Chapter 1
Minimum Weight Paths and Spanning Trees

1.1 Basic Ideas

Suppose that $\mathcal{G}$ is a directed graph with one node s singled out as the *source*. We assume also that each (directed) arc uv of $\mathcal{G}$ has a weight $c(uv)$ assigned to it. This "weight" function can represent any number of different parameters of interest: cost of using this arc, transmission speed of this link or even the physical distance between nodes. Since the weight of an arc may represent the changes we plan to make in another parameter, we allow both negative and positive weights. Figure 1.2 illustrates such a directed graph.

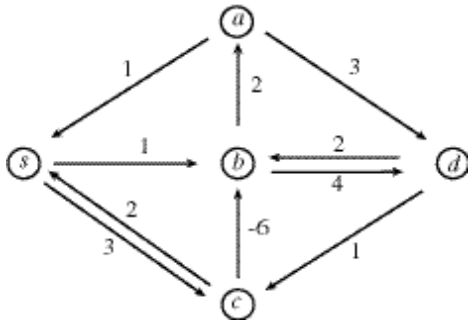

Figure 1.2 A directed graph with edge weights

If there is a sequence of edges leading from the source s to another node v then v is *reachable* from s. In Figure 1.2 all nodes are reachable from s, but if the direction of arc ab were reversed then node a would no longer be reachable. The *weight* of a path P from s to v is simply the sum $c(P)$ of the weights of the edges used in the path. So in Figure 1.2, the path s, b, a, d has weight $1 + 2 + 3 = 6$.

With this backgound, we can state the basic problem of this section. Given a directed graph $\mathcal{G}$ with source s and weight function c determine, for each reachable node v, the minimum weight of an $s - v$ path and a path that realizes this weight. This is the *single source problem*. It is often formulated with the weights being lengths, as the *Shortest Path Problem*

We should declare immediately that it is not always possible to find minimum weight paths. Figure 1.2 illustrates the difficulty. This graph contains a cycle b, d, c, b with weight $4 + 1 - 6 = -1$. Since this cycle has negative weight, by going around this cycle repeatedly, we can construct paths from s to b with lower and lower cost; there is no minimum weight for an $s - b$ path in this graph. In fact, negative cost cycles are the only obstruction.

THEOREM 1.1 *Suppose that $\mathcal{G}$ is a directed graph with source node s and weight function c defined on the arcs of $\mathcal{G}$. If in addition $\mathcal{G}$ contains no negative weight cycles, then there is a minimum weight path from s*

to each reachable node. □

EXERCISE

1.1 Prove Theorem 1.1. [Hint: Show that under the hypotheses, a simple $s - v$ path can not have more weight than an $s - v$ path containing a cycle.]

The idea of the algorithms developed in this section is to maintain a set of node labels $y(v)$ with the property that if $y(v)$ is finite then it represents the weight of some $s - v$ path. Initially we set $y(s) = 0$ and $y(v) = \infty$ for all other nodes v. (For purposes of arithmetic here, $\infty + a = \infty$ for any a.) Suppose we examine an arc uv where $y(u)$ is finite. SInce $y(u)$ is the weight of some $s - u$ path the sum $y(u) + c(uv)$ represents the cost of a path from s to v. If $y(u) + c(uv) < y(v)$ then this new path has lower weight than whatever path was used to determine $y(v)$. Since we are looking for low weight paths, we redefine $y(v) = y(u) + c(uv)$.

To be able to reconstruct the path, when an update $y(v) = y(u) + c(uv)$ is made, we record the fact that u is the node directly before v in this $s - v$ path; so the predecessor $p(v) = u$. Knowing these predecessors is enough to reconstruct the $s - v$ path (if there is one). These operations are collected into a basic update step performed on a single arc.

Update(uv): if $y(u) + c(uv) < y(v)$ then set $\{y(v) = y(u) + c(uv), p(v) = u\}$.

The general approach of the algorithm is to perform update steps on the arcs of $\mathcal{G}$ until

$$\text{every arc satisfies } y(u) + c(uv) \geq y(v). \tag{1.1}$$

Once this situation is achieved, the finite values of $y(v)$ give the minimum weights and the function $p(v)$ constructs the shortest paths. Observe though that Condition (1.1) can be satisfied with some infinite values remaining for node labels $y(w)$. This indicates that the node w has not yet been reached by the algorithm and, at termination, it means that w is not reachable in $\mathcal{G}$.

