# 6

# Tree-Search Algorithms

## Contents



## 6.1 Tree-Search

We have seen that connectedness is a basic property of graphs. But how does one determine whether a graph is connected? In the case of small graphs, it is a routine matter to do so by inspection, searching for paths between all pairs of vertices. However, in large graphs, such an approach could be time-consuming because the number of paths to examine might be prohibitive. It is therefore desirable to have a systematic procedure, or *algorithm*, which is both efficient and applicable to all graphs. The following property of the trees of a graph provides the basis for such a procedure. For a subgraph $F$ of a graph $G$, we simply write $\partial(F)$ for $\partial(V(F))$, and refer to this set as the *edge cut* associated with $F$.

Let $T$ be a tree in a graph $G$. If $V(T) = V(G)$, then $T$ is a spanning tree of $G$ and we may conclude, by Theorem 4.6, that $G$ is connected. But if $V(T) \subset V(G)$, two possibilities arise: either $\partial(T) = \emptyset$, in which case $G$ is disconnected, or $\partial(T) \neq \emptyset$. In the latter case, for any edge $xy \in \partial(T)$, where $x \in V(T)$ and $y \in V(G) \setminus V(T)$,

the subgraph of $G$ obtained by adding the vertex $y$ and the edge $xy$ to $T$ is again a tree in $G$ (see Figure 6.1).

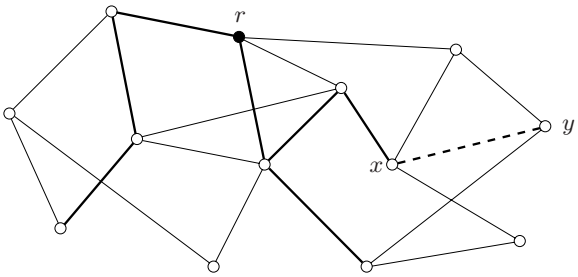



**Fig. 6.1.** Growing a tree in a graph

Using the above idea, one may generate a sequence of rooted trees in $G$, starting with the trivial tree consisting of a single root vertex $r$, and terminating either with a spanning tree of the graph or with a nonspanning tree whose associated edge cut is empty. (In practice, this involves scanning the adjacency lists of the vertices already in the tree, one by one, to determine which vertex and edge to add to the tree.) We refer to such a procedure as a *tree-search* and the resulting tree as a *search tree*.

If our objective is just to determine whether a graph is connected, any tree-search will do. In other words, the order in which the adjacency lists are considered is immaterial. However, tree-searches in which specific criteria are used to determine this order can provide additional information on the structure of the graph. For example, a tree-search known as *breadth-first search* may be used to find the distances in a graph, and another, *depth-first search*, to find the cut vertices of a graph.

The following terminology is useful in describing the properties of search trees. Recall that an *$r$-tree* is a tree with root $r$. Let $T$ be such a tree. The *level* of a vertex $v$ in $T$ is the length of the path $rTv$. Each edge of $T$ joins vertices on consecutive levels, and it is convenient to think of these edges as being oriented from the lower to the higher level, so as to form a branching. Several other terms customarily used in the study of rooted trees are borrowed from genealogy. For instance, each vertex on the path $rTv$, including the vertex $v$ itself, is called an *ancestor* of $v$, and each vertex of which $v$ is an ancestor is a *descendant* of $v$. An ancestor or descendant of a vertex is *proper* if it is not the vertex itself. Two vertices are *related* in $T$ if one is an ancestor of the other. The immediate proper ancestor of a vertex $v$ other than the root is its *predecessor* or *parent*, denoted $p(v)$, and the vertices whose predecessor is $v$ are its *successors* or *children*. Note that the (oriented) edge set of a rooted tree $T := (V(T), E(T))$ is determined by its predecessor function $p$, and conversely

$$E(T) = \{(p(v), v) : v \in V(T) \setminus \{r\}\}$$

where $r$ is the root of $T$. We often find it convenient to describe a rooted tree by specifying its vertex set and predecessor function.

For the sake of simplicity, we assume throughout this chapter that our graphs and digraphs are connected. This assumption results in no real loss of generality. We may suppose that the components have already been found by means of a tree-search. Each component may then be treated individually. We also assume that our graphs and digraphs are free of loops, which play an insignificant role here.

### Breadth-First Search and Shortest Paths

In most types of tree-search, the criterion for selecting a vertex to be added to the tree depends on the order in which the vertices already in the tree $T$ were added. A tree-search in which the adjacency lists of the vertices of $T$ are considered on a first-come first-served basis, that is, in increasing order of their time of incorporation into $T$, is known as *breadth-first search*. In order to implement this algorithm efficiently, vertices in the tree are kept in a *queue*; this is just a list $Q$ which is updated either by adding a new element to one end (the *tail* of $Q$) or removing an element from the other end (the *head* of $Q$). At any moment, the queue $Q$ comprises all vertices from which the current tree could potentially be grown.

Initially, at time $t = 0$, the queue $Q$ is empty. Whenever a new vertex is added to the tree, it joins $Q$. At each stage, the adjacency list of the vertex at the head of $Q$ is scanned for a neighbour to add to the tree. If every neighbour is already in the tree, this vertex is removed from $Q$. The algorithm terminates when $Q$ is once more empty. It returns not only the tree (given by its predecessor function $p$), but also a function $\ell : V \to \mathbb{N}$, which records the level of each vertex in the tree and, more importantly, their distances from $r$ in $G$. It also returns a function $t : V \to \mathbb{N}$ which records the time of incorporation of each vertex into the tree $T$. We keep track of the vertices in $T$ by colouring them black. The notation $G(x)$ signifies a graph $G$ with a specified vertex (or *root*) $x$. Recall that an *$x$-tree* is a tree rooted at vertex $x$.

**Algorithm 6.1** Breadth-First Search (BFS)

Input: a connected graph $G(r)$

Output: an $r$-tree $T$ in $G$ with predecessor function $p$, a level function $\ell$ such that $\ell(v) = d_G(r, v)$ for all $v \in V$, and a time function $t$

*set* $i := 0$ *and* $Q := \emptyset$
*increment* $i$ *by* 1
*colour* $r$ *black*
*set* $\ell(r) := 0$ *and* $t(r) := i$
*append* $r$ *to* $Q$
***while*** $Q$ *is nonempty* ***do***
  *consider the head* $x$ *of* $Q$
  ***if*** $x$ *has an uncoloured neighbour* $y$ ***then***
    *increment* $i$ *by* 1

*10:* *colour $y$ black*
*11:* *set $p(y) := x$, $\ell(y) := \ell(x) + 1$ and $t(y) := i$*
*12:* *append $y$ to $Q$*
*13:* ***else***
*14:* *remove $x$ from $Q$*
*15:* ***end if***
*16:* ***end while***
*17:* *return* $(p, \ell, t)$

The spanning tree $T$ returned by BFS is called a *breadth-first search tree*, or *BFS-tree*, of $G$. An example of a BFS-tree in a connected graph is shown in Figure 6.2. The labels of the vertices in Figure 6.2a indicate the times at which they were added to the tree. The distance function $\ell$ is shown in Figure 6.2b. The evolution of the queue $Q$ is as follows, the vertices being indicated by their times.

