第 1 周阅读资料

1. Shortest paths and trees

1.1. Shortest paths with nonnegative lengths

Let $D = (V, A)$ be a directed graph, and let $s, t \in V$. A *walk* is a sequence $P = (v_0, a_1, v_1, \ldots, a_m, v_m)$ where a_i is an arc from v_{i-1} to v_i for $i = 1, \ldots, m$. If $v_0, \ldots, v_m$ all are different, P is called a *path*.

If $s = v_0$ and $t = v_m$, the vertices s and t are the *starting* and *end vertex* of P, respectively, and P is called an $s-t$ *walk*, and, if P is a path, an $s-t$ *path*. The *length* of P is m. The *distance* from s to t is the minimum length of any $s-t$ path. (If no $s-t$ path exists, we set the distance from s to t equal to ∞.)

It is not difficult to determine the distance from s to t: Let V_i denote the set of vertices of D at distance i from s. Note that for each i:

(1) V_{i+1} is equal to the set of vertices $v \in V \setminus (V_0 \cup V_1 \cup \cdots \cup V_i)$ for which $(u, v) \in A$ for some $u \in V_i$.

This gives us directly an algorithm for determining the sets V_i: we set $V_0 := \{s\}$ and next we determine with rule (1) the sets $V_1, V_2, \ldots$ successively, until $V_{i+1} = \emptyset$.

In fact, it gives a linear-time algorithm:

Theorem 1.1. *The algorithm has running time $O(|A|)$.*

Proof. Directly from the description. ∎

In fact the algorithm finds the distance from s to all vertices reachable from s. Moreover, it gives the shortest paths. These can be described by a rooted (directed) tree $T = (V', A')$, with root s, such that V' is the set of vertices reachable in D from s and such that for each $u, v \in V'$, each directed $u - v$ path in T is a shortest $u - v$ path in D.[1]

Indeed, when we reach a vertex t in the algorithm, we store the arc by which t is reached. Then at the end of the algorithm, all stored arcs form a rooted tree with this property.

There is also a trivial min-max relation characterizing the minimum length of an $s-t$ path. To this end, call a subset A' of A an $s-t$ *cut* if $A' = \delta^{\text{out}}(U)$ for some subset U of V satisfying $s \in U$ and $t \notin U$.[2] Then the following was observed by Robacker [1956]:

[1]A *rooted tree*, with *root* s, is a directed graph such that the underlying undirected graph is a tree and such that each vertex $t \neq s$ has indegree 1. Thus each vertex t is reachable from s by a unique directed $s-t$ path.

[2]$\delta^{\text{out}}(U)$ and $\delta^{\text{in}}(U)$ denote the sets of arcs leaving and entering U, respectively.

Theorem 1.2. *The minimum length of an $s-t$ path is equal to the maximum number of pairwise disjoint $s-t$ cuts.*

Proof. Trivially, the minimum is at least the maximum, since each $s-t$ path intersects each $s-t$ cut in an arc. The fact that the minimum is equal to the maximum follows by considering the $s-t$ cuts $\delta^{\text{out}}(U_i)$ for $i = 0, \dots, d-1$, where d is the distance from s to t and where U_i is the set of vertices of distance at most i from s. ∎

This can be generalized to the case where arcs have a certain 'length'. For any 'length' function $l : A \to \mathbb{Q}_+$ and any walk $P = (v_0, a_1, v_1, \dots, a_m, v_m)$, let $l(P)$ be the length of P. That is:

$$(2) \qquad l(P) := \sum_{i=1}^{m} l(a_i).$$

Now the *distance* from s to t (with respect to l) is equal to the minimum length of any $s-t$ path. If no $s-t$ path exists, the distance is $+\infty$.

Again there is an easy algorithm, due to Dijkstra [1959], to find a minimum-length $s-t$ path for all t. Start with $U := V$ and set $f(s) := 0$ and $f(v) = \infty$ if $v \neq s$. Next apply the following iteratively:

(3) Find $u \in U$ minimizing $f(u)$ over $u \in U$. For each $a = (u, v) \in A$ for which $f(v) > f(u) + l(a)$, reset $f(v) := f(u) + l(a)$. Reset $U := U \setminus \{u\}$.

We stop if $U = \emptyset$. Then:

Theorem 1.3. *The final function f gives the distances from s.*

Proof. Let $\text{dist}(v)$ denote the distance from s to v, for any vertex v. Trivially, $f(v) \geq \text{dist}(v)$ for all v, throughout the iterations. We prove that throughout the iterations, $f(v) = \text{dist}(v)$ for each $v \in V \setminus U$. At the start of the algorithm this is trivial (as $U = V$).

Consider any iteration (3). It suffices to show that $f(u) = \text{dist}(u)$ for the chosen $u \in U$. Suppose $f(u) > \text{dist}(u)$. Let $s = v_0, v_1, \dots, v_k = u$ be a shortest $s-u$ path. Let i be the smallest index with $v_i \in U$.