THEOREM 1.2 *If Condition (1.1) is satisfied in $\mathcal{G}$ then there are no negative weight cycles in $\mathcal{G}$.*

Proof. Suppose that $v_0, v_1, v_2, \ldots, v_k = v_0$ is a directed cycle in $\mathcal{G}$ and that $\mathcal{G}$ satisfies Condition (1.1). Then

$$\begin{aligned} c(v_0v_1) &\geq y(v_1) - y(v_0) \\ c(v_1v_2) &\geq y(v_2) - y(v_1) \\ c(v_2v_3) &\geq y(v_3) - y(v_2) \\ &\vdots \\ c(v_{k-1}v_0) &\geq y(v_0) - y(v_{k-1}) \end{aligned}$$

Summing these inequalities, the terms on the right collapse to 0 while those on the left determine the weight of the cycle. We conclude that any cycle in $\mathcal{G}$ must have non-negative weight. □

This result suggests that we should have our algorithm repeatedly update arcs until there are no arcs left to work on. The missing ingredient is the order in which we will test the arcs. Different algorithms

can be constructed by specifying different orders for checking the arcs. One approach is to visit a node and update all of the arcs leading away from that node before moving to a new node. This process is formallized in the procedure **Scan(−)**.

Scan(v): for all arcs vw, Update (uw).

1.2 Non-negative Weights

One of the algorithms that many students encounter early in their studies is the Shortest Path algorithm of Dijkstra. This is the important special case where the weights are thought of as lengths and all lengths are non-negative. So in the world of Dijkstra's algorithm there can't be any negative weight cycles. The algorithm can easily be expressed using the procedures just defined: Among all vertices that have not yet been Scanned, choose a node v with minimum finite label $y(v)$ and perform **Scan(v)**. The algorithm ends when all vertices have been scanned.

Dijkstra's Algorithm

1.Initialize: for each $v \in V \setminus \{s\}$: Set $y(v) = \infty$ and $p(v) = undefined$. Set $y(s) = 0$ and $U = V$.

2. While U contains a node w with finite $y(w)$,

Choose $v \in U$ with minimum $y(v)$;

Scan(v);

Remove v from U.

When Dijkstra's Algorithm stops, the finite values $y(v)$ give the weight of a minimum weight $s-v$ path, the predecessor function $p(-)$ specifies the shortest paths and the nodes remaining in U are unreachable. We will postpone the analysis of the predecessor function till later. Our task here is to see why Dijkstra's Algorithm works. Since the values of the node labels are adjusted as the algorithm procedes, it is useful to define $y'(v)$ to be the value of $y(v)$ when node v is scanned.

Suppose that Dijkstra's Algorithm stops with Condition (1.1) unsatisfied. Then there is an arc uv such that

$$y(u) + c(uv) < y(v).$$

So $y(u)$ is finite and when u was scanned, $y'(u)+c(uv) \geq y(v)$. It must be that the value of $y(u)$ was reduced when a subsequent node w was scanned. When this happened the label of u was set to $y'(w)+c(wu) < y'(u)$. Since the weights on the arcs are non-negative, $y'(w) \leq y'(w) + c(wu)$. Combining the inequalities we have $y'(w) < y'(u)$ and node w scanned after u. This is impossible as we show in Theorem 1.3. This contradiction shows that, at termination, Condition (1.1) is indeed satisfied.

THEOREM 1.3 *Suppose that Dijkstra's Algorithm scans node u before node w. Then $y'(u) \leq y'(w)$*

Proof. Suppose that $y'(w) < y'(u)$ and that w is the earliest scanned node for which this happens. Thus $y'(z) \geq y'(u)$ for any node z scanned before w. When u was scanned $y'(u) \leq y(w)$ since we always choose an

unscanned node with minimum label. Since $y'(w) < y'(u)$, the label on w was reduced to $y'(w)$ when some other node z was scanned and this node z was scanned before w was scanned. Therefore $y'(z) \geq y'(u)$ and $y'(w) = y'(z) + c(zw)$. Since $c(zw) \geq 0$ we have $y'(z) \leq y'(w) < y'(u)$; a contradiction. □

1.3 The General Case

The simplest approach that will work in the general case – negative and positive weights allowed – is to simply update all the arcs $|V|$ times. This means that we update all the arcs once then run through them all again and so on. It turns out that this is enough to calculate the shortest paths if they exist.