$$\begin{aligned}
&\emptyset \to 1 \to 1\,2 \to 1\,2\,3 \to 1\,2\,3\,4 \to 1\,2\,3\,4\,5 \to 2\,3\,4\,5 \to 2\,3\,4\,5\,6\\
&\quad \to 2\,3\,4\,5\,6\,7 \to 3\,4\,5\,6\,7 \to 3\,4\,5\,6\,7\,8 \to 3\,4\,5\,6\,7\,8\,9 \to 4\,5\,6\,7\,8\,9\\
&\quad \to 4\,5\,6\,7\,8\,9\,10 \to 5\,6\,7\,8\,9\,10 \to 5\,6\,7\,8\,9\,10\,11 \to 6\,7\,8\,9\,10\,11\\
&\quad \to 6\,7\,8\,9\,10\,11\,12 \to 7\,8\,9\,10\,11\,12 \to 8\,9\,10\,11\,12 \to 9\,10\,11\,12\\
&\quad \to 9\,10\,11\,12\,13 \to 10\,11\,12\,13 \to 11\,12\,13 \to 12\,13 \to 13 \to \emptyset
\end{aligned}$$

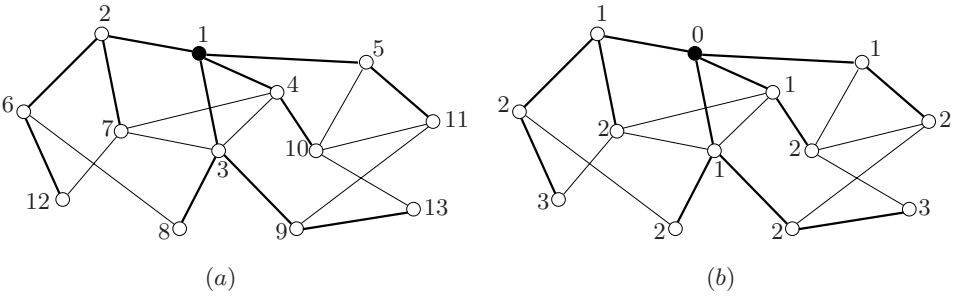



**Fig. 6.2.** A breadth-first search tree in a connected graph: (a) the time function $t$, and (b) the level function $\ell$

BFS-trees have two basic properties, the first of which justifies our referring to $\ell$ as a level function.

**Theorem 6.2** *Let $T$ be a BFS-tree of a connected graph $G$, with root $r$. Then:*

*a) for every vertex $v$ of $G$, $\ell(v) = d_T(r, v)$, the level of $v$ in $T$,*
*b) every edge of $G$ joins vertices on the same or consecutive levels of $T$; that is,*

$$|\ell(u) - \ell(v)| \leq 1, \quad \textit{for all } uv \in E$$

**Proof** The proof of (a) is left to the reader (Exercise 6.1.1). To establish (b), it suffices to prove that if $uv \in E$ and $\ell(u) < \ell(v)$, then $\ell(u) = \ell(v) - 1$.

We first establish, by induction on $\ell(u)$, that if $u$ and $v$ are any two vertices such that $\ell(u) < \ell(v)$, then $u$ joined $Q$ before $v$. This is evident if $\ell(u) = 0$, because $u$ is then the root of $T$. Suppose that the assertion is true whenever $\ell(u) < k$, and consider the case $\ell(u) = k$, where $k > 0$. Setting $x := p(u)$ and $y := p(v)$, it follows from line 11 of BFS (Algorithm 6.1) that $\ell(x) = \ell(u) - 1 < \ell(v) - 1 = \ell(y)$. By induction, $x$ joined $Q$ before $y$. Therefore $u$, being a neighbour of $x$, joined $Q$ before $v$.

Now suppose that $uv \in E$ and $\ell(u) < \ell(v)$. If $u = p(v)$, then $\ell(u) = \ell(v) - 1$, again by line 11 of the algorithm. If not, set $y := p(v)$. Because $v$ was added to $T$ by the edge $yv$, and not by the edge $uv$, the vertex $y$ joined $Q$ before $u$, hence $\ell(y) \leq \ell(u)$ by the claim established above. Therefore $\ell(v) - 1 = \ell(y) \leq \ell(u) \leq \ell(v) - 1$, which implies that $\ell(u) = \ell(v) - 1$. □

The following theorem shows that BFS runs correctly.

**Theorem 6.3** *Let $G$ be a connected graph. Then the values of the level function $\ell$ returned by BFS are the distances in $G$ from the root $r$:*

$$\ell(v) = d_G(r, v), \quad \textit{for all } v \in V$$

**Proof** By Theorem 6.2a, $\ell(v) = d_T(r, v)$. Moreover, $d_T(r, v) \geq d_G(r, v)$ because $T$ is a subgraph of $G$. Thus $\ell(v) \geq d_G(r, v)$. We establish the opposite inequality by induction on the length of a shortest $(r, v)$-path.

Let $P$ be a shortest $(r, v)$-path in $G$, where $v \neq r$, and let $u$ be the predecessor of $v$ on $P$. Then $rPu$ is a shortest $(r, u)$-path, and $d_G(r, u) = d_G(r, v) - 1$. By induction, $\ell(u) \leq d_G(r, u)$, and by Theorem 6.2b, $\ell(v) - \ell(u) \leq 1$. Therefore

$$\ell(v) \leq \ell(u) + 1 \leq d_G(r, u) + 1 = d_G(r, v)$$ □

Alternative proofs of Theorems 6.2 and 6.3 are outlined in Exercise 6.1.2.

### Depth-First Search

*Depth-first search* is a tree-search in which the vertex added to the tree $T$ at each stage is one which is a neighbour of as recent an addition to $T$ as possible. In other words, we first scan the adjacency list of the most recently added vertex $x$ for a neighbour not in $T$. If there is such a neighbour, we add it to $T$. If not, we backtrack to the vertex which was added to $T$ just before $x$ and examine its neighbours, and so on. The resulting spanning tree is called a *depth-first search tree* or *DFS-tree*.

This algorithm may be implemented efficiently by maintaining the vertices of $T$ whose adjacency lists have yet to be fully scanned, not in a queue as we did for breadth-first search, but in a stack. A *stack* is simply a list, one end of which is identified as its *top*; it may be updated either by adding a new element as its top

or else by removing its top element. In depth-first search, the stack $S$ is initially empty. Whenever a new vertex is added to the tree $T$, it is added to $S$. At each stage, the adjacency list of the top vertex is scanned for a neighbour to add to $T$. If all of its neighbours are found to be already in $T$, this vertex is removed from $S$. The algorithm terminates when $S$ is once again empty. As in breadth-first search, we keep track of the vertices in $T$ by colouring them black.

Associated with each vertex $v$ of $G$ are two times: the time $f(v)$ when $v$ is incorporated into $T$ (that is, added to the stack $S$), and the time $l(v)$ when all the neighbours of $v$ are found to be already in $T$, the vertex $v$ is removed from $S$, and the algorithm backtracks to $p(v)$, the predecessor of $v$ in $T$. (The time function $l(v)$ is not to be confused with the level function $\ell(v)$ of BFS.) The time increments by one with each change in the stack $S$. In particular, $f(r) = 1$, $l(v) = f(v) + 1$ for every leaf $v$ of $T$, and $l(r) = 2n$.