Then $f(v_i) = \text{dist}(v_i)$. Indeed, if $i = 0$, then $f(v_i) = f(s) = 0 = \text{dist}(s) = \text{dist}(v_i)$. If $i > 0$, then (as $v_{i-1} \in V \setminus U$):

$$(4) \qquad f(v_i) \leq f(v_{i-1}) + l(v_{i-1}, v_i) = \text{dist}(v_{i-1}) + l(v_{i-1}, v_i) = \text{dist}(v_i).$$

This implies $f(v_i) \leq \text{dist}(v_i) \leq \text{dist}(u) < f(u)$, contradicting the choice of u. ∎

Clearly, the number of iterations is $|V|$, while each iteration takes $O(|V|)$ time. So the algorithm has a running time $O(|V|^2)$. In fact, by storing for each vertex v the last arc a for which (3) applied we find a rooted tree $T = (V', A')$ with root s such that V' is the set of vertices reachable from s and such that if $u, v \in V'$ are such that T contains a directed $u - v$ path, then this path is a shortest $u - v$ path in D.

Thus we have:

Theorem 1.4. *Given a directed graph $D = (V, A)$, $s, t \in V$, and a length function $l : A \to \mathbb{Q}_+$, a shortest $s - t$ path can be found in time $O(|V|^2)$.*

Proof. See above. ∎

For an improvement, see Section 1.2.

A weighted version of Theorem 1.2 is as follows:

Theorem 1.5. *Let $D = (V, A)$ be a directed graph, $s, t \in V$, and let $l : A \to \mathbb{Z}_+$. Then the minimum length of an $s - t$ path is equal to the maximum number k of $s - t$ cuts $C_1, \ldots, C_k$ (repetition allowed) such that each arc a is in at most $l(a)$ of the cuts C_i.*

Proof. Again, the minimum is not smaller than the maximum, since if P is any $s - t$ path and $C_1, \ldots, C_k$ is any collection as described in the theorem:[3]

$$\begin{aligned} (5) \qquad l(P) &= \sum_{a \in AP} l(a) \geq \sum_{a \in AP} (\text{ number of } i \text{ with } a \in C_i) \\ &= \sum_{i=1}^{k} |C_i \cap AP| \geq \sum_{i=1}^{k} 1 = k. \end{aligned}$$

To see equality, let d be the distance from s to t, and let U_i be the set of vertices at distance less than i from s, for $i = 1, \ldots, d$. Taking $C_i := \delta^{\text{out}}(U_i)$, we obtain a collection $C_1, \ldots, C_d$ as required. ∎

Application 1.1: Shortest path. Obviously, finding a shortest route between cities is an example of a shortest path problem. The length of a connection need not be the geographical distance. It might represent the time or energy needed to make the connection. It might cost more time or energy to go from A to B than from B to A. This might be the case, for instance, when we take differences of height into account (when routing trucks), or air and ocean currents (when routing airplanes or ships).

Moreover, a route for an airplane flight between two airports so that a minimum amount of fuel is used, taking weather, altitude, velocities, and air currents into account, can be

[3] AP denotes the set of arcs traversed by P

found by a shortest path algorithm (if the problem is appropriately discretized — otherwise it is a problem of 'calculus of variations'). A similar problem occurs when finding the optimum route for boring say an underground railway tunnel.

Application 1.2: Dynamic programming. A company has to perform a job that will take 5 months. For this job a varying number of extra employees is needed:

(6)

month	number of extra employees needed
1	b_1=10
2	b_2=7
3	b_3=9
4	b_4=8
5	b_5=11

Recruiting and instruction costs EUR 800 per employee, while stopping engagement costs EUR 1200 per employee. Moreover, the company has costs of EUR 1600 per month for each employee that is engaged above the number of employees needed that month. The company now wants to decide what is the number of employees to be engaged so that the total costs will be as low as possible.

Clearly, in the example in any month i, the company should have at least b_i and at most 11 extra employees for this job. To solve the problem, make a directed graph $D = (V, A)$ with

(7) $$V := \{(i,x) \mid i = 1,\ldots,5; b_i \leq x \leq 11\} \cup \{(0,0),(6,0)\},$$
$$A := \{((i,x),(i+1,y)) \in V \times V \mid i = 0,\ldots,5\}.$$

(Figure 1.1).

At the arc from (i,x) to $(i+1,y)$ we take as length the sum of

(8) (i) the cost of starting or stopping engagement when passing from x to y employees (this is equal to $8(y-x)$ if $y \geq x$ and to $12(x-y)$ if $y < x$);
(ii) the cost of keeping the surplus of employees in month $i+1$ (that is, $16(y - b_{i+1})$)

(taking EUR 100 as unit).

Now the shortest path from $(0,0)$ to $(6,0)$ gives the number of employees for each month so that the total cost will be minimized. Finding a shortest path is thus a special case of *dynamic programming*.