Bellman-Ford Shortest Paths Algorithm

1. Initialize: for each $v \in V \setminus \{s\}$: Set $y(v) = \infty$ and $p(v) = undefined$. Set $y(s) = 0$.
2. For $i = 1$ to $|V|$, for every arc uv, Update uv.
3. If every arc satisfies $y(u) + c(uv) \geq y(v)$ then
(i) if $y(v)$ is finite it is the weight of a shortest path from s to v and $p(-)$ constructs the path;
(ii) if $y(v) = \infty$ then v is unreachable from s.
4. If after n iterations some arc satisfies $y(u) + c(uv) < y(v)$ then $\mathcal{G}$ contains a negative weight cycle.

If at anytime during execution of this algorithm an update makes $y(s) < 0$ then a negative weight cycle has been detected and the algorithm can stop. Hence we assume in the analysis that $y(s) = 0$ at all times. Note also that Step 4 simply declares that some negative weight cycle exists. Sometimes we actually want the cycle. We defer to later the question of how to locate at least one negative weight cycle in this case.

Why does the Bellman-Ford Algorithm work? Let $n = |V|$ and $E_1, E_2, \ldots, E_{n-1}$ denote $n-1$ complete copies of the arcs of $\mathcal{G}$. The algorithm updates the arcs of E_1 then E_2 and so on. If we now take any simple path P in $\mathcal{G}$, it will have at most $n-1$ arcs and we can find its first arc in E_1, second arc in E_2 and generally the i-th arc in E_i. This means that we are updating the arcs of P in order, with many other updates interspersed. This fact guarantees that when we finish $y(v)$ really is the weight of a shortest path.

To see this, consider a simple 2-arc path , say $a \to b \to c$. Suppose the arc ab is updated, making $c(ab) = y(b) - y(a)$, and that sometime later the next arc of the path bc is updated. By this time, the values of the labels y have been changed so we write $c(bc) = y'(c) - y'(b)$. What we know is that the y values at a node never increase so $y'(b) \leq y(b)$. Thus the weight of this path satisfies:

$$c(ab) + c(bc) = y(b) - y(a) + y'(c) - y'(b) = y'(c) + (y(b) - y'(b)) - y(a) \geq y'(c) - y(a).$$

The same argument will apply along any simple path as long as, amongst all the arc updates that are happening in the algorithm, we can select a sequence of updates that process the arcs of the path in order. Using the fact that $y(s) = 0$ we have the following result.

THEOREM 1.3 *Suppose that P is a simple $s - v$ path and that the arcs of this path are updated in order (though not necessarily consecutively). Then the weight $c(P)$ of the path satisfies $c(P) \geq y(v)$.* □

What this means is that, if the graph $\mathcal{G}$ does not contain any negative weight cycles, then when the Bellman-Ford algorithm stops, $y(v)$ is the weight of a shortest $s - v$ path as required.

1.4 Predecessor Subgraph and Negative Weight Cycles

Consider now the predecessor function $p(-)$. From the algorithm, whenever $y(b)$ is finite $p(b)$ is defined. In this case, $p(b) = a$ where there is an arc from a to b and $p(a)$ is also defined. The exception is the source s where $y(s) = 0$ but there is no predecessor. The predecessor arcs $p(b)b$ form a subgraph on the reachable nodes with the property that each such node, except the source s, is the head of exactly one of these arcs. Such a subgraph is very restricted in structure. It must be a union of a directed tree rooted at s and a finite number of simple directed cycles which may have additional directed trees rooted at their nodes (Exercise 1.2.2). Figure 1.2.1 shows an example of such a subgraph.

EXERCISE

1.2 Suppose that $\mathcal{H}$ is a directed graph with a source node s such that each node of $\mathcal{H}$ other than s is the head of exactly one arc of $\mathcal{H}$. Prove that $\mathcal{H}$ is the union of a directed tree rooted at s and a finite number of simple directed cycles which may have additional directed trees attached at their nodes.

