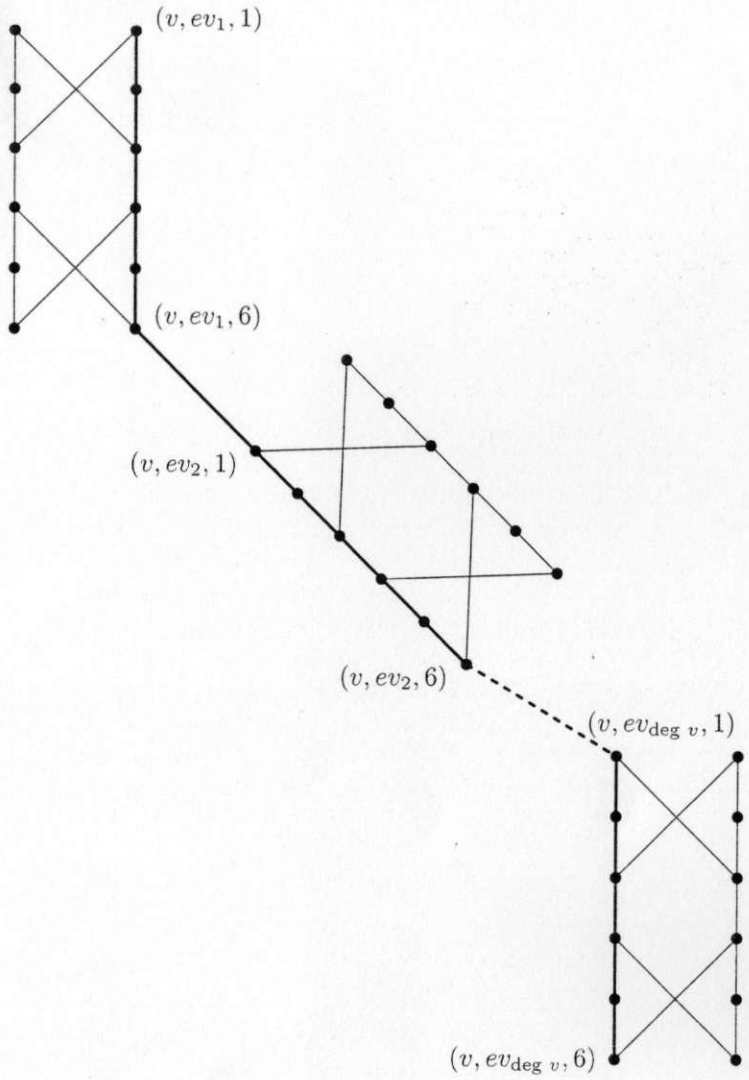

Fig. 2.6.

Moreover, K contains all edges in E'_{v_i} (for $i = 1, \ldots, k$) and the edges

$$\begin{aligned} &\{a_i, (v_i, (ev_i)_1, 1)\} && \text{for } i = 1, \ldots, k; \\ &\{a_{i+1}, (v_i, (ev_i)_{\deg v_i}, 6)\} && \text{for } i = 1, \ldots, k-1; \quad \text{and} \\ &\{a_1, (v_k, (ev_k)_{\deg v_k}, 6)\}. \end{aligned}$$

The reader may check that K is indeed a Hamiltonian cycle for G'. □

3

Shortest Paths

So many paths that wind and wind...
ELLA WHEELER WILCOX

One of the most common applications of graphs in everyday life is representing networks for traffic or for data communication. The schematic map of the German motorway system in the official guide *Autobahn Service*, the railroad or bus lines in some public transportation system, and the network of routes an airline offers are routinely represented by graphs. Therefore it is obviously of great practical interest to study paths in such graphs. In particular, we often look for paths which are good or even best in some respect: sometimes the shortest or the fastest route is required, sometimes we want the cheapest path or the one which is safest – for example, we might want the route where we are least likely to encounter a speed-control installation. Thus we will study shortest paths in graphs and digraphs in this chapter; as we shall see, this is a topic whose interest extends beyond traffic networks.

3.1 Shortest paths

Let $G = (V, E)$ be a graph or a digraph on which a mapping $w : E \to \mathbb{R}$ is defined. We call the pair (G, w) a *network*; the number $w(e)$ is called the *length* of the edge e. Of course, this terminology is not intended to exclude other interpretations such as cost, duration, capacity, weight, or probability; we will encounter several examples later. For instance, in the context of studying spanning trees, we usually interpret $w(e)$ as the weight of the edge e. But in the present chapter the reader should keep the intuitive interpretation of distances in a network of streets in mind. This naturally leads to the following definition. For each walk $W = (e_1, \ldots, e_n)$, the *length* of W is $w(W) := w(e_1) + \ldots + w(e_n)$; of course, W has to be directed for digraphs. The *distance* $d(a, b)$ between two vertices a and b in G is the minimum over all lengths of walks starting at a and ending at b. There are two difficulties with this definition: first, b might not be accessible from a, and second, a minimum might fail to exist. The first problem is solved by defining $d(a, b) = \infty$ if b is not accessible from a. The second problem arises from the possible existence

of cycles of negative length. For example, in the network shown in Figure 3.1, we can find a walk of arbitrary negative length from a to b by using the cycle (x, y, z, x) as often as needed. This problem can be avoided by restricting the definition to trails. Most of the networks we will deal with will not contain any cycles of negative length; then the distance between two vertices is well-defined even if we allow walks in the definition.

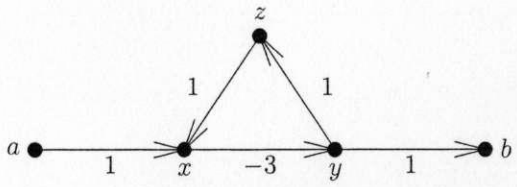

Fig. 3.1. A network

The reader might wonder why negative lengths are allowed at all and whether they occur in practice. The answer is yes, they do occur, as the following example taken from [Law76] shows; this also provides a first example for another interpretation of the length of an edge.

Example 3.1.1. A trading ship travels from port a to port b, where the route (and possible intermediary ports) may be chosen freely. The routes are represented by trails in a digraph G, and the length $w(e)$ of an edge $e = xy$ signifies the profit gained by going from x to y. For some edges, the ship might have to travel empty so that $w(e)$ is negative for these edges: the profit is actually a loss. Replacing w by $-w$ in this network, the shortest path represents the route which yields the largest possible profit.

Clearly, the practical importance of the preceding example is negligible. We will encounter genuinely important applications later when treating flows and circulations, where the existence of cycles of negative length – and finding such cycles – will be an essential tool for determining an optimal circulation.

Thus, we allow negative values for w in general and define distances as explained above. A *shortest path* from a to b then is a trail (directed in the case of digraphs) of length $d(a, b)$ from a to b. If G does not contain any cycles of negative length, we can also talk of shortest walks. Note that always $d(a, a) = 0$, since an empty sum is considered to have value 0, as usual. If we talk of shortest paths and distances in a graph (or a digraph) without giving any explicit length function, we always use the length function which assigns length $w(e) = 1$ to each edge e.

We now give an example for an interpretation of shortest paths which allows us to formulate a problem (which at first glance might seem completely out of place here) as a problem of finding shortest paths in a suitable graph.

Example 3.1.2. In many applications, the length of an edge signifies the probability of its failing – for instance, in networks of telephone lines, or broadcasting systems, in computer networks, or in transportation routes. In all these cases, one is looking for the route having the highest probability for not failing. Let $p(i, j)$ be the probability that edge (i, j) does not fail. Under the – not always realistic – assumption that failings of edges occur independently of each other, $p(e_1) \dots p(e_n)$ gives the probability that the walk $(e_1, \dots, e_n)$ can be used without interruption. We want to maximize this probability over all possible walks with start vertex a and end vertex b. Note first that the maximum of the product of the $p(e)$ is reached if and only if the logarithm of the product, namely $\log p(e_1) + \dots + \log p(e_n)$, is maximal. Moreover, $\log p(e) \leq 0$ for all e, since $p(e) \leq 1$. We now put $w(e) = -\log p(e)$; then $w(e) \geq 0$ for all e, and we have to find a walk from a to b for which $w(e_1) + \dots + w(e_n)$ becomes minimal. Thus our problem is reduced to a shortest path problem. In particular, this technique solves the problem mentioned in our introductory remarks – finding a route where it is least likely that our speed will be controlled by the police – provided that we know for all edges the probability of a speed check.

In principle, a technique for finding shortest paths can also be used to find longest paths: replacing w by $-w$, a longest path with respect to w is just a shortest path with respect to $-w$. However, good algorithms for finding shortest paths are known only for the case where G does not contain any cycles of negative length. In the general case we basically have to look at all possible paths. Note that replacing w by $-w$ in general creates cycles of negative length.

Exercise 3.1.3 (knapsack problem). Consider n given objects, each of which has an associated *weight* a_j and also a *value* c_j, where both the a_j and the c_j are positive integers. We ask for a subset of these objects such that the sum of their weights does not exceed a certain bound b and such that the sum of their values is maximal. Packing a knapsack provides a good example, which explains the terminology used. Reduce this problem to finding a longest path in a suitable network. Hint: Use an acyclic network with a start vertex s, an end vertex t, and $b + 1$ vertices for each object.

3.2 Finite metric spaces

Before looking at algorithms for finding shortest paths, we want to show that there is a connection between the notions of distance and metric space. We recall that a *metric space* is a pair (X, d) consisting of a set X and a mapping $d\colon X^2 \to \mathbb{R}_0^+$ satisfying the following three conditions for all $x, y, z \in X$:

(MS1) $d(x, y) \geq 0$, and $d(x, y) = 0$ if and only if $x = y$;
(MS2) $d(x, y) = d(y, x)$;
(MS3) $d(x, z) \leq d(x, y) + d(y, z)$.

The value $d(x,y)$ is called the *distance* between x and y; the inequality in (MS3) is referred to as the *triangle inequality*. The matrix $D = (d(x,y))_{x,y\in X}$ is called the *distance matrix* of (X,d).