1.2. Speeding up Dijkstra's algorithm with heaps

For dense graphs, a running time bound of $O(|V|^2)$ for a shortest path algorithm is best possible, since one must inspect each arc. But if $|A|$ is asymptotically smaller than $|V|^2$, one may expect faster methods.

In Dijkstra's algorithm, we spend $O(|A|)$ time on updating the values $f(u)$ and $O(|V|^2)$ time on finding a $u \in U$ minimizing $f(u)$. As $|A| \leq |V|^2$, a decrease in the running time bound requires a speed-up in finding a u minimizing $f(u)$.

A way of doing this is based on storing the u in some order so that a u minimizing $f(u)$ can be found quickly and so that it does not take too much time to restore the order if we delete a minimizing u or if we decrease some $f(u)$.

This can be done by using a 'heap', which is a rooted forest (U, F) on U, with the property that if $(u, v) \in F$ then $f(u) \leq f(v)$.[4] So at least one of the roots minimizes $f(u)$.

Let us first consider the 2-*heap*. This can be described by an ordering $u_1, \ldots, u_n$

of the elements of U such that if $i = \lfloor \frac{j}{2} \rfloor$ then $f(u_i) \leq f(u_j)$. The underlying rooted forest is in fact a rooted tree: its arcs are the pairs (u_i, u_j) with $i = \lfloor \frac{j}{2} \rfloor$.

In a 2-heap, one easily finds a u minimizing $f(u)$: it is the root u_1. The following theorem is basic for estimating the time needed for updating the 2-heap:

Theorem 1.6. *If u_1 is deleted or if some $f(u_i)$ is decreased, the 2-heap can be restored in time $O(\log p)$, where p is the number of vertices.*

Proof. To remove u_1, perform the following 'sift-down' operation. Reset $u_1 := u_n$ and $n := n - 1$. Let $i = 1$. While there is a $j \leq n$ with $2i + 1 \leq j \leq 2i + 2$ and $f(u_j) < f(u_i)$, choose one with smallest $f(u_j)$, swap u_i and u_j, and reset $i := j$.

If $f(u_i)$ has decreased perform the following 'sift-up' operation. While $i > 0$ and $f(u_j) > f(u_i)$ for $j := \lfloor \frac{i-1}{2} \rfloor$, swap u_i and u_j, and reset $i := j$. The final 2-heap is as required.

Clearly, these operations give 2-heaps as required, and can be performed in time $O(\log |U|)$. ∎

This gives the result of Johnson [1977]:

Corollary 1.6a. *Given a directed graph $D = (V, A)$, $s, t \in V$ and a length function $l : A \to \mathbb{Q}_+$, a shortest $s - t$ path can be found in time $O(|A| \log |V|)$.*

Proof. Since the number of times a minimizing vertex u is deleted and the number of times a value $f(u)$ is decreased is at most $|A|$, the theorem follows from Theorem 1.6. ∎

[4]A *rooted forest* is an acyclic directed graph $D = (V, A)$ such that each vertex has indegree at most 1. The vertices of indegree 0 are called the *roots* of D. If $(u, v) \in A$, then u is called the *parent* of v and v is called a *child* of u.

If the rooted forest has only one root, it is a *rooted tree*.

Dijkstra's algorithm has running time $O(|V|^2)$, while Johnson's heap implementation gives a running time of $O(|A| \log |V|)$. So one is not uniformly better than the other.

If one inserts a 'Fibonacci heap' in Dijkstra's algorithm, one gets a shortest path algorithm with running time $O(|A| + |V| \log |V|)$, as was shown by Fredman and Tarjan [1984].

A *Fibonacci forest* is a rooted forest (V, A), so that for each $v \in V$ the children of v can be ordered in such a way that the ith child has at least $i - 2$ children. Then:[5]

Theorem 1.7. *In a Fibonacci forest (V, A), each vertex has at most $1 + 2\log |V|$ children.*

Proof. For any $v \in V$, let $\sigma(v)$ be the number of vertices reachable from v. We show that $\sigma(v) \geq 2^{(d^{\text{out}}(v)-1)/2}$, which implies the theorem.[6]

Let $k := d^{\text{out}}(v)$ and let v_i be the ith child of v (for $i = 1, \ldots, k$). By induction, $\sigma(v_i) \geq 2^{(d^{\text{out}}(v_i)-1)/2} \geq 2^{(i-3)/2}$, as $d^{\text{out}}(v_i) \geq i - 2$. Hence $\sigma(v) = 1 + \sum_{i=1}^{k} \sigma(v_i) \geq 1 + \sum_{i=1}^{k} 2^{(i-3)/2} = 2^{(k-1)/2} + 2^{(k-2)/2} + \frac{1}{2} - \frac{1}{2}\sqrt{2} \geq 2^{(k-1)/2}$. ∎

Now a *Fibonacci heap* consists of a Fibonacci forest (U, F), where for each $v \in U$ the children of v are ordered so that the ith child has at least $i - 2$ children, and a subset T of U with the following properties:

(9)
- (i) if $(u, v) \in F$ then $f(u) \leq f(v)$;
- (ii) if v is the ith child of u and $v \notin T$ then v has at least $i - 1$ children;
- (iii) if u and v are two distinct roots, then $d^{\text{out}}(u) \neq d^{\text{out}}(v)$.