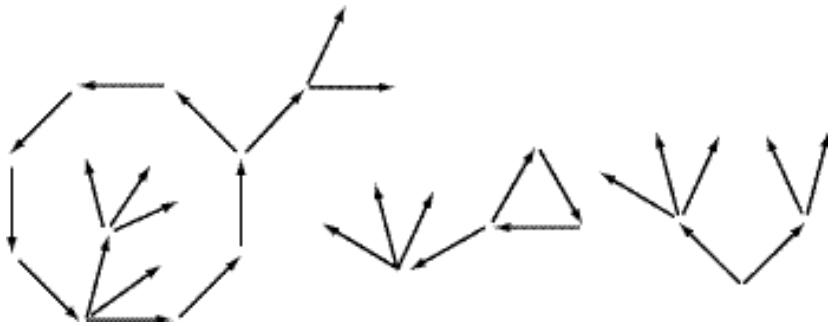

Figure 1.2.1 A possible predecessor subgraph.

Suppose that $p(b) = a$. Then when $p(b)$ was set equal to a also $y(b) = y(a) + c(a, b)$ was true. Have the values of $y(b)$ and $y(a)$ changed since then? If an update on another arc changes $y(b)$ then $p(b)$ would no longer equal a. So $y(b)$ has not changed. If $y(a)$ has changed then it is certainly no larger so we conclude that $y(b) \geq y(a) + c(a, b)$ is true whenever $p(b) = a$. That is,

$$\text{if } \; p(b) = a \text{ then } y(b) - y(a) \geq c(a, b).$$

Now suppose that the predecessor graph contains a cycle $v_0, v_1, v_2, \ldots, v_k = v_0$ with $p(v_i) = v_{i-1}$ for $i = 1, \ldots, k$. Suppose that the arc v_0v_1 was the last arc of the cycle to be updated. Freeze the values

of $y(v)$ one step before this arc v_0v_1 was updated so that the inequality $y(v_1) - y(v_0) > c(v_0, v_1)$ is strict. Then at this step

$$\begin{aligned} y(v_1) - y(v_0) &> c(v_0, v_1) \\ y(v_2) - y(v_1) &\geq c(v_1, v_2) \\ &\vdots \\ y(v_k) - y(v_{k-1}) &\geq c(v_{k-1}, v_k) \end{aligned}$$

Since $v_0 = v_k$ the left side sums to zero while the right side sums to the weight of the cycle. We conclude that any cycles in the predecessor subgraph represent negative weight cycles in the graph.

THEOREM 1.4 *If $\mathcal{G}$ has no negative weight cycles then the predecessor subgraph is a tree rooted at the source s and the paths in this tree are shortest paths. If $\mathcal{G}$ has a negative weight cycle then at the end of the Bellman–Ford algorithm, the predecessor subgraph will contain a cycle of negative weight.*

1.5 Examples

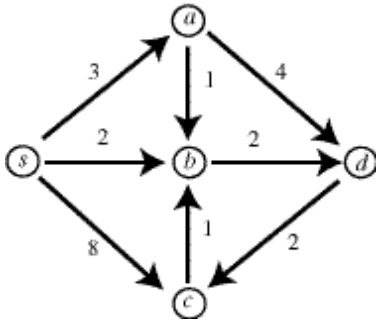

Figure 1.3 Example: non-negative weights

Line #	Scan node	Update arc	$y(a), p(a)$	$y(b), p(b)$	$y(c), p(c)$	$y(d), p(d)$
1	s	sa	$3, s$	$\infty, -$	$\infty, -$	$\infty, -$
2		sb	$3, s$	$2, s$	$\infty, -$	$\infty, -$
3		sc	$3, s$	$2, s$	$8, s$	$\infty, -$
4	b	bd	$3, s$	$2, s$	$8, s$	$4, b$
5	a	ab	$3, s$	$2, s$	$8, s$	$4, b$
6		ad	$3, s$	$2, s$	$8, s$	$4, b$
7	d	dc	$3, s$	$2, s$	$6, d$	$4, b$
8	c	cb	$3, s$	$2, s$	$6, d$	$4, b$