Now consider a network (G,w), where G is a graph and w is a positive valued mapping $w : E \to \mathbb{R}^+$. Note that a walk with start vertex a and end vertex b which has length $d(a,b)$ – where the distance between a and b is defined as in Section 3.1 – is necessarily a path. The following result states that our use of the term *distance* in this context is justified; the simple proof is left to the reader.

Lemma 3.2.1. *Let $G = (V,E)$ be a connected graph with a positive length function w. Then (V,d) is a finite metric space, where the distance function d is defined as in Section 3.1.* □

Lemma 3.2.1 suggests the question whether any finite metric space can be realized by a network. More precisely, let D be the distance matrix of a finite metric space (V,d). Does a graph $G = (V,E)$ with length function w exist such that its distance matrix with respect to w agrees with D? Hakimi and Yau [HaVa64] answered this question as follows.

Proposition 3.2.2. *Any finite metric space can be realized by a network with a positive length function.*

Proof. Let (V,d) be a finite metric space. Choose G to be the complete graph with vertex set V, and let the length function w be the given distance function d. By d' we denote the distance in the network (G,w) as defined in Section 3.1; we have to show $d = w = d'$. Thus let $W = (e_1,\ldots,e_n)$ be a trail with start vertex a and end vertex b. For $n \geq 2$, an iterative application of the triangle inequality yields:

$$w(W) = w(e_1) + \ldots + w(e_n) = d(e_1) + \ldots + d(e_n) \geq d(a,b).$$

As the one edge path $a - b$ has length $d(a,b)$, we are finished. □

Exercise 3.2.3. Find a condition under which a finite metric space can be realized by a graph, that is, by a network all of whose edges have length 1; see [KaCh65].

We have only considered the case where a metric space (V,d) is realized by a network on the vertex set V. More generally, we could allow a network on a graph $G = (V',E)$ with $V \subset V'$, where the distance $d_G(a,b)$ in G for two vertices a,b of V is the same as their distance $d(a,b)$ in the metric space. Such a realization is called *optimal* if the sum of all lengths of edges is minimal among all possible realizations. It is not obvious that such optimal realizations exist, but they do; see [Dre84] and [ImSZ84].

Example 3.2.4. The following simple example shows that the realization given in the proof of Proposition 3.2.2 is not necessarily optimal. Let $d(a,b) = d(b,c) = 4$ and $d(a,c) = 6$. The realization on K_3 has total length 14, whereas there is a realization on four vertices with total length just seven:

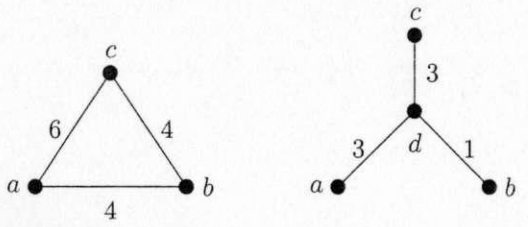

Fig. 3.2. Two realizations of a distance matrix

Realizations of metric spaces by networks have been intensively studied. In particular, the question whether a given metric space can be realized on a tree has sparked considerable interest; such a realization is necessarily optimal [HaVa64]. Bunemann [Bun74] proved that a realization on a tree is possible if and only if the following condition holds for any four vertices x,y,z,t of the given metric space:

$$d(x,y) + d(z,t) \leq \max\,(d(x,z) + d(y,t), d(x,t) + d(y,z)).$$

A different characterization (using ultra-metrics) is due to [Ban90]. We also refer the reader to [Sim88] and [Alt88]. The problem of finding an optimal realization is difficult in general: it is NP-hard [Win88].

3.3 Breadth first search and bipartite graphs

We now turn to examining algorithms for finding shortest paths. All techniques presented here also apply to multigraphs, but this generalization is of little interest: when looking for shortest paths, out of a set of parallel edges we only use the one having smallest length. In this section, we consider a particularly simple special case, namely distances in graphs (where each edge has length 1). The following algorithm was suggested by Moore [Moo59] and is known as *breadth first search*, or, for short, BFS. It is one of the most fundamental methods in algorithmic graph theory.

Algorithm 3.3.1 (BFS). Let G be a graph or digraph given by adjacency lists A_v. Moreover, let s be an arbitrary vertex of G and Q a queue.[1] The vertices of G are labelled with integers $d(v)$ as follows:

[1] Recall that a *queue* is a data structure for which elements are always appended at the end, but removed at the beginning (*first in – first out*). For a discussion of the implementation of queues we refer to [AhHU83] or [CoLR90].

Procedure BFS$(G, s; d)$

```
(1) Q ← ∅; d(s) ← 0;
(2) append s to Q;
(3) while Q ≠ ∅ do
(4)         remove the first vertex v from Q;
(5)         for w ∈ A_v do
(6)                 if d(w) is undefined
(7)                 then d(w) ← d(v) + 1;
(8)                         append w to Q
(9)                 fi
(10)        od
(11) od
```

Theorem 3.3.2. *Algorithm 3.3.1 has complexity $O(|E|)$. At the end of the algorithm, every vertex t of G satisfies*

$$d(s,t) = \begin{cases} d(t) \text{ if } d(t) \text{ is defined,} \\ \infty \quad \text{otherwise.} \end{cases}$$

Proof. Obviously, each edge is examined at most twice by BFS (in the directed case, only once), which yields the assertion about the complexity. Moreover, $d(s,t) = \infty$ if and only if t is not accessible from s, and thus $d(t)$ stays undefined throughout the algorithm. Now let t be a vertex such that $d(s,t) \neq \infty$. Then $d(s,t) \leq d(t)$, since t was reached by a path of length $d(t)$ from s. We show that equality holds by using induction on $d(s,t)$. This is trivial for $d(s,t) = 0$, that is, $s = t$. Now assume $d(s,t) = n+1$ and let $(s, v_1, \ldots, v_n, t)$ be a shortest path from s to t. Then $(s, v_1, \ldots, v_n)$ is a shortest path from s to v_n and, by our induction hypothesis, $d(s, v_n) = n = d(v_n)$. Therefore $d(v_n) < d(t)$, and thus BFS deals with v_n before t during the **while**-loop. On the other hand, G contains the edge $v_n t$ so that BFS certainly reaches t when examining the adjacency list of v_n (if not earlier). This shows $d(t) \leq n+1$ and hence $d(t) = n+1$. □

Corollary 3.3.3. *Let s be a vertex of a graph G. Then G is connected if and only if $d(t)$ is defined for each vertex t at the end of* BFS$(G, s; d)$. □

Note that the statement analogous to Corollary 3.3.3 for directed graphs is not true. If we want to check whether a given digraph is connected, we should apply BFS to the corresponding graph $|G|$. Applying BFS$(G, s; d)$ for each vertex s of a digraph allows us to decide whether G is strongly connected; clearly, this holds if and only if BFS$(G, s; d)$ always reaches all vertices t and assigns values to $d(t)$. However, this method is not very efficient, as it has complexity $O(|V||E|)$. In Chapter 8, we will see a much better technique which has complexity $O(|E|)$.

For an example, let us consider how BFS runs on the digraph G drawn in Figure 3.3. To make the algorithm deterministic, we select the vertices in alphabetical order in step (5) of the BFS. In Figures 3.4 and 3.5, we illustrate the output of BFS both for G and the associated graph $|G|$. To make things clearer, we have drawn the vertices in *levels* according to their distance to s; also, we have omitted all edges leading to vertices already labelled. Thus all we see of $|G|$ is a *spanning tree* , that is, a spanning subgraph of G which is a tree. This kind of tree will be studied more closely in Chapter 4. Note that distances in G and in $|G|$ do not always coincide, as was to be expected. However, we always have $d_G(s,t) \geq d_{|G|}(s,t)$.

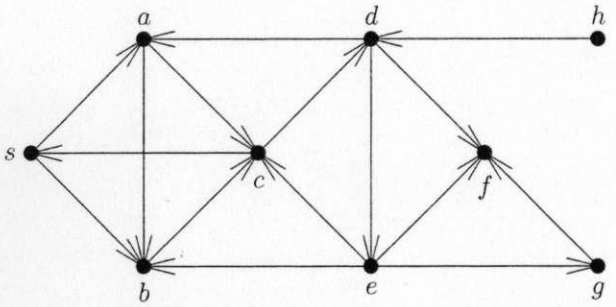

Fig. 3.3. A digraph G

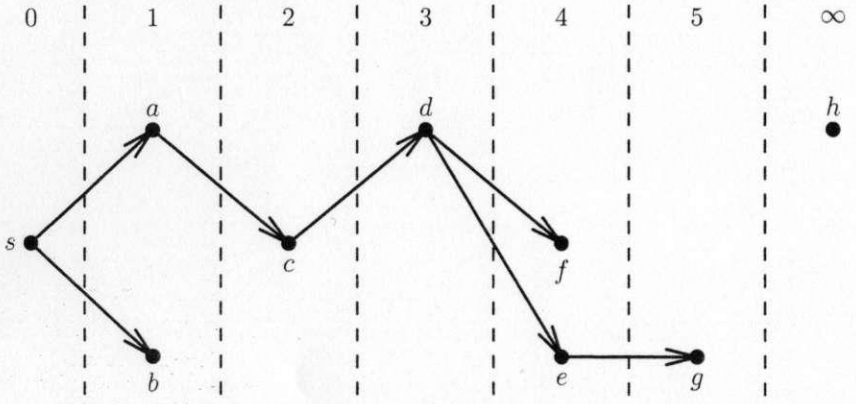

Fig. 3.4. BFS-tree for G

Exercise 3.3.4. Design a BFS-based algorithm COMP(G) which determines the connected components of a graph G.