**Algorithm 6.4** Depth-First Search (DFS)

Input: a connected graph $G$
Output: a rooted spanning tree of $G$ with predecessor function $p$, and two time functions $f$ and $l$

*set* $i := 0$ *and* $S := \emptyset$
*choose any vertex* $r$ *(as root)*
*increment* $i$ *by* 1
*colour* $r$ *black*
*set* $f(r) := i$
*add* $r$ *to* $S$
***while*** $S$ *is nonempty* ***do***
    *consider the top vertex* $x$ *of* $S$
    *increment* $i$ *by* 1
    ***if*** $x$ *has an uncoloured neighbour* $y$ ***then***
        *colour* $y$ *black*
        *set* $p(y) := x$ *and* $f(y) := i$
        *add* $y$ *to the top of* $S$
    ***else***
        *set* $l(x) := i$
        *remove* $x$ *from* $S$
    ***end if***
***end while***
*return* $(p, f, l)$

A DFS-tree of a connected graph is shown in Figure 6.3; the tree is indicated by solid lines and each vertex $v$ of the tree is labelled by the pair $(f(v), l(v))$. The evolution of the stack $S$ is as follows, the vertices being indicated by their times of incorporation into $T$.

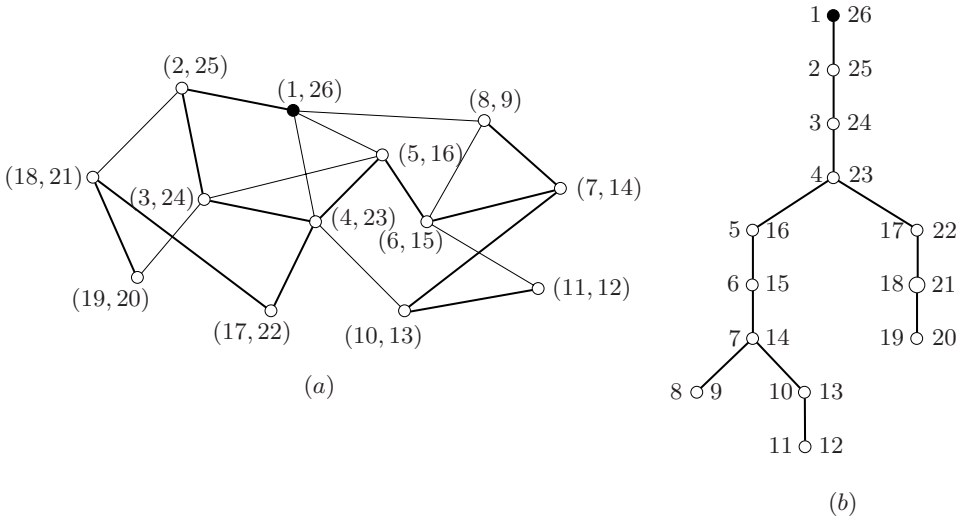



**Fig. 6.3.** (a) A depth-first search tree of a connected graph, and (b) another drawing of this tree

$$\begin{aligned}
&\emptyset \to 1 \to 1\,2 \to 1\,2\,3 \to 1\,2\,3\,4 \to 1\,2\,3\,4\,5 \to 1\,2\,3\,4\,5\,6 \to 1\,2\,3\,4\,5\,6\,7\\
&\to 1\,2\,3\,4\,5\,6\,7\,8 \to 1\,2\,3\,4\,5\,6\,7 \to 1\,2\,3\,4\,5\,6\,7\,10 \to 1\,2\,3\,4\,5\,6\,7\,10\,11\\
&\to 1\,2\,3\,4\,5\,6\,7\,10 \to 1\,2\,3\,4\,5\,6\,7 \to 1\,2\,3\,4\,5\,6 \to 1\,2\,3\,4\,5 \to 1\,2\,3\,4 \to 1\,2\,3\,4\,17\\
&\to 1\,2\,3\,4\,17\,18 \to 1\,2\,3\,4\,17\,18\,19 \to 1\,2\,3\,4\,17\,18 \to 1\,2\,3\,4\,17 \to 1\,2\,3\,4 \to 1\,2\,3\\
&\to 1\,2 \to 1 \to \emptyset
\end{aligned}$$

The following proposition provides a link between the input graph $G$, its DFS-tree $T$, and the two time functions $f$ and $l$ returned by DFS.

**Proposition 6.5** *Let $u$ and $v$ be two vertices of $G$, with $f(u) < f(v)$.*

*a) If $u$ and $v$ are adjacent in $G$, then $l(v) < l(u)$.*
*b) $u$ is an ancestor of $v$ in $T$ if and only if $l(v) < l(u)$.*

**Proof**

a) According to lines 8–12 of DFS, the vertex $u$ is removed from the stack $S$ only after all potential children (uncoloured neighbours) have been considered for addition to $S$. One of these neighbours is $v$, because $f(u) < f(v)$. Thus $v$ is added to the stack $S$ while $u$ is still in $S$, and $u$ cannot be removed from $S$ before $v$ is removed. It follows that $l(v) < l(u)$.
b) Suppose that $u$ is an ancestor of $v$ in $T$. By lines 9 and 12 of DFS, the values of $f$ increase along the path $uTv$. Applying (a) to each edge of this path yields the inequality $l(v) < l(u)$.

Now suppose that $u$ is not an ancestor of $v$ in $T$. Because $f(u) < f(v)$, $v$ is not an ancestor of $u$ either. Thus $u$ does not lie on the path $rTv$ and $v$ does not lie on the path $rTu$. Let $s$ be the last common vertex of these two paths. Again, because $f(u) < f(v)$, the proper descendants of $s$ on the path $rTv$ could have been added to the stack $S$ only after all the proper descendants of $s$ on the path $rTu$ had been removed from it (thereby leaving $s$ as top vertex). In particular, $v$ could only have been added to $S$ after $u$ had been removed, so $l(u) < f(v)$. Because $f(v) < l(v)$, we conclude that $l(u) < l(v)$. □

We saw earlier (in Theorem 6.2b) that BFS-trees are characterized by the property that every edge of the graph joins vertices on the same or consecutive levels. The quintessential property of DFS-trees is described in the following theorem.

**Theorem 6.6** *Let $T$ be a DFS-tree of a graph $G$. Then every edge of $G$ joins vertices which are related in $T$.*

**Proof** This follows almost immediately from Proposition 6.5. Let $uv$ be an edge of $G$. Without loss of generality, suppose that $f(u) < f(v)$. By Proposition 6.5a, $l(v) < l(u)$. Now Proposition 6.5b implies that $u$ is an ancestor of $v$, so $u$ and $v$ are related in $T$. □

### Finding the Cut Vertices and Blocks of a Graph

In a graph which represents a communications network, the cut vertices of the graph correspond to centres whose breakdown would disrupt communications. It is thus important to identify these sites, so that precautions may be taken to reduce the vulnerability of the network. Tarjan (1972) showed how this problem can be solved efficiently by means of depth-first search.

While performing a depth-first search of a graph $G$, it is convenient to orient the edges of $G$ with respect to the DFS-tree $T$. We orient each tree edge from parent to child, and each nontree edge (whose ends are related in $T$, by Theorem 6.6) from descendant to ancestor. The latter edges are called *back edges*. The following characterization of cut vertices is an immediate consequence of Theorem 6.6.