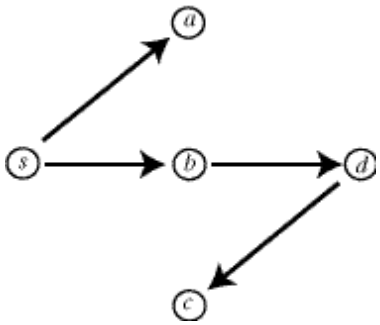

Figure 1.4 The Predecessor Subgraph of the example

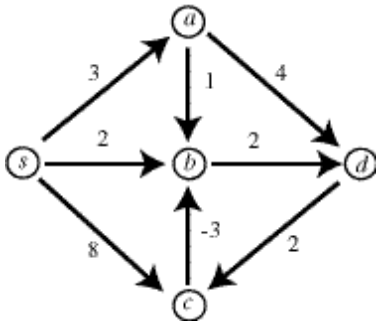

Figure 1.5 Example: positive and negative weights

Line #	Pass #	Update	$y(a), p(a)$	$y(b), p(b)$	$y(c), p(c)$	$y(d), p(d)$
1	*I*	*sa*	$3, s$	$\infty, -$	$\infty, -$	$\infty, -$
2		*sb*	$3, s$	$2, s$	$\infty, -$	$\infty, -$
3		*sc*	$3, s$	$2, s$	$8, s$	$\infty, -$
4		*ab*	$3, s$	$2, s$	$8, s$	$\infty, -$
5		*ad*	$3, s$	$2, s$	$8, s$	$7, a$
6		*bd*	$3, s$	$2, s$	$8, s$	$4, b$
7		*cb*	$3, s$	$2, s$	$8, d$	$4, b$
8		*dc*	$3, s$	$2, s$	$6, d$	$4, b$
9	*II*	*sa*	$3, s$	$2, s$	$6, d$	$4, b$
10		*sb*	$3, s$	$2, s$	$6, d$	$4, b$
11		*sc*	$3, s$	$2, s$	$6, d$	$4, b$
12		*ab*	$3, s$	$2, s$	$6, d$	$4, b$
13		*ad*	$3, s$	$2, s$	$6, d$	$4, b$
14		*bd*	$3, s$	$2, s$	$6, d$	$4, b$
15		*cb*	$3, s$	$2, s$	$6, d$	$4, b$
16		*dc*	$3, s$	$2, s$	$6, d$	$4, b$

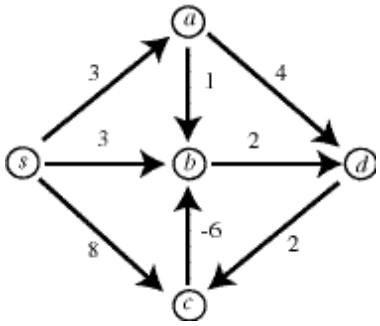

Figure 1.6 Example: negative weight cycle

Line #	Pass #	Update	$y(a), p(a)$	$y(b), p(b)$	$y(c), p(c)$	$y(d), p(d)$
1	*I*	*sa*	$3, s$	$\infty, -$	$\infty, -$	$\infty, -$
2		*sb*	$3, s$	$3, s$	$\infty, -$	$\infty, -$
3		*sc*	$3, s$	$3, s$	$8, s$	$\infty, -$
4		*ab*	$3, s$	$3, s$	$8, s$	$\infty, -$
5		*ad*	$3, s$	$3, s$	$8, s$	$7, a$
6		*bd*	$3, s$	$3, s$	$8, s$	$5, b$
7		*cb*	$3, s$	$2, c$	$8, d$	$5, b$
8		*dc*	$3, s$	$2, c$	$7, d$	$5, b$
9	*II*	*sa*	$3, s$	$2, c$	$7, d$	$5, b$
10		*sb*	$3, s$	$2, c$	$7, d$	$5, b$
11		*sc*	$3, s$	$2, c$	$7, d$	$5, b$
12		*ab*	$3, s$	$2, c$	$7, d$	$5, b$
13		*ad*	$3, s$	$2, c$	$7, d$	$5, b$
14		*bd*	$3, s$	$2, c$	$7, d$	$4, b$
15		*cb*	$3, s$	$1, c$	$7, d$	$4, b$
16		*dc*	$3, s$	$1, c$	$6, d$	$4, b$

On the next pass Condition (1.1) is still not satisfied: we would set $y(b) = 0, p(b) = c$. After the fourth pass through the arcs, Condition (1.1) would still not be satisfied. We conclude that this network contains a negative weight cycle on the accessible nodes. The predecessor graph is as follows. This is already clear at the end of the first pass.