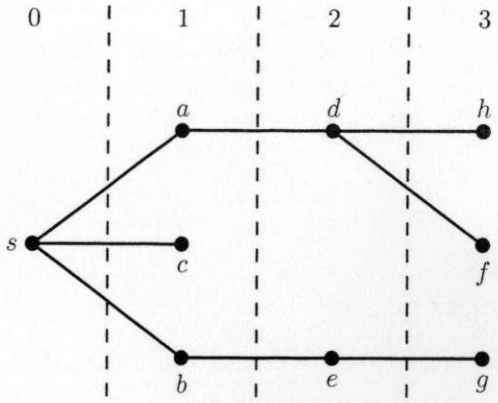

Fig. 3.5. BFS-tree for $|G|$

Next we consider a particularly important class of graphs, namely the bipartite graphs. As we shall see soon, BFS gives an easy way to decide whether or not a graph belongs to this class. Here a graph $G = (V, E)$ is said to be *bipartite* if there is a partition $V = S \dot\cup T$ of its vertex set such that the sets of edges $E|S$ and $E|T$ are empty, that is, each edge of G is incident with one vertex in S and one vertex in T. The following theorem gives a very useful characterization of these graphs.

Theorem 3.3.5. *A graph G is bipartite if and only if it does not contain any cycles of odd length.*

Proof. First suppose that G is bipartite and let $V = S \dot\cup T$ be the corresponding partition of its vertex set. Consider an arbitrary closed trail in G, say

$$C: v_1 - v_2 - \ldots - v_n - v_1.$$

We may assume $v_1 \in S$. Then

$$v_2 \in T, \quad v_3 \in S, \quad v_4 \in T, \quad \ldots \quad , \quad v_n \in T, \quad v_1 \in S,$$

as there are no edges within S or T. Hence n must be even.

Conversely, suppose that G does not contain any cycles of odd length. We may assume that G is connected. Choose some vertex x_0. Let S be the set of all vertices x having even distance $d(x, x_0)$ from x_0, and let T be the complement of S. Now suppose that there is an edge xy in G with $x, y \in S$. Let W_x and W_y be shortest paths from x_0 to x and y, respectively. By our definition of S, both these paths have even length. Let us denote the last common vertex of W_x and W_y by z (traversing both paths starting at x_0), and call their final parts (leading from z to x and y, respectively) W'_x and W'_y. Then it is easily seen that

$$x \overset{W'_x}{\text{---}} z \overset{W'_y}{\text{---}} y \overset{xy}{\text{---}} x$$

is a cycle of odd length in G, a contradiction. Similarly, G cannot contain an edge xy with $x, y \in T$. Hence $S \dot\cup T$ is a partition of V such that there are no edges within S or T, and G is bipartite. □

The proof of Theorem 3.3.5 shows how we may use the distances $d(s, t)$ in G (from a given start vertex s) for finding an appropriate partition of the vertex set of a given bipartite graph G. These distances can be determined using BFS; of course, we should modify Algorithm 3.3.1 in such a way that it detects cycles of odd length, in case G is not bipartite. This is actually rather simple: when BFS examines an edge e for the first time, a cycle of odd length containing e exists if and only if e has both its vertices in the same level. This gives us the desired criterion for checking whether G is bipartite or not; moreover, if G is bipartite, the part of G determined by s consists of those vertices which have even distance from s. These observations lead to the following algorithm and the subsequent theorem.

Algorithm 3.3.6. Let G be a connected graph and s a vertex of G.

Procedure BIPART($G, s; S, T$,bip)

```
(1) Q ← ∅, d(s) ← 0, bip ← true, S ← ∅;
(2) append s to Q;
(3) while Q ≠ ∅ and bip = true do
(4)         remove the first vertex v of Q;
(5)         for w ∈ A_v do
(6)                 if d(w) is undefined
(7)                 then d(w) ← d(v) + 1; append w to Q
(8)                 else if d(v) = d(w) then bip ← false fi
(9)                 fi
(10)        od
(11) od;
(12) if bip = true then for v ∈ V do
(13)        if d(v) ≡ 0 (mod 2) then S ← S ∪ {v} fi
(14)        od;
(15)        T ← V \ S
(16) fi
```

Theorem 3.3.7. *Algorithm 3.3.6 checks with complexity $O(|E|)$ whether a given connected graph G is bipartite; if this is the case, it also determines the corresponding partition of the vertex set.* □

Exercise 3.3.8. Describe a BFS-based algorithm which finds with complexity $O(|V||E|)$ a shortest cycle in – and thus the girth of – a given graph G.

The problem of finding a shortest cycle was extensively studied by Itai and Rodeh [ItRo78] who also treated the analogous problem for directed graphs. The best known algorithm has a complexity of $O(|V|^2)$; see [YuZw97]. BFS can also be used to find a shortest cycle of even or odd length, respectively; see [Mon83].

3.4 Bellman's equations and acyclic digraphs

We now turn to the problem of determining shortest paths in a general network; actually, all known algorithms for this problem even find a shortest path from the start vertex s to each vertex t which is accessible from s. Choosing t in a special way does not decrease the complexity of the algorithms. As agreed in Section 3.1, we always assume that G does not contain any cycles of negative length. Moreover, we assume from now on that G is a directed graph so that all paths used are also directed.[2]

Without loss of generality, we may assume that G has vertex set $V = \{1, \ldots, n\}$. Let us write $w_{ij} := w(ij)$ if G contains the edge ij, and $w_{ij} = \infty$ otherwise. Furthermore, let u_i denote the distance $d(s, i)$, where s is the start vertex; in most cases, we will simply take $s = 1$. Now any shortest path from s to i has to contain a final edge ki, and deleting this edge yields a shortest path from s to k. Hence the distances u_i have to satisfy the following system of equations due to Bellman [Bel58], where we assume for the sake of simplicity that $s = 1$.

Proposition 3.4.1 (Bellman's equations). *Let $s = 1$. Then*

$$\text{(B)} \quad u_1 = 0 \quad \textit{and} \quad u_i = \min\{u_k + w_{ki} : i \neq k\} \quad \textit{for } i = 2, \ldots, n. \qquad \square$$

We will now show that the system of equations (B) has a unique solution – namely the distances u_i in G – provided that G contains only cycles of positive length and that each vertex is accessible from 1. To this purpose, let u_i $(i = 1, \ldots, n)$ be any solution of (B) and choose some vertex $j \neq 1$. We want to construct a path of length u_j from 1 to j. To do so, we first choose some edge kj with $u_j = u_k + w_{kj}$, then an edge ik with $u_k = u_i + w_{ik}$, etc. Let us show that this construction cannot yield a cycle. Suppose, to the contrary, we were to get a cycle, say

$$C: \; v_1 \;\text{—}\; v_2 \;\text{—}\; \ldots \;\text{—}\; v_m \;\text{—}\; v_1.$$

Then we would have the following equations which imply $w(C) = 0$, a contradiction to our assumption that G contains cycles of positive length only:

[2] For nonnegative length functions, the undirected case can be treated by considering the complete orientation $\vec{G}$ instead of G. If we want to allow edges of negative length, we need a construction which is considerably more involved, see Section 14.6.

$$\begin{aligned} u_{v_1} &= u_{v_m} + w_{v_m v_1} \\ &= u_{v_{m-1}} + w_{v_{m-1} v_m} + w_{v_m v_1} \\ &= \ldots \\ &= u_{v_1} + w_{v_1 v_2} + \ldots + w_{v_m v_1}. \end{aligned}$$

Thus our construction can only stop at the special vertex 1, yielding a path from 1 to j. Also, for each vertex i occurring on this path, the part of the path leading to i has length u_i. Continuing in this way for all other vertices not yet occurring in the path(s) constructed so far – where we construct a new path backward only until we reach some vertex on one of the paths constructed earlier – we obtain a directed spanning tree with root 1. In particular, we may apply this technique to the distances in G, since they satisfy Bellman's equations. This proves the following result.

Theorem 3.4.2. *If 1 is a root of G and if all cycles of G have positive length with respect to w, then G contains a spanning arborescence with root 1 for which the path from 1 to any other vertex in G always is a shortest path.* $\square$

A spanning arborescence with root s is usually called a *shortest path tree* for the network (G, w) if, for each vertex v, the path from s to v in T has length $d(s, v)$; we will often use the shorter term *SP-tree*. Thus Theorem 3.4.2 shows that an SP-tree exists provided that all cycles of G have positive length with respect to w.

Now let $u_1, \ldots, u_n$ be the distances in G, and let $u'_1, \ldots, u'_n$ be a further solution of (B). Suppose $u_j \neq u'_j$ for some j. The above construction shows that u'_j is the length of some – not necessarily shortest – path from 1 to j. As $u_j = d(1, j)$, this means $u'_j > u_j$. Let kj be the last edge in a path of length u'_j from 1 to j. By induction, we may assume $u_k = u'_k$. But then $u'_j > u'_k + w_{kj}$ which contradicts (B). Hence $u_j = u'_j$ for all $j = 1, \ldots, n$, proving the desired uniqueness result.

Theorem 3.4.3. *If 1 is a root of G and if all cycles of G have positive length with respect to w, then Bellman's equations have a unique solution, namely the distances $u_j = d(1, j)$.* $\square$

In view of the preceding results, we have to solve the system of equations (B). We begin with the simplest possible case, where G is an acyclic digraph. As we saw in Section 2.6, we can find a topological sorting of G in $O(|E|)$ steps. After having executed TOPSORT, let us replace each vertex v by its number topnr(v). Then every edge ij in G satisfies $i < j$, and we may simplify Bellman's equations as follows:

$$u_1 = 0 \quad \text{and} \quad u_i = \min\{u_k + w_{ki} : k = 1, \ldots, i-1\} \quad \text{for } i = 2, \ldots, n.$$

Obviously, this system of equations can be solved recursively in $O(|E|)$ steps if we use backward adjacency lists, where each list contains the edges with a common head. This proves the following result.

Theorem 3.4.4. *Let N be a network on an acyclic digraph G with root s. Then one can construct a shortest path tree with root s in $O(|E|)$ steps.* □

Mehlhorn and Schmidt [MeSc86] found a larger class of graphs (containing the acyclic digraphs) for which with complexity $O(|E|)$ it is possible to determine the distances with respect to a given vertex.

Exercise 3.4.5. Show that, under the same conditions as in Theorem 3.4.4, we can also with complexity $O(|E|)$ determine a system of longest paths from s to all other vertices. Does this yield an efficient algorithm for the knapsack problem of Exercise 3.1.3? What happens if we drop the condition that the graph should be acyclic?