So by Theorem 1.7, (9)(iii) implies that there exist at most $2 + 2\log |U|$ roots.

The Fibonacci heap will be described by the following data structure:

(10)
- (i) for each $u \in U$, a doubly linked list C_u of children of u (in order);
- (ii) a function $p : U \to U$, where $p(u)$ is the parent of u if it has one, and $p(u) = u$ otherwise;
- (iii) the function $d^{\text{out}} : U \to \mathbb{Z}_+$;
- (iv) a function $b : \{0, \ldots, t\} \to U$ (with $t := 1 + \lfloor 2\log |V| \rfloor$) such that $b(d^{\text{out}}(u)) = u$ for each root u;
- (v) a function $l : U \to \{0, 1\}$ such that $l(u) = 1$ if and only if $u \in T$.

[5] $d^{\text{out}}(v)$ and $d^{\text{in}}(v)$ denote the outdegree and indegree of v.

[6] In fact, $\sigma(v) \geq F(d^{\text{out}}(v))$, where $F(k)$ is the kth Fibonacci number, thus explaining the name Fibonacci forest.

Theorem 1.8. *When finding and deleting n times a u minimizing $f(u)$ and decreasing m times the value $f(u)$, the structure can be updated in time $O(m + p + n \log p)$, where p is the number of vertices in the initial forest.*

Proof. Indeed, a u minimizing $f(u)$ can be identified in time $O(\log p)$, since we can scan $f(b(i))$ for $i = 0, \ldots, t$. It can be deleted as follows. Let $v_1, \ldots, v_k$ be the children of u. First delete u and all arcs leaving u from the forest. In this way, $v_1, \ldots, v_k$ have become roots, of a Fibonacci forest, and conditions (9)(i) and (ii) are maintained. To repair condition (9)(iii), do for each $r = v_1, \ldots, v_k$ the following:

```
(11)    $repair(r)$:
        if $d^{\text{out}}(s) = d^{\text{out}}(r)$ for some root $s \neq r$, then:
        {if $f(s) \leq f(r)$, add $s$ as last child of $r$ and repair($r$);
        otherwise, add $r$ as last child of $s$ and repair($s$)}.
```

Note that conditions (9)(i) and (ii) are maintained, and that the existence of a root $s \neq r$ with $d^{\text{out}}(s) = d^{\text{out}}(r)$ can be checked with the functions b, d^{out}, and p. (During the process we update the data structure.)

If we decrease the value $f(u)$ for some $u \in U$ we apply the following to u:

```
(12)    $make\ root(u)$:
        if $u$ has a parent, $v$ say, then:
        {delete arc $(v, u)$ and repair($u$);
        if $v \notin T$, add $v$ to $T$; otherwise, remove $v$ from $T$ and make root($v$)}.
```

Now denote by incr(..) and decr(..) the number of times we increase and decrease .. , respectively. Then:

(13)
$$\begin{aligned}&\text{number of calls of make root} = \text{decr}(f(u)) + \text{decr}(T)\\ &\leq \text{decr}(f(u)) + \text{incr}(T) + p \leq 2\text{decr}(f(u)) + p = 2m + p,\end{aligned}$$

since we increase T at most once after we have decreased some $f(u)$.

This also gives, where R denotes the set of roots:

(14)
$$\begin{aligned}&\text{number of calls of repair} = \text{decr}(F) + \text{decr}(R)\\ &\leq \text{decr}(F) + \text{incr}(R) + p = 2\text{decr}(F) + p\\ &\leq 2(n \log p + \text{number of calls of make root}) + p \leq 2(n \log p + 2m + p) + p.\end{aligned}$$

Since deciding calling make root or repair takes time $O(1)$ (by the data structure), we have that the algorithm takes time $O(m + p + n \log p)$. ∎

As a consequence one has:

Corollary 1.8a. *Given a directed graph $D = (V, A)$, $s, t \in V$ and a length function $l : A \to \mathbb{Q}_+$, a shortest $s - t$ path can be found in time $O(|A| + |V| \log |V|)$.*

Proof. Directly from the description of the algorithm. ∎

1.3. Shortest paths with arbitrary lengths

If lengths of arcs may take negative values, it is not always the case that a shortest walk exists. If the graph has a directed circuit of negative length, then we can obtain $s - t$ walks of arbitrary small negative length (for appropriate s and t).

However, it can be shown that if there are no directed circuits of negative length, then for each choice of s and t there exists a shortest $s - t$ walk (if there exists at least one $s - t$ path).