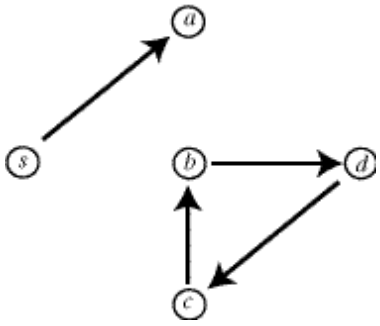

Figure 1.4 The Predecessor Subgraph of the example

1.6 Minimum Spanning Trees

Start with an undirected, connected graph $\mathcal{G}$. A *spanning tree of* $\mathcal{G}$ is a connected subgraph of $\mathcal{G}$ that simmultaneously contains all the nodes of $\mathcal{G}$ and contains no cycles (i.e. is *acyclic*). Figure 1.5 shows a graph on 10 nodes with a spanning tree indicated by dashed arcs.

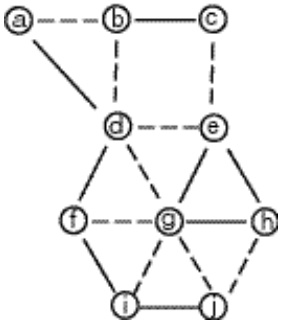

Figure 1.5 A graph with spanning tree marked by dashes.

Think about the complete graph for a moment and it will be clear that a graph can contain many different spanning trees. Once weights are added to the arcs it is natural to ask for the spanning tree of minimum weight for $\mathcal{G}$. This is the Minimum Spanning Tree Problem; a standard elementary problem in combinatorial optimization.

There are several distinct algorithms available for solving this problem. We will develop two here that illustrate the basic duality of optimization theory. We begin with an informal outline of each algorithm.

Algorithm A:

Work with the complete node set.

Start with no arcs selected.

Work through the arcs in order of increasing weight.

Add each arc to the subgraph $\mathcal{S}$ if and only if it does not form a cycle with the arcs already in $\mathcal{S}$.

Stop when no more arcs can be added (i.e. when all nodes are covered by the arcs in $\mathcal{S}$).

Algorithm B:

Work with the complete node set.

Start with all the arcs selected.

Work through the arcs in order of decreasing weight.

Toss out an arc from the subgraph $\mathcal{S}$ if and only if $\mathcal{S}$ stays connected when the arc is removed.

Stop when no more arcs can be removed (i.e. when there are no more cycles in $\mathcal{S}$).

In each case the algorithm maintains a set of arcs in a subgraph and works to modify this subgraph into a spanning tree. The method used guarantees that this spanning tree has minimum weight. In Algorithm A, we work with a subgraph $\mathcal{S}$ that is always acyclic and build it up using least weight arcs until it is connected and spans $\mathcal{G}$. In Algorithm B we proceed in the dual direction. The subgraph always spans $\mathcal{G}$ and we toss out heavy weight arcs until the graph is acyclic.

Figure 1.6 is a connected, undirected graph with arc weights. We will illustrate the two algorithms on this example.

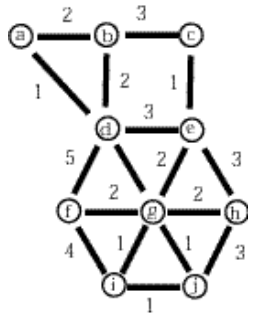

Figure 1.6 A graph with arc weights.

For Algorithm A we add the arcs in increasing order of weight. For the example the order is as folllows with the resulting minimum spanning tree shown in Figur 1.7.

arc	ce	ad	gi	gj	ab	eg	gh	fg	de
weight	1	1	1	1	2	2	2	2	3

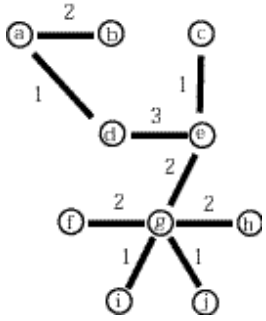

Figure 1.7 The minimum spanning tree found by Algorithm A.