Let us return to SP-trees again. We want to prove the following important strengthening of Theorem 3.4.2:

Theorem 3.4.6. *Let G be a digraph with root s, and let $w: E \to \mathbb{R}$ be a length function on G. If the network (G, w) does not contain any directed cycles of negative length, then there exists an SP-tree with root s for (G, w).*

Proof. Let $v \neq s$ be an arbitrary vertex of G. By hypothesis, v is accessible from s; let W be a trail of shortest length $d(s, v)$ from s to v. As (G, w) does not contain any directed cycles of negative length, W is even a shortest walk from s to v. Now let u be the last vertex on W before v, so that the final edge of W is $e = uv$. Then $W \setminus e$ has to be a shortest trail from s to u: if W' were a trail from s to u shorter than $W \setminus e$, then $W' - e$ would be a shorter walk from s to v than W. Hence

$$d(s, v) = d(s, u) + w(uv). \tag{3.1}$$

Thus we may, for each vertex $v \neq s$, choose an edge $e = e_v = uv$ satisfying condition (3.1). This gives $|V| - 1$ edges which together form a spanning arborescence T of G with root s.[3] It is now easy to see that the unique path P_t from s to v in T always has length $d(s, v)$: this follows by induction on the number of edges contained in P_t, using the fact that all edges of T satisfy condition (3.1). Thus T is the desired SP-tree for (G, w). □

Exercise 3.4.7. Show that the condition that no cycles of negative length exist is necessary for proving Theorem 3.4.6: if (G, w) contains a directed cycle of negative length, then thereis no SP-tree for (G, w).

Exercise 3.4.8. Let T be a spanning arborescence with root s in a network (G, w) which does not contain any directed cycles of negative length. Show that T is an SP-tree if and only if the following condition holds for each edge $e = uv$ of G:

$$d_T(s, v) \leq d_T(s, u) + w(uv), \tag{3.2}$$

where $d_T(s, u)$ denotes the distance from s to u in the network $(T, w|T)$.

[3] The reader should check this for himself as an exercise; a formal proof can be found in Lemma 4.8.1.

3.5 An application: Scheduling projects

We saw in Exercise 3.4.5 that it is easy to find longest paths in an acyclic digraph. We will use this fact to solve a rather simple instance of the problem of making up a schedule for a project. If we want to carry out a complex project such as, for example, building a dam, a shopping center or an airplane – the various tasks ought to be well coordinated to avoid loss of time and money. This is the goal of network planning, which is, according to [Mue73] "the tool from operations research used most." [Ta92] states that these techniques 'enjoy tremendous popularity among practitioners in the field'. We restrict ourselves to the simple case where we have restrictions on the chronological sequence of the tasks only: there are some tasks which we cannot begin before certain others are finished. We are interested in the shortest possible time the project takes, and would like to know the points of time when each of the tasks should be started. Two very similar methods to solve this problem, namely the *critical path method (CPM)* and the *project evaluation and review technique (PERT)* were developed between 1956 and 1958 by two different groups, cf. [Ta92] and [Mue73]. CPM was introduced by E. I. du Pont de Nemours & Company to help schedule construction projects, and PERT was developed by Remington Rand for the U.S. Navy to help schedule the research and development activities for the Polaris missile program. CPM-PERT is based on determining longest paths in an acyclic digraph. We shall use a formulation where the activities in the project are represented by vertices; alternatively, one could also represent them by arcs, cf. [Ta92].

First, we assign a vertex $i \in \{1, \ldots, N\}$ of a digraph G to each of the N tasks of our project. We let ij be an edge of G if and only if task i has to be finished before beginning task j. The edge ij then has length $w_{ij} = d_i$ equal to the time task i takes. Note that G has to be acyclic, because otherwise the tasks in a cycle in G could never be started. As we have seen in Lemma 2.6.2, G contains at least one vertex v with $d_{\text{in}}(v) = 0$ and, analogously, at least one vertex w with $d_{\text{out}}(w) = 0$. We introduce a new vertex s (the start of the project) and add edges sv for all vertices v with $d_{\text{in}}(v) = 0$; similarly, we introduce a new vertex z (the end of the project) and add edges wz for all vertices w with $d_{\text{out}}(w) = 0$. All the new edges sv have length 0, whereas the edges wz are given length d_w. In this way we get a larger digraph H with root s; by Theorem 2.6.3, we may assume H to be topologically sorted.

Now we denote the earliest possible point of time at which we could start task i by t_i. As all the tasks immediately preceding i have to be finished before, we get the following system of equations:

$$\text{(CPM)} \qquad t_s = 0 \quad \text{and} \quad t_i = \max\{t_k + w_{ki} : ki \text{ an edge in } H\}.$$

This system of equations is analogous to Bellman's equations and describes the longest paths in H, compare Exercise 3.4.5. As in Theorem 3.4.3, (CPM) has a unique solution which again is easy to calculate recursively, since H is topologically sorted and thus only contains edges ij with $i < j$. The minimal

amount of time the project takes is the length $T = t_z$ of a longest path from s to z. If the project is actually to be finished at time T, the latest point of time T_i where we can still start task i is given recursively by

$$T_z = T \quad \text{and} \quad T_i = \min\{T_j - w_{ij} : ij \text{ an edge in } H\}.$$

Thus $T_z - T_i$ is the length of a longest path from i to z. Of course, we should get $T_s = 0$, which is useful for checking our calculations. The difference $m_i = T_i - t_i$ between the earliest point of time and the latest point of time for beginning task i is called *float* or *slack*. All tasks i having float $m_i = 0$ are called *critical*, because they have to be started exactly at the point of time $T_i = t_i$, as otherwise the whole project would be delayed. Note that each longest path from s to z contains critical tasks only; for that reason each such path is called a *critical path* for H. In general, there will be more than one critical path.

In practice, H will not contain all edges ij for which i has to be finished before j, but only those edges for which i is an immediate predecessor of j so that there are no intermediate tasks between i and j. As an example, let us consider a simplified schedule for building a house. First, we need a list of the tasks, the amount of time they take, and which tasks have to be finished before which other tasks; this information can be found in Table 3.2. The corresponding digraph is shown in Figure 3.6. We have drawn the edges as undirected edges to make the figure somewhat simpler: all edges are to be considered as directed from left to right.

The way the digraph is drawn in Figure 3.6, it is not necessary to state a topological sorting of the vertices explicitly; see Exercise 3.5.2. Using (CPM), we calculate consecutively

$$\begin{aligned} &t_s = 0, \quad t_1 = 0, \quad t_2 = 0, \quad t_3 = 3, \quad t_4 = 5, \quad t_5 = 7, \quad t_8 = 7, \\ &t_6 = 14, \quad t_{11} = 14, \quad t_{13} = 17, \quad t_7 = 17, \quad t_9 = 18, \quad t_{10} = 18, \\ &t_{12} = 20, \quad t_{14} = 22, \quad t_{15} = 25, \quad t_{16} = 28, \quad T = t_z = 33. \end{aligned}$$

Similarly, we compute the T_i and the floats m_i:

$$\begin{aligned} &T_z = 33, \quad m_z = 0; \quad T_{16} = 28, \quad m_{16} = 0; \quad T_{15} = 25, \quad m_{15} = 0; \\ &T_{12} = 29, \quad m_{12} = 9; \quad T_{14} = 22, \quad m_{14} = 0; \quad T_9 = 27, \quad m_9 = 9; \\ &T_{10} = 21, \quad m_{10} = 3; \quad T_7 = 20, \quad m_7 = 3; \quad T_{13} = 17, \quad m_{13} = 0; \\ &T_6 = 17, \quad m_6 = 3; \quad T_{11} = 14, \quad m_{11} = 0; \quad T_5 = 7, \quad m_5 = 0; \\ &T_8 = 18, \quad m_8 = 11; \quad T_4 = 5, \quad m_4 = 0; \quad T_3 = 3, \quad m_3 = 0; \\ &T_1 = 0, \quad m_1 = 0; \quad T_2 = 1, \quad m_2 = 1; \quad T_s = 0, \quad m_s = 0. \end{aligned}$$

Thus the critical tasks are $s, 1, 3, 4, 5, 11, 13, 14, 15, 16, z$, and they form (in this order) the critical path, which is unique for this example.

Table 3.1. Project of building a house

Vertex	Task	Amount of time	Preceding tasks
1	prepare the building site	3	–
2	deliver the building material	2	–
3	dig the foundation-hole	2	1,2
4	build the foundation	2	3
5	build the walls	7	4
6	build the roof supports	3	5
7	cover the roof	1	6
8	connect the water pipes to the house	3	4
9	plasterwork outside	2	7,8
10	install the windows	1	7,8
11	put in the ceilings	3	5
12	lay out the garden	4	9,10
13	install inside plumbing	5	11
14	put insulation on the walls	3	10,13
15	paint the walls	3	14
16	move	5	15

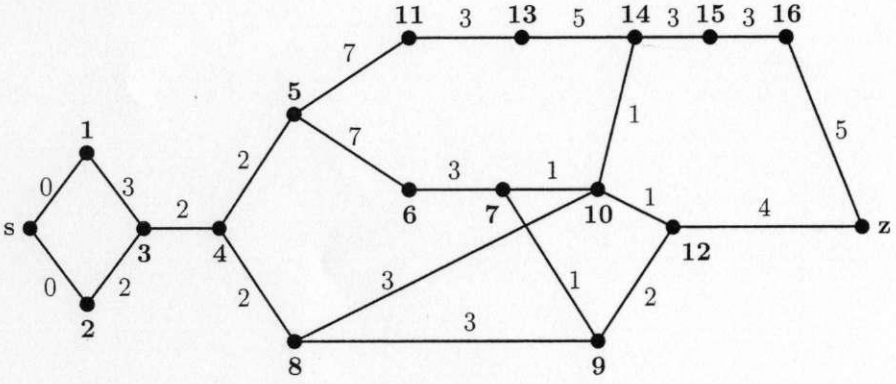

Fig. 3.6. Digraph for the project of building a house

Further information on project scheduling can be found in the books [Ta92] and [Mue73], and in the references given there. Of course, there is much more to scheduling than the simple method we considered. In practice there are often further constraints that have to be satisfied, such as scarce resources like limited amounts of machinery or restricted numbers of workers at a given point of time. For a good general overview of scheduling, the reader is referred to [LaLRS93]. We close this section with a couple of exercises; the first of these is taken from [Mue73].