**Theorem 6.7** *Let $T$ be a DFS-tree of a connected graph $G$. The root of $T$ is a cut vertex of $G$ if and only if it has at least two children. Any other vertex of $T$ is a cut vertex of $G$ if and only if it has a child no descendant of which dominates (by a back edge) a proper ancestor of the vertex.* □

Let us see how depth-first search may be used to find the cut vertices and blocks of a (connected) graph in linear time; that is, in time proportional to the number of edges of the graph.

Let $T$ be a DFS-tree of a connected graph $G$, and let $B$ be a block of $G$. Then $T \cap B$ is a tree in $G$ (Exercise 5.2.8b). Moreover, because $T$ is a rooted tree, we may associate with $B$ a unique vertex, the root of the tree $T \cap B$. We call this vertex

the *root* of $B$ with respect to $T$. It is the first vertex of $B$ to be incorporated into $T$. Note that the cut vertices of $G$ are just the roots of blocks (with the exception of $r$, if it happens to be the root of a single block). Thus, in order to determine the cut vertices and blocks of $G$, it suffices to identify these roots. It turns out that one can do so during the execution of depth-first search.

To this end, we consider the function $f^* : V \to \mathbb{N}$ defined as follows. If some proper ancestor of $v$ can be reached from $v$ by means of a directed path consisting of tree edges (possibly none) followed by one back edge, $f^*(v)$ is defined to be the least $f$-value of such an ancestor; if not, we set $f^*(v) := f(v)$. Observe, now, that a vertex $v$ is the root of a block if and only if it has a child $w$ such that $f^*(w) \geq f(v)$.

The function $f^*$ can be computed while executing depth-first search (see Exercise 6.1.12), and the criterion for roots of blocks may be checked at the same time. Thus the roots of the blocks of $G$, as well as the blocks themselves, can be determined in linear time.

The roots of the blocks of a graph with respect to a DFS-tree are shown in Figure 6.4. The pair $(f(v), l(v))$ is given for each vertex $v$. We leave it to the reader to orient the edges of $G$ as described above, and to compute the function $f^*$.

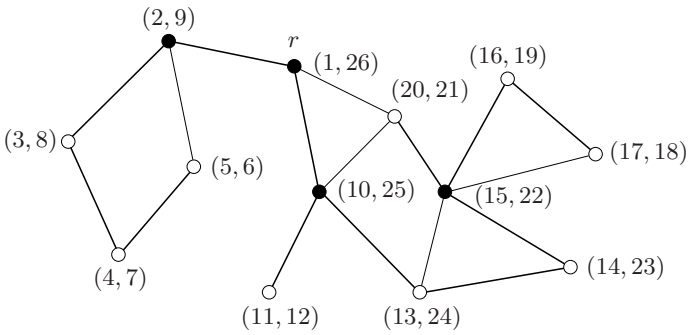



**Fig. 6.4.** Finding the cut vertices and blocks of a graph by depth-first search

## Exercises

⋆**6.1.1** Let $T$ be a BFS-tree of a connected graph $G$. Show that $\ell(v) = d_T(r, v)$, for all $v \in V$.

**6.1.2**

a) Let $T$ be a BFS-tree of a connected graph $G$ and let $z$ denote the last vertex to enter $T$. Show that $T - z$ is a BFS-tree of $G - z$.
b) Using (a), give inductive proofs of Theorems 6.2 and 6.3.

**6.1.3** Refine Algorithm 6.1 (breadth-first search) so that it returns either a bipartition of the graph (if the graph is bipartite) or an odd cycle (if it is not).

**6.1.4** Describe an algorithm based on breadth-first search for finding a shortest odd cycle in a graph.

**6.1.5** Let $G$ be a Moore graph (defined in Exercise 3.1.12). Show that all BFS-trees of $G$ are isomorphic.

**6.1.6** Let $T$ be a DFS-tree of a nontrivial connected simple graph $G$, and let $v$ be the root of a block $B$ of $G$. Show that the degree of $v$ in $T \cap B$ is one.

———————≀———————

**6.1.7** For a connected graph $G$, define $\sigma(G) := \sum\{d(u,v) : u, v \in V\}$.

a) Let $G$ be a connected graph. For $v \in V$, let $T_v$ be a breadth-first search tree of $G$ rooted at $v$. Show that $\sum_{v \in V} \sigma(T_v) = 2(n-1)\sigma(G)$.
b) Deduce that every connected graph $G$ has a spanning tree $T$ such that $\sigma(T) \leq 2(1 - \frac{1}{n})\sigma(G)$. (R.C. Entringer, D.J. Kleitman and L. Székely)

**6.1.8** Let $T$ be a rooted tree. Two breadth-first searches of $T$ (starting at its root) are distinct if their time functions $t$ differ. Likewise, two depth-first searches of $T$ are distinct if at least one of their time functions $f$ and $l$ differ. Show that the number of distinct breadth-first searches of the tree $T$ is equal to the number of distinct depth-first searches of $T$, and that this number is precisely $\prod\{n(v)! : v \in V(T)\}$, where $n(v)$ is the number of children of $v$ in $T$ (and $0! = 1$).

**6.1.9** Let $G$ be a connected graph, let $x$ be a vertex of $G$, and let $T$ be a spanning tree of $G$ which maximizes the function $\sum\{d_T(x,v) : v \in V\}$. Show that $T$ is a DFS-tree of $G$. (Zs. Tuza)

⋆**6.1.10** Let $G$ be a connected graph in which every DFS-tree is a Hamilton path (rooted at one end). Show that $G$ is a cycle, a complete graph, or a complete bipartite graph in which both parts have the same number of vertices.
(G. Chartrand and H.V. Kronk)

**6.1.11** Chord of a Cycle
A *chord* of a cycle $C$ in a graph $G$ is an edge in $E(G) \setminus E(C)$ both of whose ends lie on $C$. Let $G$ be a simple graph with $m \geq 2n - 3$, where $n \geq 4$. Show that $G$ contains a cycle with at least one chord. (L. Pósa)

⋆**6.1.12**

a) Let $G$ be a connected graph and $T$ a DFS-tree of $G$, where the edges of $T$ are oriented from parent to child, and the back edges from descendant to ancestor. For $v \in V$, set:

$$g(v) := \min\{f(w) : (v,w) \in E(G) \setminus E(T)\}$$
$$h(v) := \min\{f^*(w) : (v,w) \in E(T)\}$$

Show that:

i) the function $f^*$ may be computed recursively by the formula

$$f^*(v) = \min\{f(v), g(v), h(v)\}$$

ii) a nonroot vertex $v$ of $T$ is a cut vertex of $G$ if and only if $f(v) \leq h(v)$.

b) Refine Algorithm 6.4 (Depth-First Search) so that it returns the cut vertices and the blocks of a connected graph. (R.E. Tarjan)

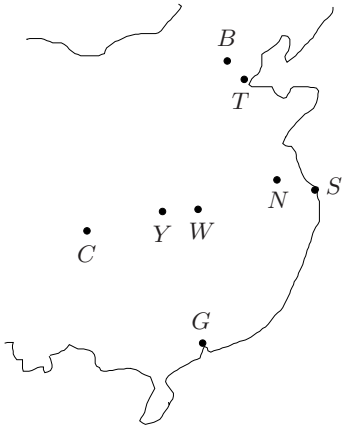
**6.1.13** Let $G$ be a simple connected graph, and let $w : V \to \mathbb{Z}$ be a weight function on $V$ such that $\sum_{v \in V} w(v) \geq m - n + 1$. For $X \subset V$, the *move* $M_X$ consists of distributing a unit weight from each vertex of $X$ to each of its neighbours in $V \setminus X$ (so that the weight of a vertex $v$ of $V \setminus X$ increases by $d_X(v)$).