For Algorithm B we toss out the arcs in decreasing order of weight. For the example the order is as folllows with the resulting minimum spanning tree shown in Figur 1.8.

arc	df	fi	de	eh	hj	dg	ab	gj
weight	5	4	3	3	3	3	2	1

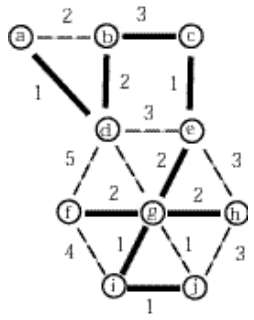

Figure 1.8 The minimum spanning tree found by Algorithm B.

Since the graph has the same weight on different arcs there is a choice in the actual arc order used by each algorithm. This means that the actual spanning tree produced might depend on the implementation and details of the algorithm used.

Next we will prove that Algorithms A and B actually work. Stopping is not really an issue since each works systematically through a finite set and hence can't run forever. But at the end, do we have a minimum spanning tree (MST)? To establish this, we will use a simple idea to move from one spanning tree to another.

Suppose that $\mathcal{T}$ is a spanning tree in a graph $\mathcal{G}$. Take any arc $e = uv$ of $\mathcal{G}$ which is not in $\mathcal{T}$. Since $\mathcal{T}$ spans $\mathcal{G}$, there is a unique path $\mathcal{P}$ in the tree $\mathcal{T}$ between u and v. This means that the subgraph $\mathcal{T} \cup \{e\}$ formed by adding the arc e to $\mathcal{T}$ contains a unique cycle formed by e and the path $\mathcal{P}$. If we now choose an arc f of $\mathcal{P}$ and remove it from $\mathcal{T} \cup \{e\}$, the cycle will be broken and we are left with another spanning tree. Since we will use this transformation repeatedly, lets denote by SWAP($\mathcal{T}$,e,f) the spanning tree obtained by removing f from and adding e to the spanning tree $\mathcal{T}$. So $\mathcal{T}$ becomes "$\mathcal{T} + e - f$". Also we will denote by $w(e)$ the weight on the arc e.

THEOREM 1.5 *Algorithm A produces a MST*

Proof. Suppose that Algorithm A selects the arcs in order $e_1, e_2, \ldots, e_{n-1}$ to produce a spanning tree $\mathcal{T}$. We show that the set of arcs choosen at each step is contained in some MST. In the end this can only be $\mathcal{T}$.

First we show that the arc e_1 is contained in some MST. If not then let $\mathcal{T}^*$ be a MST and look at the subgraph $\mathcal{T} \cup \{e\}$. This subgraph contains a unique cycle $\mathcal{C}$; let f be any arc of $\mathcal{T}^*$ in this cycle. Then SWAP($\mathcal{T}$,e,f) is a spanning tree with weight equal to $w(\mathcal{T}) - w(f) + w(e)$. Since e_1 has minimal weight by the specification of the algorithm, SWAP($\mathcal{T}$,e,f) is a MST containing e_1.

Choose k such that the set $\{e_1, \ldots, e_k\}$ is contained in a MST, say $\mathcal{T}^*$, but $\{e_1, \ldots, e_k, e_{k+1}\}$ is not contained in any MST. Then $\mathcal{T}^* \cup \{e_{k+1}\}$ contains a unique cycle and this cycle contains an arc f not in $\mathcal{T}$, the spanning tree produced by Algorithm A.

When e_{k+1} was chosen by Algorithm A, the arc f was a legal option. To see this, note that all of $e_1, \ldots, e_k, f$ are contained in the tree $\mathcal{T}^*$ so form an acyclic subgraph. Since f was not selected it must be that $w(f) \geq w(e_{k+1})$, else it would have been chosen. Then SWAP($\mathcal{T}^*$,e_{k+1},f) is a MST containing $\{e_1, \ldots, e_k, e_{k+1}\}$ contrary to the choice of k.

Therefore the full set of arcs, $\mathcal{T}$, chosen by Algorithm A is a spanning tree contained in a MST, so $\mathcal{T}$ is a MST. □

The proof of the correctness of Algorithm B is similar but dual to the proof just given.