Exercise 3.5.1. A factory wants to replace an old production facility by a new one; the necessary tasks are listed in the table below. Draw the corresponding network and determine the values t_i, T_i, and m_i.

Table 3.2. Project of building a new production facility

Vertex	Task	Amount of time	Preceding tasks
1	ask for offers, compare and order	25	–
2	take apart the old facility	8	–
3	remove the old foundation	5	2
4	plan the new foundation	9	1
5	term of delivery for the new facility	21	1
6	build the new foundation	9	3,4
7	install the new facility	6	5,6
8	train the staff	15	1
9	install electrical connections	2	7
10	test run	1	8,9
11	acceptance test and celebration	2	10

Exercise 3.5.2. Let G be an acyclic digraph with root s. The *rank* $r(v)$ of a vertex v is the maximal length of a directed path from s to v. Use the methods introduced in this chapter to find an algorithm which determines the rank function.

Exercise 3.5.3. Let G be an acyclic digraph with root s, given by adjacency lists A_v. Show that the following algorithm computes the rank function on G, and determine its complexity:

Procedure RANK$(G, s; r)$

(1) create a list S_0, whose only element is s;
(2) $r(s) \leftarrow 0$; $k \leftarrow 0$;
(3) **for** $v \in V$ **do** $d(v) \leftarrow d_{in}(v)$ **od**;
(4) **while** $S_k \neq \emptyset$ **do**
(5) create a new list S_{k+1};
(6) **for** $v \in S_k$ **do**
(7) **for** $w \in A_v$ **do**
(8) **if** $d(w) = 1$
(9) **then** append w to S_{k+1}; $r(w) \leftarrow k+1$; $p(w) \leftarrow v$
(10) **fi**;
(11) $d(w) \leftarrow d(w) - 1$
(12) **od**
(13) **od**;
(14) $k \leftarrow k+1$
(15) **od**

How can we determine $d(w)$? How can a longest path from s to v be found? Can RANK be used to find a topological sorting of G?

3.6 The algorithm of Dijkstra

In this section, we consider networks having all lengths nonnegative. In this case Bellman's equations can be solved by the algorithm of Dijkstra [Dij59], which is probably the most popular algorithm for finding shortest paths.

Algorithm 3.6.1. Let (G, w) be a network, where G is a graph or a digraph and all lengths $w(e)$ are nonnegative. The adjacency list of a vertex v is denoted by A_v. We want to calculate the distances with respect to a vertex s.

Procedure DIJKSTRA$(G, w, s; d)$

(1) $d(s) \leftarrow 0$, $T \leftarrow V$;
(2) **for** $v \in V \setminus \{s\}$ **do** $d(v) \leftarrow \infty$ **od**;
(3) **while** $T \neq \emptyset$ **do**
(4) find some $u \in T$ such that $d(u)$ is minimal;
(5) $T \leftarrow T \setminus \{u\}$;
(6) **for** $v \in T \cap A_u$ **do** $d(v) \leftarrow \min(d(v), d(u) + w_{uv})$ **od**
(7) **od**

Theorem 3.6.2. *Algorithm 3.6.1 determines with complexity $O(|V|^2)$ the distances with respect to some vertex s in (G, w). More precisely, at the end of the algorithm*

$$d(s,t) = d(t) \quad \textit{for each vertex } t.$$

Proof. Obviously, $d(t) = \infty$ if and only if t is not accessible from s. Now assume $d(t) \neq \infty$. Then $d(s,t) \leq d(t)$, as the algorithm reaches t via a directed path of length $d(t)$ from s to t. We will show the converse inequality $d(t) \leq d(s,t)$ by using induction on the order in which vertices are removed from T. The first vertex removed is s; trivially $d(s) = 0 = d(s,s)$. Now suppose that the inequality is true for all vertices t that were removed from T before u. We may assume that $d(u)$ is finite. Moreover, let

$$s = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} v_n = u$$

be a shortest path from s to u. Then

$$d(s, v_h) = \sum_{j=1}^{h} w(e_j) \qquad \text{for } h = 0, \dots, n.$$

Choose i as the maximal index such that v_i was removed from T before u. By the induction hypothesis,

$$d(s, v_i) = d(v_i) = \sum_{j=1}^{i} w(e_j).$$

Let us consider the iteration where v_i is removed from T in the **while** loop. As v_{i+1} is adjacent to v_i, the inequality $d(v_{i+1}) \leq d(v_i) + w(e_{i+1})$ is established during this iteration. But $d(v_{i+1})$ cannot be increased again in the subsequent iterations and, hence, this inequality still holds when u is removed. Thus

$$d(v_{i+1}) \leq d(v_i) + w(e_{i+1}) = d(s, v_i) + w(e_{i+1}) = d(s, v_{i+1}) \leq d(s, u). \quad (3.3)$$

Suppose first $v_{i+1} \neq u$, that is, $i \neq n-1$. By equation (3.3), $d(s,u) < d(u)$ would imply $d(v_{i+1}) < d(u)$; but then v_{i+1} would have been removed from T before u in view of the selection rule in step (4), contradicting the fact that we chose i to be maximal. Hence indeed $d(u) \leq d(s,u)$, as asserted. Finally, for $u = v_{i+1}$, the desired inequality follows directly from equation (3.3). This establishes the correctness of Dijkstra's algorithm. For the complexity, note that in step (4) the minimum of the $d(v)$ has to be calculated (for $v \in T$), which can be done with $|T|-1$ comparisons. In the beginning of the algorithm, $|T| = |V|$, and then $|T|$ is decreased by 1 with each iteration. Thus we need $O(|V|^2)$ steps altogether for the execution of (4). It is easy to see that all other operations can also be done in $O(|V|^2)$ steps. □

We remark that the algorithm of Dijkstra might not work if there are negative weights in the network, even if no cycles of negative length exist. Note that the estimate in equation (3.3) does not hold any more if $w(e_{i+1}) < 0$. An algorithm which works also for negative weights can be found in Exercise 3.6.9.

Exercise 3.6.3. Modify Dijkstra's algorithm in such a way that it actually gives a shortest path from s to t, not just the distance $d(s,t)$. If s is a root of G, construct an SP-tree for (G, w).

Example 3.6.4. Consider the network given in Figure 3.7 with vertex set $V = \{1, \ldots, 8\}$. With $s = 1$, Algorithm 3.6.1 is executed as follows, where the final values for d is indicated in bold face:

start values: $\mathbf{d(1) = 0}$, $d(i) = \infty$ for $i = 2, \ldots, 8$, $T = V$.
Iteration I: $u = 1$, $T = \{2, \ldots, 8\}$, $d(2) = 28$, $d(3) = 2$, $\mathbf{d(5) = 1}$;
Iteration II: $u = 5$, $T = \{2, 3, 4, 6, 7, 8\}$, $d(2) = 9$, $\mathbf{d(3) = 2}$, $d(6) = 27$;
Iteration III: $u = 3$, $T = \{2, 4, 6, 7, 8\}$, $\mathbf{d(2) = 9}$, $d(6) = 26$, $d(8) = 29$;
Iteration IV: $u = 2$, $T = \{4, 6, 7, 8\}$, $\mathbf{d(4) = 18}$, $d(6) = 19$;
Iteration V: $u = 4$, $T = \{6, 7, 8\}$, $\mathbf{d(6) = 19}$, $d(7) = 26$, $d(8) = 25$;
Iteration VI: $u = 6$, $T = \{7, 8\}$, $\mathbf{d(8) = 20}$;
Iteration VII: $u = 8$, $T = \{7\}$, $\mathbf{d(7) = 26}$;
Iteration VIII: $u = 7$, $T = \emptyset$.

Exercise 3.6.5. Calculate the distance s with respect to $s = 1$ for the underlying undirected network.

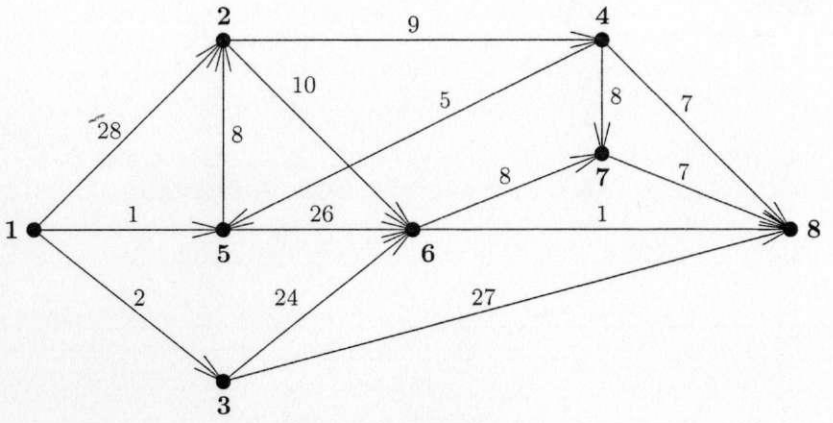

Fig. 3.7. A network

Let us return to the complexity of Dijkstra's algorithm. Initializing the variables in (1) and (2) takes $O(|V|)$ steps. During the entire **while** loop, each edge $e = uv$ is considered exactly once, namely during the iteration where u is removed from T. Thus step (6) contributes only $O(|E|)$ to the complexity of the algorithm, which is – at least for sparse graphs – much better than $O(|V|^2)$. Therefore it makes sense to try to reduce the number of comparisons in step (4) by using an appropriate data structure.