a) Show that the weight can be made nonnegative at each vertex by means of a sequence of moves.

b) Show that this is no longer necessarily true if $\sum_{v \in V} w(v) \leq m - n$. (M. Baker and S. Norine)

## 6.2 Minimum-Weight Spanning Trees

An electric grid is to be set up in China, linking the cities of Beijing, Chongqing, Guangdong, Nanjing, Shanghai, Tianjin, and Wuhan to the Three Gorges generating station situated at Yichang. The locations of these cities and the distances (in kilometres) between them are given in Figure 6.5. How should the grid be constructed so that the total connection distance is as small as possible?




| | C | G | N | S | T | W | Y |
|---|---|---|---|---|---|---|---|
| B | 1457 | 1892 | 901 | 1078 | 111 | 1057 | 1117 |
| C | – | 978 | 1199 | 1430 | 1442 | 750 | 473 |
| G | – | – | 1133 | 1197 | 1820 | 837 | 867 |
| N | – | – | – | 267 | 800 | 459 | 727 |
| S | – | – | – | – | 970 | 681 | 962 |
| T | – | – | – | – | – | 988 | 1080 |
| W | – | – | – | – | – | – | 285 |

**Fig. 6.5.** The China hydro-electric grid problem

The table in Figure 6.5 determines a weighted complete graph with vertices $B$, $C$, $G$, $N$, $S$, $T$, $W$, and $Y$. Our problem amounts to finding, in this graph, a connected spanning subgraph of minimum weight. Because the weights are positive, this subgraph will be a spanning tree.

More generally, we may consider the following problem.

**Problem 6.8** Minimum-Weight Spanning Tree
Given*: a weighted connected graph $G$,*
Find*: a minimum-weight spanning tree $T$ in $G$.*

For convenience, we refer to a minimum-weight spanning tree as an *optimal tree*.

### The Jarník–Prim Algorithm

The Minimum-Weight Spanning Tree Problem (6.8) can be solved by means of a tree-search due to Jarník (1930) and Prim (1957). In this algorithm, which we call the *Jarník–Prim Algorithm*, an arbitrary vertex $r$ is selected as the root of $T$, and at each stage the edge added to the current tree $T$ is any edge of least weight in the edge cut associated with $T$.

As in breadth-first and depth-first search, the vertices of $T$ are coloured black. Also, in order to implement the above tree-search efficiently, each uncoloured vertex $v$ is assigned a provisional cost $c(v)$. This is the least weight of an edge linking $v$ to some black vertex $u$, if there is such an edge, in which case we assign $u$ as provisional predecessor of $v$, denoted $p(v)$. Initially, each vertex has infinite cost and no predecessor. These two provisional labels are updated at each stage of the algorithm.

**Algorithm 6.9** The Jarník–Prim Algorithm

Input*:* a weighted connected graph $(G, w)$
Output*:* an optimal tree $T$ of $G$ with predecessor function $p$, and its weight $w(T)$

*1:* *set $p(v) := \emptyset$ and $c(v) := \infty$, $v \in V$, and $w(T) := 0$*
*2:* *choose any vertex $r$ (as root)*
*3:* *replace $c(r)$ by $0$*
*4:* ***while** there is an uncoloured vertex **do***
*5:* *choose such a vertex $u$ of minimum cost $c(u)$*
*6:* *colour $u$ black*
*7:* ***for** each uncoloured vertex $v$ such that $w(uv) < c(v)$ **do***
*8:* *replace $p(v)$ by $u$ and $c(v)$ by $w(uv)$*
*9:* *replace $w(T)$ by $w(T) + c(u)$*
*10:* ***end for***
*11:* ***end while***
*12:* *return $(p, w(T))$*

In practice, the set of uncoloured vertices and their costs are kept in a structure called a *priority queue*. Although this is not strictly a queue as defined earlier, the vertex of minimum cost is always located at the head of the queue (hence the 'priority') and can therefore be accessed immediately. Furthermore, the 'queue' is structured so that it can be updated rather quickly when this vertex is removed (coloured black), or when the costs are modified (as in line 9 of the Jarník–Prim Algorithm). As to how this can be achieved is outlined in Section 6.4.

We call a rooted spanning tree output by the Jarník–Prim Algorithm a *Jarník–Prim tree*. The construction of such a tree in the electric grid graph is illustrated (not to scale) in Figure 6.6, the edges being numbered according to the order in which they are added.

In step 1, Yichang (Y) is chosen as the root. No vertex has yet been coloured. Because $c(Y) = 0$, and $c(v) = \infty$ for every other vertex $v$, vertex $Y$ is chosen as $u$ in step 2, and coloured black. All uncoloured vertices are assigned $Y$ as predecessor, and their costs are reduced to:

$$c(B) = 1117, \quad c(C) = 473, \quad c(G) = 867$$

$$c(N) = 727, \quad c(S) = 962, \quad c(T) = 1080, \quad c(W) = 285$$

The weight of the tree $T$ remains zero.

In the second iteration of step 2, $W$ is selected as the vertex $u$ and coloured black. The predecessors of the uncoloured vertices, and their costs, become:

$$p(B) = W, \quad p(C) = Y, \quad p(G) = W, \quad p(N) = W, \quad p(S) = W, \quad p(T) = W$$

$$c(B) = 1057, \quad c(C) = 473, \quad c(G) = 837, \quad c(N) = 459, \quad c(S) = 681, \quad c(T) = 988$$

and $w(T)$ is increased to 285.

In the third iteration of step 2, $N$ is selected as the vertex $u$ and coloured black. The predecessors of the uncoloured vertices, and their costs, become:

$$p(B) = N, \quad p(C) = Y, \quad p(G) = W, \quad p(S) = N, \quad p(T) = N$$

$$c(B) = 901, \quad c(C) = 473, \quad c(G) = 837, \quad c(S) = 267, \quad c(T) = 800$$

and $w(T)$ is increased to $285 + 459 = 744$.

This procedure continues until all the vertices are coloured black. The total length of the grid thereby constructed is 3232 kilometres.

The following theorem shows that the algorithm runs correctly.