Theorem 1.9. *Let each directed circuit have nonnegative length. Then for each pair $s.t$ of vertices for which there exists at least one $s - t$ walk. there exists a shortest $s - t$ walk, which is a path.*

Proof. Clearly, if there exists an $s - t$ walk, there exists an $s - t$ path. Hence there exists also a shortest path P, that is, an $s - t$ path that has minimum length *among all $s - t$ paths*. This follows from the fact that there exist only finitely many paths.

We show that P is shortest among *all* $s - t$ walks. Let P have length L. Suppose that there exists an $s - t$ walk Q of length less than L. Choose such a Q with a minimum number of arcs. Since Q is not a path (as it has length less than L), Q contains a directed circuit C. Let Q' be the walk obtained from Q by removing C. As $l(C) \geq 0$, $l(Q') = l(Q) - l(C) \leq l(Q) < L$. So Q' is another $s - t$ walk of length less than L, however with a smaller number of arcs than Q. This contradicts the assumption that Q has a minimum number of arcs. ∎

Also in this case there is an easy algorithm, the *Bellman-Ford method* (Bellman [1958], Ford [1956]), determining a shortest $s - t$ path.

Let $n := |V|$. The algorithm calculates functions $f_0, f_1, f_2, \ldots, f_n : V \to \mathbb{R} \cup \{\infty\}$ successively by the following rule:

(15) (i) Put $f_0(s) := 0$ and $f_0(v) := \infty$ for all $v \in V \setminus \{s\}$.
(ii) For $k < n$, if f_k has been found, put

$$f_{k+1}(v) := \min\{f_k(v), \min_{(u,v) \in A} (f_k(u) + l(u, v))\}$$

for all $v \in V$.

Then, assuming that there is no directed circuit of negative length, $f_n(v)$ is equal to the length of a shortest $s-v$ walk, for each $v \in V$. (If there is no $s-v$ path at all, $f_n(v) = \infty$.)

This follows directly from the following theorem:

Theorem 1.10. *For each $k = 0, \ldots, n$ and for each $v \in V$,*

(16) $$f_k(v) = \min\{l(P) \mid P \text{ is an } s-v \text{ walk traversing at most } k \text{ arcs}\}.$$

Proof. By induction on k from (15). ∎

So the above method gives us the length of a shortest $s-t$ path. It is not difficult to derive a method finding an explicit shortest $s-t$ path. To this end, determine parallel to the functions $f_0, \ldots, f_n$, a function $g : V \to V$ by setting $g(v) = u$ when we set $f_{k+1}(v) := f_k(u) + l(u, v)$ in (15)(ii). At termination, for any v, the sequence v, $g(v)$, $g(g(v)), \ldots, s$ gives the reverse of a shortest $s-v$ path. Therefore:

Corollary 1.10a. *Given a directed graph $D = (V, A)$, $s, t \in V$ and a length function $l : A \to \mathbb{Q}$, such that D has no negative-length directed circuit, a shortest $s-t$ path can be found in time $O(|V||A|)$.*

Proof. Directly from the description of the algorithm. ∎

Application 1.3: Knapsack problem. Suppose we have a knapsack with a volume of 8 liter and a number of articles $1, 2, 3, 4, 5$. Each of the articles has a certain volume and a certain value:

(17)

article	volume	value
1	5	4
2	3	7
3	2	3
4	2	5
5	1	4

So we cannot take all articles in the knapsack and we have to make a selection. We want to do this so that the total value of articles taken into the knapsack is as large as possible.

We can describe this problem as one of finding x_1, x_2, x_3, x_4, x_5 such that:

(18)
$$x_1, x_2, x_3, x_4, x_5 \in \{0, 1\},$$
$$5x_1 + 3x_2 + 2x_3 + 2x_4 + x_5 \leq 8,$$
$$4x_1 + 7x_2 + 3x_3 + 5x_4 + 4x_5 \text{ is as large as possible.}$$

We can solve this problem with the shortest path method as follows. Make a directed graph in the following way (cf. Figure 1.2):

There are vertices (i, x) for $0 \leq i \leq 6$ and $0 \leq x \leq 8$ and there is an arc from $(i-1, x)$ to (i, y) if $y = x$ or $y = x + a_i$ (where a_i is the volume of article i) if $i \leq 5$ and there are arcs from each $(5, x)$ to $(6, 8)$. We have deleted in the picture all vertices and arcs that do not belong to any directed path from $(0, 0)$.

The length of arc $((i-1, x), (i, y))$ is equal to 0 if $y = x$ and to $-c_i$ if $y = x + a_i$ (where c_i denotes the value of i). Moreover, all arcs ending at $(6, 8)$ have length 0.

Now a shortest path from $(0, 0)$ to $(6, 8)$ gives us the optimal selection.