THEOREM 1.6 *Algorithm B produces a MST*

Proof. Suppose that Algorithm B removes the arcs in order $e_1, e_2, \ldots, e_m$ to produce a spanning tree $\mathcal{T}$. We show that $\mathcal{T}$ is a MST by showing that the arcs remaining at each step contain some MST. At the end this can only be $\mathcal{T}$.

First we show that there is a MST that does not contain e_1. Let $\mathcal{T}^*$ be a MST containing e_1. Since Algorithm B removes e_1 it must be that removing e_1 does not disconnect the graph so e_1 is contained in a cycle $\mathcal{C}$ of $\mathcal{G}$. Let f be any other arc of $\mathcal{C}$. Since f is part of a cycle in $\mathcal{G}$ it was a candidate for removal. Since it was not removed, it follows that $w(f) \leq w(e_1)$. Then SWAP($\mathcal{T}^*$,f,e_1) has a weight no larger than $\mathcal{T}^*$, so is a MST not containing e_1 as required.

Choose k such that when Algorithm B removes the set $\{e_1, \ldots, e_k\}$, the remaining arcs contain a MST, but when e_{k+1} is removed there is no longer a MST in the remaining arcs. Thus there is a MST, say $\mathcal{T}^*$ not using any of the arcs in $\{e_1, \ldots, e_k\}$ but including the arc e_{k+1}. Since removing e_{k+1} does not disconnect the graph, it must be contained in a cycle $\mathcal{C}$ not using any of $e_1, \ldots, e_k$. Let f be any arc of $\mathcal{C}$ not contained in the tree $\mathcal{T}^*$. Because of the cycle $\mathcal{C}$ we know that removing f would not disconnect the graph. Since it was not removed, it follows that $w(f) \leq w(e_{k+1})$. Thus SWAP($\mathcal{T}^*$,f,e_{k+1}) has weight less than or equal to $w(\mathcal{T}^*)$. Thus SWAP($\mathcal{T}^*$,f,e_{k+1}) is a MST remaining after $\{e_1, \ldots, e_{k+1}\}$ is removed.

Therefore the arcs $\mathcal{T}$ remaining at the end are precisely a spanning tree containing a MST, so $\mathcal{T}$ is a MST. □

Algorithms A and B show that we can indeed find minimum spanning trees. It is clear from the proofs why they work and they also illustrate duality. As algorithms though they are not efficient. For example, in a straightforward approach to Algorithm A, we would sort the arcs ($|E| \log |E|$) into increasing weight and then process the arcs one at a time to decide if $\mathcal{S} \cup \{e\}$ contains a cycle. The overall complexity is too high. The proofs and the exampe make it clear that there is a lot of room to tighten the algorithm. For example, if there are n nodes in $\mathcal{G}$ then the MST will only contain $n - 1$ arcs. This is much less than the potential

number of arcs in $\mathcal{G}$ so perhaps we can find a refinement of Algorithm A that avoids looking at substantial numbers of arcs.

In Algorithm C, we will again work with arcs of increasing weight but not the whole arc set. The idea is to build a tree rather than simply an acyclic set of arcs. We start with any node v (which forms a subtree with no arcs). With a partially constructed tree $\mathcal{S}$, we find a node u not yet in the tree but joined to the tree by an arc of minimum weight amongst all arcs with one end in the tree. The key is that we work through the nodes not the arcs. To make this work we need a pointer $closest(u)$ which is a node of the growing tree $\mathcal{S}$ such that the arc between u and $closest(u)$ has minimum weight among the arcs joining u to $\mathcal{S}$. The weight of this arc is denoted $d(u)$.

Algorithm C:

Start with $\mathcal{S} = \{v\}$.

For each $u \neq v$ let $closest(u) = v$ and set $d(u) = w(uv)$ if there is such an arc and ∞ if there is not.

While nodes u remain outside $\mathcal{S}$:

Find u with minimum $d(u)$.

Add u and the arc between u and $closest(u)$ to $\mathcal{S}$.

Update the values of $closest(w)$ and $d(w)$ for nodes remaining outside $\mathcal{S}$.

EXERCISES

1.6.1 Prove that Algorithm C is correct.

1.6.2 Show that the complexity of Algorithm C is $O(n^2)$ where n is the number of nodes in $\mathcal{G}$.