**Theorem 6.10** *Every Jarník–Prim tree is an optimal tree.*

**Proof** Let $T$ be a Jarník–Prim tree with root $r$. We prove, by induction on $v(T)$, that $T$ is an optimal tree. The first edge added to $T$ is an edge $e$ of least weight in the edge cut associated with $\{r\}$; in other words, $w(e) \leq w(f)$ for all edges $f$ incident with $r$. To begin with, we show that some optimal tree includes this edge $e$. Let $T^*$ be an optimal tree. We may assume that $e \notin E(T^*)$. Thus $T^* + e$

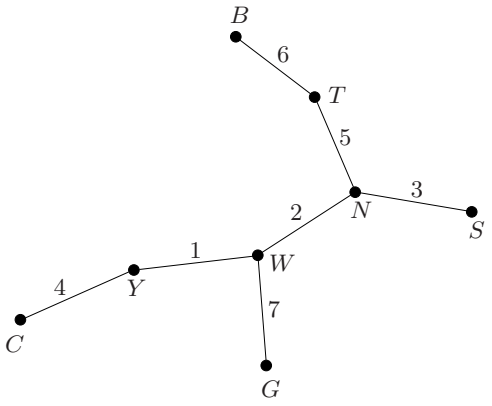



**Fig. 6.6.** An optimal tree returned by the Jarník–Prim Algorithm

contains a unique cycle $C$. Let $f$ be the other edge of $C$ incident with $r$. Then $T^{**} := (T^* + e) \setminus f$ is a spanning tree of $G$. Moreover, because $w(e) \leq w(f)$,

$$w(T^{**}) = w(T^*) + w(e) - w(f) \leq w(T^*)$$

As $T^*$ is an optimal tree, equality must hold, so $T^{**}$ is also an optimal tree. Moreover, $T^{**}$ contains $e$.

Now consider the graph $G' := G / e$, and denote by $r'$ the vertex resulting from the contraction of $e$. There is a one-to-one correspondence between the set of spanning trees of $G$ that contain $e$ and the set of all spanning trees of $G'$ (Exercise 4.2.1a). Thus, to show that the final tree $T$ is an optimal tree of $G$, it suffices to show that $T' := T / e$ is an optimal tree of $G'$. We claim that $T'$ is a Jarník–Prim tree of $G'$ rooted at $r'$.

Consider the *current* tree $T$ at some stage of the Jarník–Prim Algorithm. We assume that $T$ is not simply the root vertex $r$, and thus includes the edge $e$. Let $T' := T / e$. Then $\partial(T) = \partial(T')$, so an edge of minimum weight in $\partial(T)$ is also an edge of minimum weight in $\partial(T')$. Because the *final* tree $T$ is a Jarník–Prim tree of $G$, we deduce that the final tree $T'$ is a Jarník–Prim tree of $G'$. As $G'$ has fewer vertices than $G$, it follows by induction that $T'$ is an optimal tree of $G'$. We conclude that $T$ is an optimal tree of $G$. □

The history of the Jarník–Prim Algorithm is described by Korte and Nešetřil (2001). A second algorithm for solving Problem 6.8, based on another approach, is presented in Section 8.5.

## Exercises

⋆**6.2.1** Let $(G, w)$ be a weighted connected graph whose edges have distinct weights. Show that $G$ has a unique optimal tree.

**6.2.2** Let $(G, w)$ be a weighted connected graph. Show that a spanning tree $T$ of $G$ is optimal if and only if, for each edge $e \in E \setminus T$ and each edge $f \in C_e$ (the fundamental cycle of $G$ with respect to $T$), $w(e) \geq w(f)$.

**6.2.3** Let $(G, w)$ be a weighted connected graph. Show that a spanning tree $T$ of $G$ is optimal if and only if, for each edge $e \in T$ and each edge $f \in B_e$ (the fundamental bond of $G$ with respect to $T$), $w(e) \leq w(f)$.

**6.2.4** Let $(G, w)$ be a weighted connected graph (with positive weights). Describe an algorithm for finding a spanning tree the product of whose weights is minimum.

**6.2.5** Let $T$ be an optimal spanning tree in a weighted connected graph $(G, w)$, and let $x$ and $y$ be two vertices of $G$. Show that the path $xTy$ is an $xy$-path of minimum weight in $G$.

**6.2.6** Let $T$ be an optimal spanning tree in a weighted connected graph $(G, w)$.

a) Show that $T$ is a spanning tree whose largest edge-weight is minimum.
b) Give an example of a weighted connected graph $(G, w)$ and a spanning tree $T$ of $G$ whose largest edge-weight is minimum, but which is not an optimal spanning tree of $G$.

## 6.3 Branching-Search

One can explore directed graphs in much the same way as undirected graphs, but by growing branchings rather than rooted trees. Starting with the branching consisting of a single vertex $r$, its *root*, one adds one arc at a time, together with its head, the arc being selected from the outcut associated with the current branching. The procedure terminates either with a spanning branching of the digraph or with a nonspanning branching whose associated outcut is empty. Note that the latter outcome may well arise even if the digraph is connected. Indeed, the vertex set of the final branching is precisely the set of vertices of the digraph that are reachable by directed paths from $r$. We call the above procedure *branching-search*.

As with tree-search, branching-search may be refined by restricting the choice of the arc to be added at each stage. In this way, we obtain directed versions of breadth-first search and depth-first search. We discuss two important applications of branching-search. The first is an extension of directed BFS to weighted directed graphs, the second an application of directed DFS.

### Finding Shortest Paths in Weighted Digraphs

We have seen how breadth-first search can be used to determine shortest paths in graphs. In practice, one is usually faced with problems of a more complex nature. Given a one-way road system in a city, for instance, one might wish to determine a

shortest route between two specified locations in the city. This amounts to finding a directed path of minimum weight connecting two specified vertices in the weighted directed graph whose vertices are the road junctions and whose arcs are the roads linking these junctions.

**Problem 6.11** SHORTEST PATH
GIVEN*: a weighted directed graph* $(D, w)$ *with two specified vertices* $x$ *and* $y$*,*
FIND*: a minimum-weight directed* $(x, y)$*-path in* $D$*.*

For clarity of exposition, we refer to the weight of a directed path in a weighted digraph as its *length*. In the same vein, by a *shortest* directed $(x, y)$-path we mean one of minimum weight, and this weight is the *distance* from $x$ to $y$, denoted $d(x, y)$. For example, the path indicated in the graph of Figure 6.7 is a shortest directed $(x, y)$-path (Exercise 6.3.1) and $d(x, y) = 3 + 1 + 2 + 1 + 2 + 1 + 2 + 4 = 16$. When all the weights are equal to one, these definitions coincide with the usual notions of length and distance.

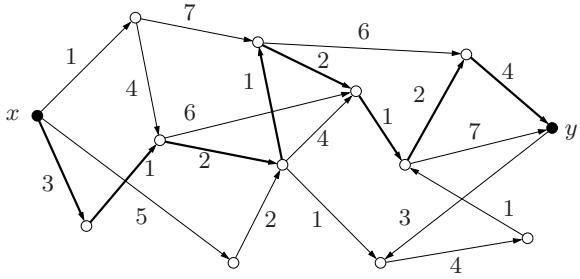



**Fig. 6.7.** A shortest directed $(x, y)$-path in a weighted digraph

It clearly suffices to deal with the shortest path problem for strict digraphs, so we assume that this is the case here. We also assume that all weights are positive. Arcs of weight zero can always be contracted. However, the presence of negative weights could well lead to complications. If the digraph should contain directed cycles of negative weight, there might exist $(x, y)$-walks which are shorter than any $(x, y)$-path — indeed, ones of arbitrarily small (negative) length — and this eventuality renders shortest path algorithms based on branching-search, such as the one described below, totally ineffective (see Exercise 6.3.3). On the other hand, when all weights are positive, the shortest path problem can be solved efficiently by means of a branching-search due to Dijkstra (1959).