Application 1.4: PERT-CPM. For building a house certain activities have to be executed. Certain activities have to be done before other and every activity takes a certain number of days:

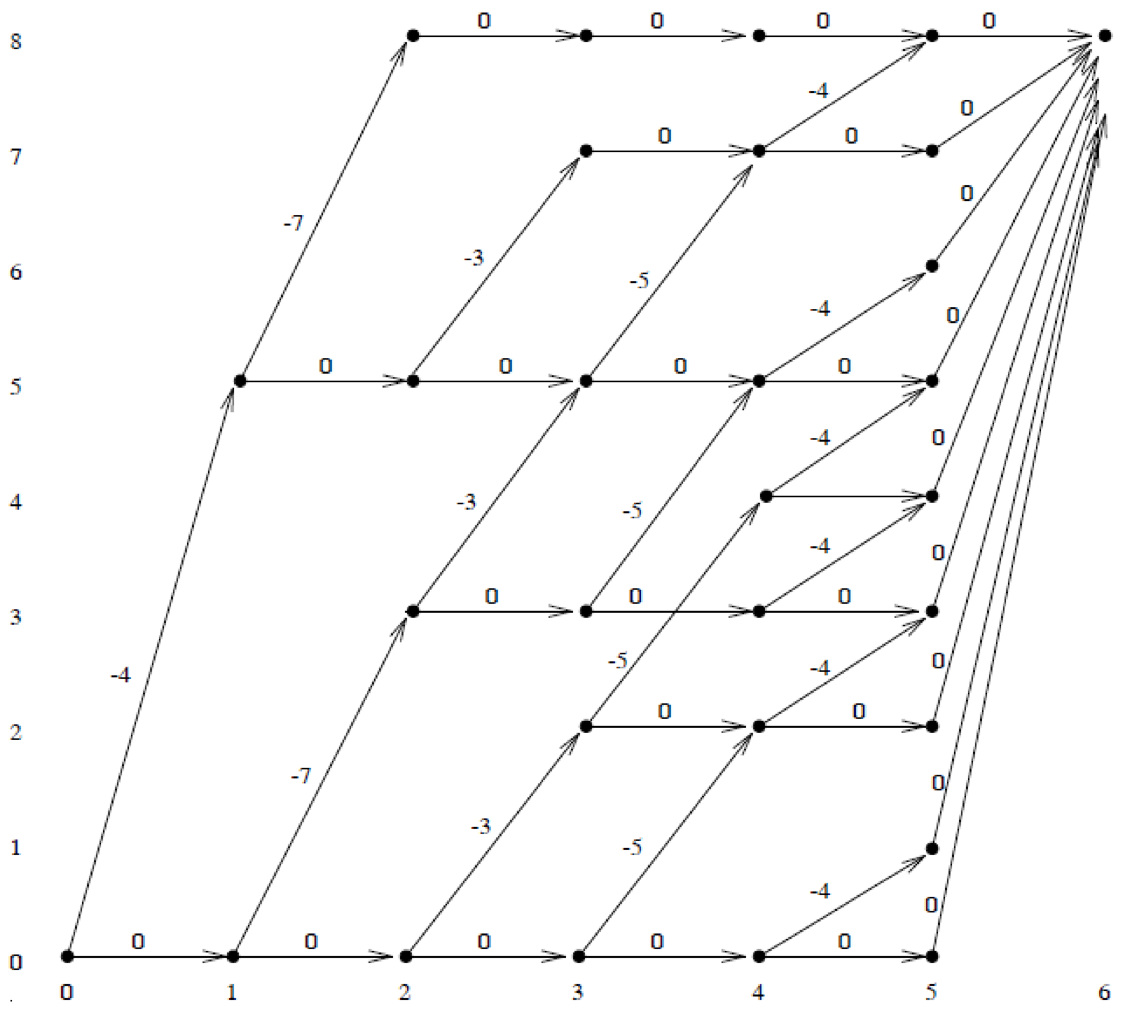

Figure 1.2

(19)	activity	days needed	to be done before activity #
1.	groundwork	2	2
2.	foundation	4	3
3.	building walls	10	4,6,7
4.	exterior plumbing	4	5,9
5.	interior plumbing	5	10
6.	electricity	7	10
7.	roof	6	8
8.	finishing off outer walls	7	9
9.	exterior painting	9	14
10.	panelling	8	11,12
11.	floors	4	13
12.	interior painting	5	13
13.	finishing off interior	6	
14.	finishing off exterior	2	

We introduce two dummy activities 0 (start) and 15 (completion), each taking 0 days, where activity 0 has to be performed before all other activities and 15 after all other activities.

The project can be represented by a directed graph D with vertices $0, 1, \ldots, 14, 15$, where there is an arc from i to j if i has to be performed before j. The length of arc (i, j) will be the number t_i of days needed to perform activity i. This graph with length function is called the *project network*.