Recall that a *priority queue* (sometimes also called a *heap*) is a data structure consisting of a number of elements each of which is associated with a real number, its *priority*. Permissible operations include inserting elements according to their priority as well as determining and removing the element with the smallest priority; the latter operation is usually referred to as DELETEMIN. As shown in computer science, a priority queue with n elements can be implemented in such a way that each of these two operations can be executed with complexity $O(\log n)$; we need a refinement of this standard implementation which enables us also to remove a given element or reduce its priority with the same complexity $O(\log n)$. We do not go into any details here but refer the reader to [AhHU83] or [CoLR90]. Using these results, we put the vertex set of our digraph into a priority queue T in Dijkstra's algorithm, with d as the priority function. This leads to the following modified algorithm.

Algorithm 3.6.6. Let (G, w) be a given network, where G is a graph or a digraph and all lengths $w(e)$ are nonnegative. We denote the adjacency list of v by A_v. Moreover, let T be a priority queue with priority function d. The algorithm calculates the distances with respect to a vertex s.

Procedure DIJKSTRAPQ$(G, w, s; d)$.

(1) $T \leftarrow \{s\}$, $d(s) \leftarrow 0$;

```
(2) for s ∈ V \ {s} do d(v) ← ∞ od;
(3) while T ≠ ∅ do
(4)         u := min T;
(5)         DELETEMIN (T);
(6)         for v ∈ A_u do
(7)                 if d(v) = ∞
(8)                 then d(v) ← d(u) − w_uv;
(9)                         insert v with priority d(v) into T
(10)                else if d(u) + w_uv < d(v)
(11)                        then change the priority of v to d(v) ← d(u) + w_uv
(12)                        fi
(13)                fi
(14)        od
(15) od
```

As noted before, each of the operations during the **while** loop can be performed in $O(\log |V|)$ steps, and altogether we need at most $O(|E|) + O(|V|)$ such operations. If G is connected, this gives the following result.

Theorem 3.6.7. *Let (G, w) be a connected network, where w is nonnegative. Then Algorithm 3.6.6 (the modified algorithm of Dijkstra) has complexity $O(|E| \log |V|)$.* □

The discussion above provides a nice example for the fact that sometimes we can decrease the complexity of a graph theoretical algorithm by selecting more appropriate (which usually means more complex) data structures. But this is not a one-way road: conversely, graph theory is a most important tool for implementing data structures. For example, priority queues are usually implemented using a special types of trees (for instance, so-called 2-3-trees). A nice treatment of the close interplay between algorithms from graph theory and data structures may be found in [Tar83].

Exercise 3.6.8. Let s be a vertex of a planar network with a nonnegative length function. What complexity does the calculation of the distances with respect to s have?

Using even more involved data structures, we can further improve the results of Theorem 3.6.7 and Exercise 3.6.8. Implementing a priority queue appropriately (for instance, as a *Fibonacci Heap*), inserting an element or reducing the priority of a given element can be done in $O(1)$ steps; DELETEMIN still requires $O(\log n)$ steps. Thus one may reduce the complexity of Algorithm 3.6.6 to $O(|E| + |V| \log |V|)$; see [FrTa87]. The best theoretical bound known at present is $O(|E| + (|V| \log |V|)/(\log \log |V|))$; see [FrWi94]. This algorithm, however, is of no practical interest as the constants hidden in the big-O notation are too large. If all lengths are relatively small (say, bounded by a constant C), one may achieve a complexity of $O(|E| + |V|(\log C)^{1/2})$; see [AhMOT90]. For the planar case, there is an algorithm with complexity $O(|V|(\log |V|)^{1/2})$; see [Fre87]. A short but nice discussion of various algorithmic approaches of practical interest is in [Ber93]. More information about practical aspects may be found in [GaPa88] and [HuDi88].

To end this section, we present an algorithm which can also treat instances where negative lengths occur, as long as no cycles of negative length exist. This is due to Ford [For56] and Bellman [Bel58].

Exercise 3.6.9. Let (G, w) be a network without cycles of negative length. Show that the following algorithm calculates the distances with respect to a given vertex s and determine its complexity:

Procedure BELLFORD$(G, w, s; d)$

```
(1) d(s) ← 0;
(2) for v ∈ V \ {s} do d(v) ← ∞ od;
(3) repeat
(4)         for v ∈ V do d'(v) ← d(v) od;
(5)         for v ∈ V do d(v) ← min (d'(v), min {d'(u) + w_uv : uv ∈ E}) od
(6) until d(v) = d'(v) for all v ∈ V.
```

Apply this algorithm to Example 3.6.4, treating the vertices in the order $1, \ldots, 8$.

3.7 An application: Train schedules

In this section, we discuss a practical problem which can be solved using the algorithm of Dijkstra, namely finding optimal connections in a public transportation system.[4] Such a system consists of several lines (of trains or buses) which are served at regular intervals. Typical examples are the German Intercity network or the American Greyhound bus lines. If someone wants to use such a system to get from one point to another in the network, it may be necessary to change lines a couple of times, each time having to wait for the connection. Often there might be a choice between several routes; we are interested in finding the fastest one. This task is done in practice by interactive information systems, giving travellers the optimal routes to their destinations. For example, the state railway company of the Netherlands uses such a schedule information system based on the algorithm of Dijkstra, as described in [SiTu89]. We now use a somewhat simplified example to illustrate how such a problem can be modelled so that the algorithm of Dijkstra applies. For the sake of simplicity, we we restrict our interpretation to train lines and train stations, and we have our trains begin their runs at fixed intervals. Of course, any set of events occurring *at regular intervals* can be treated similarly.

We begin by constructing a digraph $G = (V, E)$ which has the train stations as vertices and the tracks between two stations as edges. With each edge

[4] I owe the material of this section to my former student, Dr. Michael Guckert.

e, we associate a travel time $f(e)$; here parallel edges might be used to model trains going at different speeds. Edges always connect two consecutive points of a line where the train stops, that is, stations a train just passes through do not occur on this line. Thus lines are just paths or cycles[5] in G. With each line L, we associate a time interval T_L representing the amount of time between two consecutive trains of this line. For each station v on a line L, we define the *time cycle* $t_L(v)$ which specifies at which times the trains of line L leave station v; this is stated modulo T_L. Now let

$$L: v_0 \overset{e_1}{\text{---}} v_1 \text{ ---} \dots \text{ ---} v_{n-1} \overset{e_n}{\text{---}} v_n$$

be a line. Clearly, the time of departure at station v_i is the sum of the time of departure at station v_{i-1} and the travelling time $f(e_i)$ from v_{i-1} to v_i, taken modulo T_L.[6] Hence the values $t_L(v_i)$ are determined as follows:[7]

$$\begin{aligned} &t_L(v_0) := s_L \pmod{T_L};\\ &t_L(v_i) := t_L(v_{i-1}) + f(e_i) \pmod{T_L} \quad \text{for } i = 1, \dots, n. \end{aligned} \tag{3.4}$$

The schedule of line L is completely determined by (3.4): the trains depart from station v_i at the time $t_L(v_i)$ (modulo T_L) in intervals of length T_L.

Next we have to calculate the waiting times involved in changing trains. Let $e = uv$ and $e' = vw$ be edges of lines L and L', respectively. A train of line L' leaves the station v at the times

$$t_{L'}(v),\ t_{L'}(v) + T_{L'},\ t_{L'}(v) + 2T_{L'}, \dots$$

and a train of line L reaches station v at the times[8]

$$t_L(v),\ t_L(v) + T_L,\ t_L(v) + 2T_L, \dots.$$

Now assume that L and L' have different time cycles. Then the waiting time depends not only on the time cycles, but also on the precise point of time modulo the least common multiple T of T_L and $T_{L'}$. Let us illustrate this by an example. Suppose the time cycle of line L is 12 minutes, while that of L' is 10 minutes so that $T = 60$. For $t_L(v) = 0$ and $t_{L'}(v) = 5$ we get the following schedules at v:

[5] Remember the Circle line in the London Underground system!

[6] We will neglect the amount of time a train stops at station v_i. This can be taken into account by either adding it to the travelling time $f(e_i)$ or by introducing an additional term $w_L(v_i)$ which then has to be added to $t_L(v_{i-1}) + f(e_i)$.

[7] Note that we cannot just put $t_L(v_0) = 0$, as different lines may leave their start stations at different times.

[8] More precisely, the trains of line L leave station v at these times, that is, they reach v a little bit earlier. We assume that this short time interval suffices for the process of changing trains, so that we can leave this out of our considerations as well.

Line L:	0	12	24		36	48
Line L' :	5	15	25	35	45	55

Thus the waiting time for the next train of line L' varies between one minute and nine minutes in this example. To simplify matters, we now assume that all time cycles are actually the same. Then the waiting time at station v is given by

$$w(v_{LL'}) = t_{L'}(v) - t_L(v) \pmod{T}.$$

This even applies in case $L = L'$: then we do not have to change trains.

Exercise 3.7.1. Reduce the case of different time cycles to the special case where all time cycles are equal.

We now construct a further digraph $G' = (V', E')$ which will allow us to find an optimal connection between two stations directly by finding a shortest path. Here a *connection* between two vertices v_0 and v_n in G means a path

$$P: v_0 \overset{e_1}{\text{---}} v_1 \text{ ---} \dots \overset{e_n}{\text{---}} v_n$$

in G together with the specification of the line L_i corresponding to edge e_i for $i = 1, \dots, n$, and the travelling time for this connection is

$$f(e_1) + w(v_{L_1L_2}) + f(e_2) + w(v_{L_2L_3}) + \dots + w(v_{L_{n-1}L_n}) + f(e_n). \tag{3.5}$$

This suggests the following definition of G'. For each vertex $v \in V$ and each line L serving station v, we have two vertices $(v, L)_{\text{in}}$ and $(v, L)_{\text{out}}$ in V'; for each edge $e = vw$ contained in some line L, there is an edge $(v, L)_{\text{out}}(w, L)_{\text{in}}$ in E'. Moreover, for each vertex v contained in both lines L and L', there is an edge $(v, L)_{\text{in}}(v, L')_{\text{out}}$. Then a directed path from v_0 to v_n in G' corresponds in fact to a connection between v_0 and v_n, and this even includes the information which lines to use and where to change trains. In order to obtain the travelling time (3.5) as the length of the corresponding path in G', we simply define a weight function w' on G' as follows:

$$\begin{aligned} w'((v, L)_{\text{out}}(w, L)_{\text{in}}) &:= f(vw)\\ w'((v, L)_{\text{in}}(v, L')_{\text{out}}) &:= w(v_{LL'}). \end{aligned}$$

Now our original problem is solved by applying Dijkstra's algorithm to the network (G', w'). Indeed, we may find all optimal connections leaving station v by applying this algorithm (modified as in Exercise 3.6.3) starting from all vertices in (G', w') which have the form $(v, L)_{\text{out}}$.