Although similar in spirit to directed breadth-first search, *Dijkstra's Algorithm* bears a resemblance to the Jarník–Prim Algorithm in that provisional labels are assigned to vertices. At each stage, every vertex $v$ of the current branching $B$ is labelled by its predecessor in $B$, $p(v)$, and its distance from $r$ in $B$, $\ell(v) := d_B(r, v)$. In addition, each vertex $v$ which is not in $B$ but is an outneighbour of some vertex in $B$, is labelled with a provisional predecessor $p(v)$ and a provisional distance $\ell(v)$,

namely, the length of a shortest directed $(r, v)$-path in $D$ all of whose internal vertices belong to $B$. The rule for selecting the next vertex and edge to add to the branching depends only on these provisional distances.

**Algorithm 6.12** Dijkstra's Algorithm

Input: a positively weighted digraph $(D, w)$ with a specified vertex $r$
Output: an $r$-branching in $D$ with predecessor function $p$, and a function $\ell : V \to \mathbb{R}^+$ such that $\ell(v) = d_D(r, v)$ for all $v \in V$

*set* $p(v) := \emptyset$, $v \in V$, $\ell(r) := 0$, *and* $\ell(v) := \infty$, $v \in V \setminus \{r\}$
***while*** *there is an uncoloured vertex* $u$ *with* $\ell(u) < \infty$ ***do***
    *choose such a vertex* $u$ *for which* $\ell(u)$ *is minimum*
    *colour* $u$ *black*
    ***for*** *each uncoloured outneighbour* $v$ *of* $u$ *with* $\ell(v) > \ell(u) + w(u, v)$ ***do***
        *replace* $p(v)$ *by* $u$ *and* $\ell(v)$ *by* $\ell(u) + w(u, v)$
    ***end for***
***end while***
*return* $(p, \ell)$

Dijkstra's Algorithm, like the Jarník–Prim Algorithm, may be implemented by maintaining the uncoloured vertices and their distances in a priority queue. We leave it to the reader to verify that the algorithm runs correctly (Exercise 6.3.2).

### Directed Depth-First Search

Directed BFS (the unweighted version of Dijkstra's Algorithm) is a straightforward analogue of BFS; the labelling procedure is identical, and the branching-search terminates once all vertices reachable from the root have been found. Directed DFS, on the other hand, involves a slight twist: whenever the branching-search comes to a halt, an uncoloured vertex is selected and the search is continued afresh with this vertex as root. The end result is a spanning branching forest of the digraph, which we call a *DFS-branching forest*.

**Algorithm 6.13** Directed Depth-First Search (Directed DFS)

Input: a digraph $D$
Output: a spanning branching forest of $D$ with predecessor function $p$, and two time functions $f$ and $l$

*set* $i := 0$ *and* $S := \emptyset$
***while*** *there is an uncoloured vertex* ***do***
    *choose any uncoloured vertex* $r$ *(as root)*
    *increment* $i$ *by* 1
    *colour* $r$ *black*
    *set* $f(r) := i$
    *add* $r$ *to* $S$
    ***while*** $S$ *is nonempty* ***do***
        *consider the top vertex* $x$ *of* $S$

```
10:        increment i by 1
11:        if x has an uncoloured outneighbour y then
12:           colour y black
13:           set p(y) := x and f(y) := i
14:           add y as the top vertex of S
15:        else
16:           set l(x) := i
17:           remove x from S
18:        end if
19:     end while
20:  end while
21:  return (p, f, l)
```

Directed DFS has many applications. One is described below, and several others are outlined in exercises (6.3.6, 6.3.7, 6.3.8, 6.3.13). In these applications, it is convenient to distinguish three types of arcs of $D$, apart from those in the DFS-branching-forest $F$.

An arc $(u, v) \in A(D) \setminus A(F)$ is a *forward arc* if $u$ is an ancestor of $v$ in $F$, a *back arc* if $u$ is a descendant of $v$ in $F$, and a *cross arc* if $u$ and $v$ are unrelated in $F$ and $u$ was discovered after $v$. In terms of the time functions $f$ and $l$:

- ▷ $(u, v)$ is a *forward arc* if $f(u) < f(v)$ and $l(v) < l(u)$,
- ▷ $(u, v)$ is a *back arc* if $f(v) < f(u)$ and $l(u) < l(v)$,
- ▷ $(u, v)$ is a *cross arc* if $l(v) < f(u)$.

The directed analogue of Theorem 6.6, whose proof is left as an exercise (6.3.4), says that these arcs partition $A(D) \setminus A(F)$.

**Theorem 6.14** *Let $F$ be a DFS-branching forest of a digraph $D$. Then each arc of $A(D) \setminus A(F)$ is a forward arc, a back arc, or a cross arc.* □

### Finding the Strong Components of a Digraph

The strong components of a digraph can be found in linear time by using directed DFS. The basic idea is similar to the one employed for finding the blocks of an undirected graph, but is slightly more complicated.

The following proposition shows how the vertices of the strong components of $D$ are disposed in $F$. Observe that forward arcs play no role with respect to reachability in $D$ because any forward arc can be replaced by the directed path in $F$ connecting its ends. We may therefore assume that there are no such arcs in $D$.

**Proposition 6.15** *Let $D$ be a directed graph, $C$ a strong component of $D$, and $F$ a DFS-branching forest in $D$. Then $F \cap C$ is a branching.*

**Proof** Each component of $F \cap C$ is contained in $F$, and thus is a branching. Furthermore, vertices of $C$ which are related in $F$ necessarily belong to the same

component of $F \cap C$, because the directed path in $F$ connecting them is contained in $C$ also (Exercise 3.4.3).

Suppose that $F \cap C$ has two distinct components, with roots $x$ and $y$. As remarked above, $x$ and $y$ are not related in $F$. We may suppose that $f(x) < f(y)$. Because $x$ and $y$ belong to the same strong component $C$ of $D$, there is a directed $(x, y)$-path $P$ in $C$, and because $f(x) < f(y)$, there must be an arc $(u, v)$ of $P$ with $f(u) < f(y)$ and $f(v) \geq f(y)$. This arc can be neither a cross arc nor a back arc, since $f(u) < f(v)$. It must therefore be an arc of $F$, because we have assumed that there are no forward arcs. Therefore $l(v) < l(u)$. If $u$ and $y$ were unrelated, we would have $l(u) < f(y)$. But this would imply that $f(v) < l(v) < l(u) < f(y)$, contradicting the fact that $f(v) \geq f(y)$. We conclude that $u$ is a proper ancestor of $y$, and belongs to the same component of $F \cap C$ as $y$. But this contradicts our assumption that $y$ is the root of this component. □

By virtue of Proposition 6.15, we may associate with each strong component $C$ of $D$ a unique vertex, the root of the branching $F \cap C$. As with blocks, it suffices to identify these roots in order to determine the strong components of $D$. This can be achieved by means of a supplementary branching-search. We leave the details as an exercise (Exercise 6.3.12).

## Exercises

**6.3.1** By applying Dijkstra's Algorithm, show that the path indicated in Figure 6.7 is a shortest directed $(x, y)$-path.

⋆**6.3.2** Prove that Dijkstra's Algorithm runs correctly.

⋆**6.3.3** Apply Dijkstra's Algorithm to the directed graph with negative weights shown in Figure 6.8. Does the algorithm determine shortest directed $(r, v)$-paths for all vertices $v$?