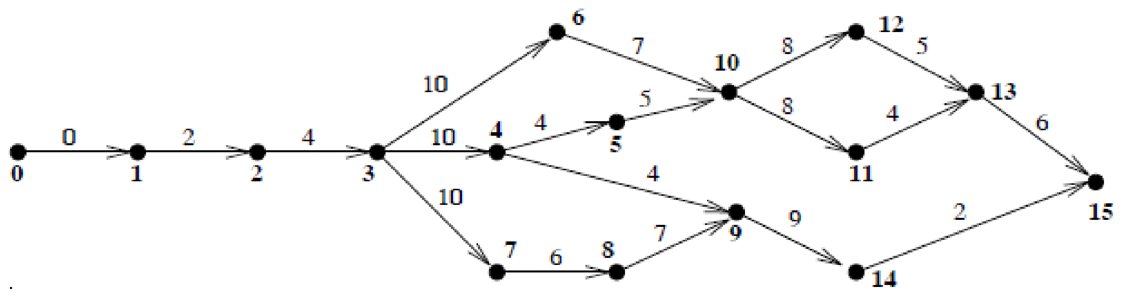

Figure 1.3

Now a *longest* path from 0 to 15 gives the minimum number of days needed to build the house. Indeed, if l_i denotes the length of a longest path from 0 to i, we can start activity i on day l_i. If activity j has been done after activity i, then $l_j \geq l_i + t_i$ by definition of longest path. So there is sufficient time for completing activity i and the schedule is practically feasible. That is, there is the following min-max relation:

(20) the minimum number of days needed to finish the project is equal to the maximum length of a path in the project network.

A longest path can be found with the Bellman-Ford method, as it is equivalent to a shortest path when we replace each length by its opposite. Note that D should not have any directed circuits since otherwise the whole project would be infeasible.

So the project network helps in planning the project and is the basis of the so-called 'Program Evaluation and Review Technique' (PERT). (Actually, one often represents activities by arcs instead of vertices, giving a more complicated way of defining the graph.)

Any longest path from 0 to 15 gives the minimum number of days needed to complete the project. Such a path is called a *critical path* and gives us the bottlenecks in the project. It tells us which activities should be controlled carefully in order to meet a deadline. At least one of these activities should be sped up if we wish to complete the project faster. This is the basis of the 'Critical Path Method' (CPM).

Application 1.5: Price equilibrium. A small example of an economical application is as follows. Consider a number of remote villages, say B, C, D, E and F. Certain pairs of villages are connected by routes (like in Figure 1.4).

If villages X and Y are connected by a route, let $k_{X,Y}$ be the cost of transporting one liter of oil from X to Y.

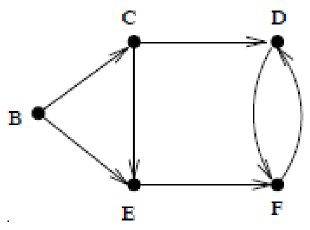

Figure 1.4

At a certain day, one detects an oil well in village B, and it makes oil freely available in village B. Now one can follow how the oil price will develop, assuming that no other oil than that from the well in B is available and that only once a week there is contact between adjacent villages.

It will turn out that the oil prices in the different villages will follow the iterations in the Bellman-Ford algorithm. Indeed in week 0 (the week in which the well was detected) the price in B equals 0, while in all other villages the price is ∞, since there is simply no oil available yet.

In week 1, the price in B equals 0, the price in any village Y adjacent to B is equal to $k_{B,Y}$ per liter and in all other villages it is still ∞.

In week $i+1$ the liter price $p_{i+1,Y}$ in any village Y is equal to the minimum value of $p_{i,Y}$ and all $p_{i,X} + k_{X,Y}$ for which there is a connection from X to Y.

There will be price equilibrium if for each village Y one has:

(21) it is not cheaper for the inhabitants of Y to go to an adjacent village X and to transport the oil from X to Y.

Moreover, one has the min-max relation for each village Y:

(22) the maximum liter price in village Y is equal to the the minimum length of a path in the graph from B to Y

taking $k_{X,Y}$ as length function.

A comparable, but less spatial example is: the vertices of the graph represent oil products (instead of villages) and $k_{X,Y}$ denotes the cost per unit of transforming oil product X to oil product Y. If oil product B is free, one can determine the costs of the other products in the same way as above.

1.4. Minimum spanning trees

Let $G = (V, E)$ be a connected undirected graph and let $l : E \to \mathbb{R}$ be a function, called the *length* function. For any subset F of E, the *length* $l(F)$ of F is, by definition:

(23) $$l(F) := \sum_{e\in F} l(e).$$

In this section we consider the problem of finding a spanning tree in G of minimum length. There is an easy algorithm for finding a minimum-length spanning tree, essentially due to Borůvka [1926]. There are a few variants. The first one we discuss is sometimes called the *Dijkstra-Prim method* (after Prim [1957] and Dijkstra [1959]).

Choose a vertex $v_1 \in V$ arbitrarily. Determine edges $e_1, e_2 \dots$ successively as follows. Let $U_1 := \{v_1\}$. Suppose that, for some $k \geq 0$, edges $e_1, \dots, e_k$ have been chosen, forming a spanning tree on the set U_k. Choose an edge $e_{k+1} \in \delta(U_k)$ that has minimum length among all edges in $\delta(U_k)$.[7] Let $U_{k+1} := U_k \cup e_{k+1}$.