In this context, let us mention some other problems concerning the *design* of a schedule for several lines having fixed time cycles, that is, the problem of how to choose the times of departure s_L for the lines L for given time cycles T_L. In general, we might want the desired schedule to be optimal with respect to one of the following objectives.

- The longest waiting time (or the sum of all the waiting times) should be minimal.
- The shortest time interval between the departure of two trains from a station should be maximal; that is, we want a safety interval between successive trains.
- The sum of all travelling times between any two stations should be minimal; we might also give each of the routes a weight in this sum corresponding to its importance, maybe according to the expected number of travellers.

These problems are considerably more difficult; in fact, they are NP-hard in general, although polynomial solutions are known when the number of lines is small. We refer to the literature; in particular, for the first two problems see [Gul80], [Bur86], and [BrBH90]. The last problem was studied in detail by Guckert [Guc96], and the related problem of minimizing the sum of the waiting times of all travellers was treated by Domschke [Dom89]. Both these authors described and tested various heuristics.

3.8 The algorithm of Floyd and Warshall

Sometimes it is not enough to calculate the distances with respect to a certain vertex s in a given network: we need to know the distances between all pairs of vertices. Of course, we may repeatedly apply one of the algorithms treated before, varying the start vertex s over all vertices in V. This results in the following complexities, depending on the specific algorithm used.

algorithm of Moore:	$O(\|V\|\|E\|)$;
algorithm of Dijkstra:	$O(\|V\|^3)$ or $O(\|V\|\|E\|\log\|V\|)$;
algorithm of Bellman and Ford:	$O(\|V\|^2\|E\|)$.

These complexities could even be improved a bit according to the remarks at the end of Section 3.6. Takaoka [Tak92] presented an algorithm with complexity $O(|V|^3(\log\log|V|/\log|V|)^{1/2})$. In the planar case one can achieve a complexity of $O(|V|^2)$; see [Fre87].

In this section, we study an algorithm for this problem which has just the same complexity as the original version of Dijkstra's algorithm, namely $O(|V|^3)$. However, it offers the advantage of allowing some lengths to be negative – though, of course, we cannot allow cycles of negative length. This algorithm is due to Floyd [Flo62], see also Warshall [War62], and works by determining the *distance matrix* $D = (d(v,w))_{v,w\in V}$ of our network.

Algorithm 3.8.1 (Algorithm of Floyd and Warshall). Let (G,w) be a network not containing any cycles of negative length, and assume $V = \{1,\dots,n\}$. Put $w_{ij} = \infty$ if ij is not an edge in G.

Procedure FLOYD$(G,w;d)$

```
(1) for i = 1 to n do
(2)     for j = 1 to n do
(3)         if i ≠ j then d(i,j) ← w_ij else d(i,j) ← 0 fi
(4)     od
(5) od;
(6) for k = 1 to n do
(7)     for i = 1 to n do
(8)         for j = 1 to n do
(9)             d(i,j) ← min (d(i,j), d(i,k) + d(k,j))
(10)        od
(11)    od
(12) od
```

Theorem 3.8.2. *Algorithm 3.8.1 computes the distance matrix D for (G,w) with complexity $O(|V|^3)$.*

Proof. The complexity of the algorithm is obvious. Let $D_0 = (d_{ij}^0)$ denote the matrix defined in step (3) and $D_k = (d_{ij}^k)$ the matrix generated during the k-th iteration in step (9). Then D_0 contains all lengths of paths consisting of one edge only. Using induction, it is easy to see that (d_{ij}^k) is the shortest length of a directed path from i to j containing only intermediate vertices from $\{1,\dots,k\}$. As we assumed that (G,w) does not contain any cycles of negative length, the assertion follows for $k = n$. □

Exercise 3.8.3. Modify algorithm 3.8.1 so that it not only calculates the distance matrix, but also determines shortest paths between any two vertices.

Example 3.8.4. For the network shown in Figure 3.8, the algorithm of Floyd and Warshall computes the accompanying matrices.

Exercise 3.8.5. Apply Algorithm 3.8.1 to the network in Figure 3.9 [Law76].

In Section 2.6, we looked at acyclic digraphs associated with partially ordered sets. Such a digraph G is *transitive*: if there is a directed path from u to v, then G has to contain the edge uv. Now let G be an arbitrary acyclic digraph. Let us add the edge uv to G for each pair of vertices (u,v) such that v is accessible from u, but uv is not already an edge . This operation yields the *transitive closure* of G. Clearly, the transitive closure of an acyclic digraph is again acyclic and thus corresponds to a partially ordered set. By definition, two vertices u and v have distance $d(u,v) \neq \infty$ if and only if uv is an edge of the transitive closure of G. Hence the algorithm of Floyd and Warshall can be used to compute transitive closures with complexity $O(|V|^3)$.

Exercise 3.8.6. Simplify Algorithm 3.8.1 for computing the transitive closure by interpreting the adjacency matrix of an acyclic digraph as a Boolean matrix; see [War62].

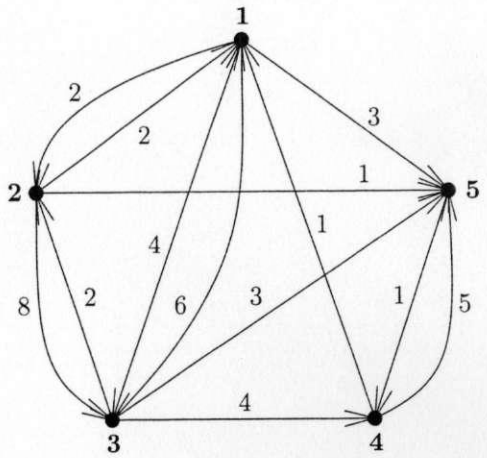

Fig. 3.8. A network

$$D_0 = \begin{pmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 8 & \infty & 1 \\ 6 & 2 & 0 & 4 & 3 \\ 1 & \infty & \infty & 0 & 5 \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 6 & \infty & 1 \\ 6 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 2 & 4 & \infty & 3 \\ 2 & 0 & 6 & \infty & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad D_3 = \begin{pmatrix} 0 & 2 & 4 & 8 & 3 \\ 2 & 0 & 6 & 10 & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 2 & 4 & 8 & 3 \\ 2 & 0 & 6 & 10 & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ 2 & 4 & 6 & 1 & 0 \end{pmatrix} \quad D_5 = \begin{pmatrix} 0 & 2 & 4 & 4 & 3 \\ 2 & 0 & 6 & 2 & 1 \\ 4 & 2 & 0 & 4 & 3 \\ 1 & 3 & 5 & 0 & 4 \\ 2 & 4 & 6 & 1 & 0 \end{pmatrix}$$

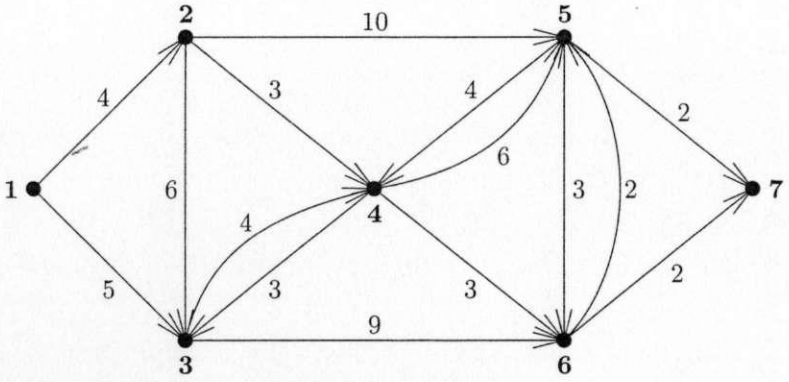

Fig. 3.9. A network

Let us mention a further way of associating an acyclic digraph to a partially ordered set. More generally, consider any acyclic digraph G. If uv is an edge in G and if there exists a directed path of length ≥ 2 from u to v in G, we remove the edge uv from G. This operation yields a digraph called the *transitive reduction* G_{red} of G. If G is the digraph associated with a partially ordered set as in Section 2.6, G_{red} is also called the *Hasse diagram* of G. If we want to draw a Hasse diagram, we usually put the vertices of equal rank on the same horizontal level. Figure 3.10 shows the Hasse diagram of the partially ordered set of the divisors of 36. The orientation of the edges is not shown explicitly: it is understood that all edges are oriented from bottom to top. As an exercise, the reader might draw some more Hasse diagrams.

Exercise 3.8.7. Design an algorithm for constructing the reduction of an acyclic digraph with complexity $O(|V|^3)$ and show that G and G_{red} have the same transitive closure. Hint: Modify the Floyd and Warshall algorithm so that it may be used here to determine longest paths.

For more about the transitive closure and the transitive reduction of an acyclic digraph see [Meh84]. Schnorr [Schn78] gave an algorithm for constructing the transitive closure with an average complexity of $O(|E|)$.

Let us consider a final application of the algorithm of Floyd and Warshall. Sometimes we are interested in finding the *center* of some network.[9] Let (G, w) be a network not containing any cycles of negative length. Then the *eccentricity* of a vertex v is defined as

$$\varepsilon(v) = \max\{d(v,u) : u \in V\}.$$

[9] It is obvious how this notion could be applied in the context of traffic or communication networks.