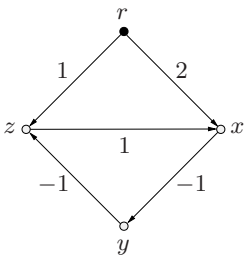



**Fig. 6.8.** Apply Dijkstra's Algorithm to this weighted directed graph (Exercise 6.3.3)

⋆**6.3.4** Prove Theorem 6.14.

**6.3.5** Describe an algorithm based on directed breadth-first search for finding a shortest directed odd cycle in a digraph.

**6.3.6** Describe an algorithm based on directed depth-first search which accepts as input a directed graph $D$ and returns a maximal (but not necessarily maximum) acyclic spanning subdigraph of $D$.

**6.3.7** Describe an algorithm based on directed depth-first search which accepts as input a tournament $T$ and returns a directed Hamilton path of $T$.

**6.3.8** Describe an algorithm based on directed depth-first search which accepts as input a directed graph $D$ and returns either a directed cycle in $D$ or a topological sort of $D$ (defined in Exercise 2.1.11).

**6.3.9** Bellman's Algorithm
Prove the validity of the following algorithm, which accepts as input a topological sort $Q$ of a weighted acyclic digraph $(D, w)$, with first vertex $r$, and returns a function $\ell : V \to \mathbb{R}$ such that $\ell(v) = d_D(r, v)$ for all $v \in V$, and a branching $B$ (given by a predecessor function $p$) such that $rBv$ is a shortest directed $(r, v)$-path in $D$ for all $v \in V$ such that $d_D(r, v) < \infty$. (R. Bellman)

set $\ell(v) := \infty$, $p(v) := \emptyset$, $v \in V$
remove $r$ from $Q$
set $\ell(r) := 0$
**while** $Q$ is nonempty **do**
  remove the first element $y$ from $Q$
  **for** all $x \in N^-(y)$ **do**
    **if** $\ell(x) + w(x, y) < \ell(y)$ **then**
      replace $\ell(y)$ by $\ell(x) + w(x, y)$ and $p(y)$ by $x$
    **end if**
  **end for**
**end while**
return $(\ell, p)$.

**6.3.10** Let $D := (D, w)$ be a weighted digraph with a specified root $r$ from which all other vertices are reachable. A *negative* directed cycle is one whose weight is negative.

a) Show that if $D$ has no negative directed cycles, then there exists a spanning $r$-branching $B$ in $D$ such that, for each $v \in V$, the directed path $rBv$ is a shortest directed $(r, v)$-path in $D$.
b) Give an example to show that this conclusion need not hold if $D$ has negative directed cycles.

**6.3.11** Bellman–Ford Algorithm
Let $D := (D, w)$ be a weighted digraph with a specified root $r$ from which all other

vertices of $D$ are reachable. For each nonnegative integer $k$, let $d_k(v)$ denote the weight of a shortest directed $(r,v)$-walk using at most $k$ arcs, with the convention that $d_k(v) = \infty$ if there is no such walk. (Thus $d_0(r) = 0$ and $d_0(v) = \infty$ for all $v \in V \setminus \{r\}$.)

a) Show that the $d_k(v)$ satisfy the following recursion.

$$d_k(v) = \min\{d_{k-1}(v), \min\{d_{k-1}(u) + w(u,v) : u \in N^-(v)\}\}$$

b) For each of the weighted digraphs shown in Figure 6.9, compute $\mathbf{d}_k := (d_k(v) : v \in V)$ for $k = 0, 1, \ldots, 6$.

c) Show that:
   i) if $\mathbf{d}_k \neq \mathbf{d}_{k-1}$ for all $k$, $1 \leq k \leq n$, then $D$ contains a negative directed cycle,
   ii) if $\mathbf{d}_k = \mathbf{d}_{k-1}$ for some $k$, $1 \leq k \leq n$, then $D$ contains no negative directed cycle, and $d_k(v)$ is the distance from $r$ to $v$, for all $v \in V$.

d) In the latter case, describe how to find a spanning $r$-branching $B$ of $D$ such that, for each $v \in V$, the directed $(r,v)$-path in $B$ is a shortest directed $(r,v)$-path in $D$. (R. Bellman; L.R. Ford; E.F. Moore; A. Shimbel)

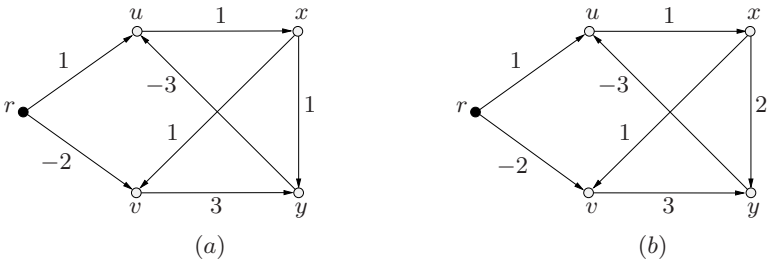



**Fig. 6.9.** Examples for the Bellman–Ford Algorithm (Exercise 6.3.11)

⋆**6.3.12** Finding the strong components of a digraph.
Let $D$ be a digraph, and let $F$ be a DFS-branching forest of $D$. Denote by $D'$ the converse of the digraph obtained from $D$ by deleting all cross edges. By Proposition 6.15, it suffices to consider each component of $D'$ separately, so we assume that $D'$ has just one component.

a) Show that the set of vertices reachable from the root $r$ in $D'$ induces a strong component of $D$.

b) Apply this idea iteratively to obtain all strong components of $D$ (taking care to select each new root appropriately).

c) Implement this procedure by employing branching-search.

**6.3.13** The *diameter* of a directed graph is the maximum distance between any two vertices of the graph. (Thus a directed graph is of finite diameter if and only if it is strong.) Let $G$ be a 2-edge-connected graph and $P$ a longest path in $G$. By Robbins' Theorem (5.10), $G$ has a strong orientation. Show that:

a) no strong orientation of $G$ has diameter exceeding the length of $P$,
b) some strong orientation of $G$ has diameter equal to the length of $P$.

(G. Gutin)

## 6.4 Related Reading

Data Structures

We have discussed in this chapter algorithms for resolving various problems expeditiously. The efficiency of these algorithms can be further enhanced by storing and managing the data involved in an appropriate structure. For example, a data structure known as a *heap* is commonly used for storing elements and their associated values, called *keys* (such as edges and their weights). A heap is a rooted binary tree $T$ whose vertices are in one-to-one correspondence with the elements in question (in our case, vertices or edges). The defining property of a heap is that the key of the element located at vertex $v$ of $T$ is required to be at least as large as the keys of the elements located at vertices of the subtree of $T$ rooted at $v$. This condition implies, in particular, that the key of the element at the root of $T$ is one of greatest value; that element can thus be accessed instantly. Moreover, heaps can be reconstituted rapidly following small modifications such as the addition of an element, the removal of an element, or a change in the value of a key. A *priority queue* (the data structure used in both Dijkstra's Algorithm and the Jarník–Prim Algorithm) is simply a heap equipped with procedures for performing such readjustments rapidly. Heaps were conceived by Williams (1964).

It should be evident that data structures play a vital role in the efficiency of algorithms. For further information on this topic, we refer the reader to Knuth (1969), Aho et al. (1983), Tarjan (1983), or Cormen et al. (2001).