By the connectedness of G we know that we can continue choosing such an edge until $U_k = V$. In that case the selected edges form a spanning tree T in G. This tree has minimum length, which can be seen as follows.

Call a forest F *greedy* if there exists a minimum-length spanning tree T of G that contains F.

Theorem 1.11. *Let F be a greedy forest, let U be one of its components, and let $e \in \delta(U)$. If e has minimum length among all edges in $\delta(U)$, then $F \cup \{e\}$ is again a greedy forest.*

Proof. Let T be a minimum-length spanning tree containing F. Let P be the unique path in T between the end vertices of e. Then P contains at least one edge f that belongs to $\delta(U)$. So $T' := (T \setminus \{f\}) \cup \{e\}$ is a tree again. By assumption, $l(e) \leq l(f)$ and hence $l(T') \leq l(T)$. Therefore, T' is a minimum-length spanning tree. As $F \cup \{e\} \subseteq T'$, it follows that $F \cup \{e\}$ is greedy. ∎

Corollary 1.11a. *The Dijkstra-Prim method yields a spanning tree of minimum length.*

[7] $\delta(U)$ is the set of edges e satisfying $|e \cap U| = 1$.

Proof. It follows inductively with Theorem 1.11 that at each stage of the algorithm we have a greedy forest. Hence the final tree is greedy — equivalently, it has minimum length. ∎

In fact one may show:

Theorem 1.12. *Implementing the Dijkstra-Prim method using Fibonacci heaps gives a running time of* $O(|E| + |V| \log |V|)$.

Proof. The Dijkstra-Prim method is similar to Dijkstra's method for finding a shortest path. Throughout the algorithm, we store at each vertex $v \in V \setminus U_k$, the length $f(v)$ of a shortest edge $\{u, v\}$ with $u \in U_k$, organized as a Fibonacci heap. A vertex u_{k+1} to be added to U_k to form U_{k+1} should be identified and removed from the Fibonacci heap. Moreover, for each edge e connecting u_{k+1} and some $v \in V \setminus U_{k+1}$, we should update $f(v)$ if the length of $u_{k+1}v$ is smaller than $f(v)$.

Thus we find and delete $\leq |V|$ times a u minimizing $f(u)$ and we decrease $\leq |E|$ times a value $f(v)$. Hence by Theorem 1.8 the algorithm can be performed in time $O(|E| + |V| \log |V|)$. ∎

The Dijkstra-Prim method is an example of a so-called *greedy* algorithm. We construct a spanning tree by throughout choosing an edge that seems the best at the moment. Finally we get a minimum-length spanning tree. Once an edge has been chosen, we never have to replace it by another edge (no 'back-tracking').

There is a slightly different method of finding a minimum-length spanning tree, *Kruskal's* method (Kruskal [1956]). It is again a greedy algorithm, and again iteratively edges $e_1, e_2, \ldots$ are chosen, but by some different rule.

Suppose that, for some $k \geq 0$, edges $e_1, \ldots, e_k$ have been chosen. Choose an edge e_{k+1} such that $\{e_1, \ldots, e_k, e_{k+1}\}$ forms a forest, with $l(e_{k+1})$ as small as possible. By the connectedness of G we can (starting with $k = 0$) iterate this until the selected edges form a spanning tree of G.

Corollary 1.12a. *Kruskal's method yields a spanning tree of minimum length.*

Proof. Again directly from Theorem 1.11. ∎

In a similar way one finds a *maximum*-length spanning tree.

Application 1.6: Minimum connections. There are several obvious practical situations where finding a minimum-length spanning tree is important, for instance, when designing a road system, electrical power lines, telephone lines, pipe lines, wire connections on a chip. Also when clustering data say in taxonomy, archeology, or zoology, finding a minimum spanning tree can be helpful.

Application 1.7: The maximum reliability problem. Often in designing a network one is not primarily interested in minimizing length, but rather in maximizing 'reliability' (for instance when designing energy or communication networks). Certain cases of this problem can be seen as finding a *maximum* length spanning tree, as was observed by Hu [1961]. We give a mathematical description.

Let $G = (V, E)$ be a graph and let $s : E \to \mathbb{R}_+$ be a function. Let us call $s(e)$ the *strength* of edge e. For any path P in G, the *reliability* of P is, by definition, the minimum strength of the edges occurring in P. The *reliability* $r_G(u, v)$ of two vertices u and v is equal to the maximum reliability of P, where P ranges over all paths from u to v.

Let T be a spanning tree of maximum strength, i.e., with $\sum_{e \in ET} s(e)$ as large as possible. (Here ET is the set of edges of T.) So T can be found with any maximum spanning tree algorithm.

Now T has the same reliability as G, for each pair of vertices u, v. That is:

(24) $$r_T(u, v) = r_G(u, v) \text{ for each } u, v \in V.$$

We leave the proof as an exercise (Exercise 1.11).