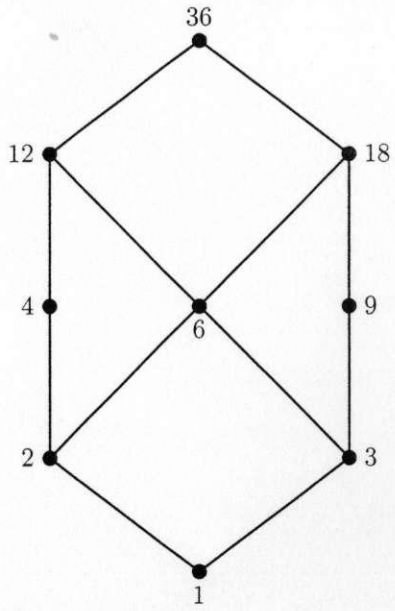

Fig. 3.10. Hasse diagram of the divisors of 36

A *center* of a network is a vertex having minimal eccentricity. The centers of a given network can be determined easily using the algorithm of Floyd and Warshall as follows. At the end of the algorithm, $\varepsilon(i)$ simply is the maximum of the i-th row of the matrix $D = (d(i,j))$, and the centers are those vertices for which this maximum is minimal. For example, the vertices of the network of Example 3.8.4 have eccentricities $\varepsilon(1) = 4$, $\varepsilon(2) = 6$, $\varepsilon(3) = 4$, $\varepsilon(4) = 5$ and $\varepsilon(5) = 6$, so that 1 and 3 are centers of the network. It is obvious that the complexity of the additional operations needed – namely finding the required maxima and minima – is dominated by the complexity $O(|V|^3)$ of the algorithm of Floyd and Warshall. Thus we have the following result.

Theorem 3.8.8. *Let N be a network without cycles of negative length. Then the centers of N can be determined with complexity $O(|V|^3)$.* □

If we take all edges in a given graph (directed or not) to have length 1, the above definition yields the eccentricities of the vertices and the centers of the graph in the graph theory sense. Sometimes we are interested in the maximal eccentricity of all vertices of a graph. This value is called the *diameter* of the graph; again, this is a notion of interest in communications networks, see [Chu86]. For more on communication networks, we also refer to [Bie89] and [Ber92]. It is a difficult (in fact, NP-hard) problem to choose and assign centers for networks under the restrictions occurring in practical applications, see [BaKP93].

To close this section, we briefly discuss the *dynamic* variant of the problem of determining shortest paths between any two vertices in a network. Suppose we have found a solution for some optimization problem, using an appropriate algorithm. For some reason, we need to change the input data slightly and find an optimal solution for the modified problem. Can we do so using the optimal solution we know already, without having to run the whole algorithm again? For our problem of finding shortest paths, this means keeping up to date the distance matrix D as well as information needed for constructing shortest paths (as, for example, the matrix $P = (p(i,j))$ used in the solution of Exercise 3.8.3) while inserting some edges or reducing lengths of edges. Compare this procedure with calculating all the entries of the matrices D and P again. If all lengths $w(e)$ are integers in the interval $[1, C]$, it is obvious that at most $O(Cn^2)$ such operations can be performed because an edge may be inserted at most once, and the length of each edge can be reduced at most C times. While a repeated application of the algorithm of Floyd and Warshall for a sequence of such operations would need $O(Cn^5)$ steps, it is also possible to solve the problem with complexity just $O(Cn^3 \log nC)$, using an adequate data structure. If we are treating an instance with graph theoretic distances, that is, for $C = 1$, a sequence of $O(n^2)$ insertions of edges needs only $O(n^3 \log n)$ steps. We refer the reader to [AuIMN91] for this topic.

3.9 Cycles of negative length

Later in this book (when treating flows and circulations in Chapter 10), we will need a method to decide whether a given network contains a directed cycle of negative length; moreover, we should also be able to find such a cycle explicitly. We shall now modify the algorithm of Floyd and Warshall to meet these requirements. The essential observation is as follows: a network (G, w) contains a directed cycle of negative length passing through the vertex i if and only if Algorithm 3.8.1 yields a negative value for $d(i,i)$.

Algorithm 3.9.1. Let (G, w) be a network with vertex set $V = \{1, \ldots, n\}$.
Procedure NEGACYCLE$(G, w; d, p, \text{neg}, K)$

```
(1) neg ← false, k ← 0;
(2) for i = 1 to n do
(3)       for j = 1 to n do
(4)             if i ≠ j then d(i,j) ← w_ij else d(i,j) ← 0 fi;
(5)             if i = j or d(i,j) = ∞ then p(i,j) ← 0 else p(i,j) ← i fi
(6)       od
(7) od;
(8) while neg = false and k < n do
(9)       k ← k + 1;
(10)      for i = 1 to n do
```

```
(11)          if d(i,k) + d(k,i) < 0
(12)          then neg ← true; CYCLE(G,p,k,i;K)
(13)          else for j = 1 to n do
(14)                   if d(i,k) + d(k,j) < d(i,j)
(15)                   then d(i,j) ← d(i,k) + d(k,j); p(i,j) ← p(k,j)
(16)                   fi
(17)               od
(18)          fi
(19)      od
(20) od
```

Here CYCLE denotes a procedure which uses p for constructing a cycle of negative length containing i and k. Note that $p(i,j)$ is, at any given point of the algorithm, the predecessor of j on a - at that point of time - shortest path from i to j. CYCLE can be described informally as follows. First, set $v_0 = i$, then $v_1 = p(k,i)$, $v_2 = p(k,v_1)$, etc., until $v_a = k = p(k,v_{a-1})$ for some index a. Then continue with $v_{a+1} = p(i,k)$, $v_{a+2} = p(i,v_{a+1})$, etc., until an index b is reached for which $v_{a+b} = v_0 = i = p(i,v_{a+b-1})$. Now the cycle we have found uses each edge in the direction opposite to its orientation, so that $(v_{a+b} = v_0, v_{a+b-1}, \ldots, v_1, v_0)$ is the desired directed cycle of negative length through i and k. It can then be stored in a list K. We leave it to the reader to state this procedure in a formally correct way.

If (G,w) does not contain any directed cycles of negative length, the variable neg has value false at the end of Algorithm 3.9.1. In this case, d contains the distances in (G,w) as in the original algorithm of Floyd and Warshall. The matrix $(p(i,j))$ may then be used to find a shortest path between any two given vertices; this is similar to the procedure CYCLE discussed above. Altogether, we get the following result.

Theorem 3.9.2. *Algorithm 3.9.1 decides with complexity $O(|V|^3)$ whether or not a given network (G,w) contains cycles of negative length; in case it does, such a cycle is constructed. Otherwise, the algorithm yields the distance matrix $(d(i,j))$ for (G,w).* □

Exercise 3.9.3. Let G be a digraph on n vertices having a root s, and let w be a length function on G. Modify the algorithm of Bellman and Ford (see Exercise 3.6.9) so that it determines whether (G,w) contains a cycle of negative length. If there is no such cycle, the algorithm should determine an SP-tree with root s using a procedure SPTREE. Write down such a procedure explicitly.

Exercise 3.9.4. Modify the algorithm of Floyd and Warshall so that it determines the shortest length of a directed cycle in a network not containing any cycles of negative length.

3.10 Path algebras

Let (G,w) be a network without cycles of negative length. According to Bellman's equations (Proposition 3.4.1), the distances u_i with respect to a vertex i then satisfy the conditions

$$(\mathrm{B}) \quad u_1 = 0 \quad \text{and} \quad u_i = \min\{u_k + w_{ki} : i \neq k\} \quad \text{for } i = 2, \ldots, n.$$

In this section, we consider the question whether such a system of equations might be solved using methods from linear algebra. In fact, this is possible by introducing appropriate algebraic structures called *path algebras*. We only sketch the basic ideas here; for details we refer to the literature, in particular [Car71, Car79, GoMi84, Zim81].[10]

We begin with a suitable transformation of the system (B). Recall that we put $w_{ij} = \infty$ if ij is not an edge of our network; therefore we extend $\mathbb{R}$ to $\overline{\mathbb{R}} = \mathbb{R} \cup \{\infty\}$. Moreover, we introduce two operations $\oplus$ and $*$ on $\overline{\mathbb{R}}$:

$$a \oplus b := \min(a,b) \qquad \text{and} \qquad a * b := a + b,$$

where, as usual, we define $a + \infty$ to be ∞. Obviously, (B) can be written as

$$u_1 = \min(0, \min\{u_k + w_{k1} : k \neq 1\}) \qquad \text{and}$$
$$u_i = \min(\infty, \min\{u_k + w_{ki} : k \neq i\}),$$

since (G,w) does not contain any cycles of negative length. Using the operations introduced above, we get the system of equations

$$(\mathrm{B}') \qquad u_1 = \bigoplus_{k=1}^{n} (u_k * w_{k1}) \oplus 0, \quad u_i = \bigoplus_{k=1}^{n} (u_k * w_{ki}) \oplus \infty,$$

where we set $w_{ii} = \infty$ for $i = 1, \ldots, n$. We can now define matrices over $\overline{\mathbb{R}}$ and apply the operations $\oplus$ and $*$ to them in analogy to the usual definitions from linear algebra. Then (B') (and hence (B)) can be written as a linear system of equations:

$$(\mathrm{B}'') \qquad u = u * W \oplus b,$$

where $u = (u_1, \ldots, u_n)$, $b = (0, \infty, \ldots, \infty)$ and $W = (w_{ij})_{i,j=1,\ldots,n}$.

Thus Bellman's equations may be viewed as a linear system of equations over the algebraic structure $(\overline{\mathbb{R}}, \oplus, *)$. Then the algorithm of Bellman and Ford given in Exercise 3.6.9 admits the following interpretation. First set

$$u^{(0)} = b \qquad \text{and then recursively} \qquad u^{(k)} = u^{(k-1)} * W \oplus b,$$

until the sequence eventually converges to $u^{(k)} = u^{(k-1)}$, which in our case occurs for $k = n$ or earlier. Hence the algorithm of Bellman and Ford is

[10] This section is included just to provide some more theoretical background. As it will not be used in the rest of the book, it may be skipped.