# A Combinatorial Optimization Primer

Notes for Mathematics 69.382*

Winter Term 2001

Brian Mortimer

**Chapter 0 Introduction**

**Chapter 1 Minimum Weight Paths and Spanning Trees**

1.1 Basic Ideas

1.2 Non-negative Weights

1.3 The General Case

1.4 Predecessor Subgraph and Negative Weight Cycles

1.5 Examples

1.6 Minimum Spanning Trees

**Chapter 2 Maximum Flow in a Network**

2.1 Flows in Networks

2.2 Augmenting Paths

2.3 Cuts in a Network

2.4 The Augmenting Path Algorithm for Max-Flow

2.5 Implementation and Performance [Not yet Available]

2.6 Applications [Not yet Available]

**Interlude** Linear Programming Formulations of Some Combinatorial Problems

**Chapter 3 Maximum Flow by Pre-flow Push**

3.1 Pre-flows and Distance Labellings

3.2 Preflow Push Algorithms

3.3 Performance

**Chapter 4 Min-cost Flows**

4.1 Minimum Cost Flow Problems

4.2 The Incremental Network

4.3 Algorithm for Min-cost Flow

4.4 Another Min-cost Flow Algorithm

4.5 Application

# 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.

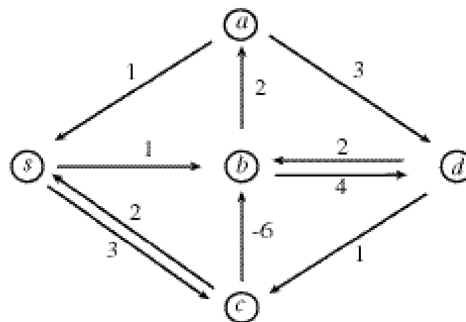



Figure 1.2 A directed graph with edge weights

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

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

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

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

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 $c(P) \geq y(v)$.*$\square$

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.

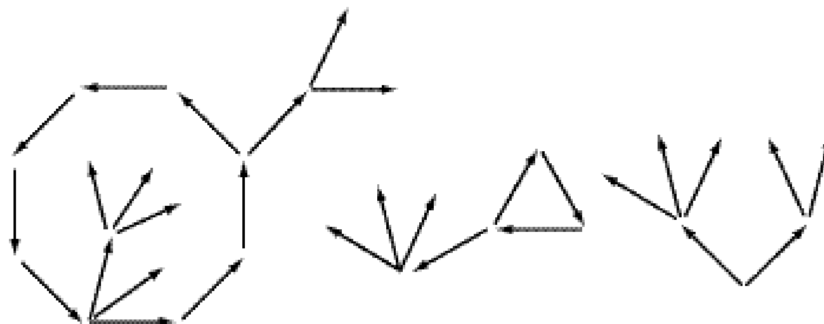


Figure 1.2.1 A possible predecessor subgraph.

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 $v_0 v_1$ 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

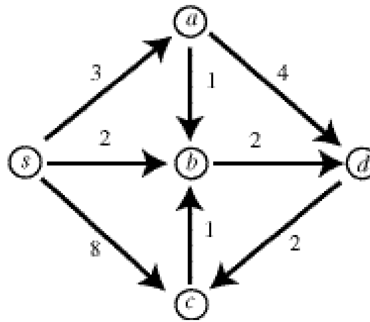



Figure 1.3 Example: non-negative weights

| 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 | $c$ | $cb$ | $3, s$ | $2, s$ | $6, d$ | $4, b$ |

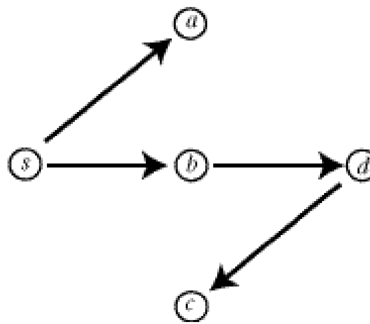



Figure 1.4 The Predecessor Subgraph of the example

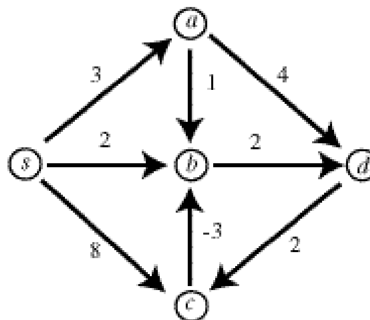



Figure 1.5 Example: positive and negative weights

| 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 | | $dc$ | $3, s$ | $2, s$ | $6, d$ | $4, b$ |

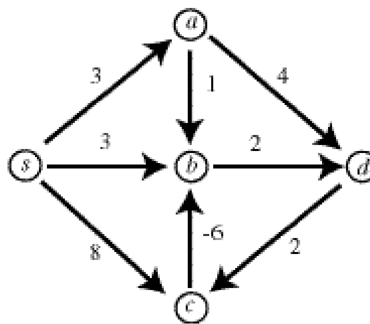



Figure 1.6 Example: negative weight cycle

| 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.

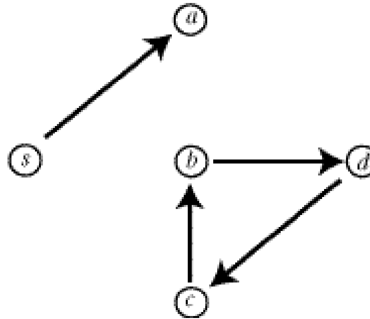



Figure 1.4 The Predecessor Subgraph of the example

## 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.

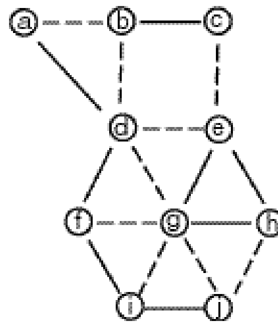



Figure 1.5 A graph with spanning tree marked by dashes.

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.

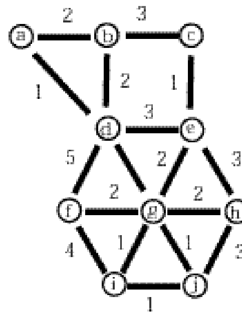



Figure 1.6 A graph with arc weights.

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 |

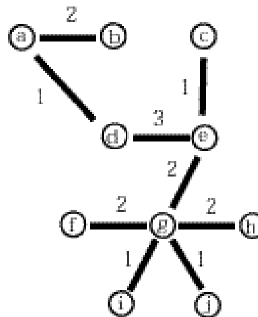


Figure 1.7 The minimum spanning tree found by Algorithm A.

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 |

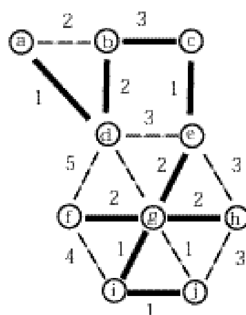



Figure 1.8 The minimum spanning tree found by Algorithm B.

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 $\infty$ 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 $\mathcal{G}$.

## 1.1 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 simultaneously 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 edges.

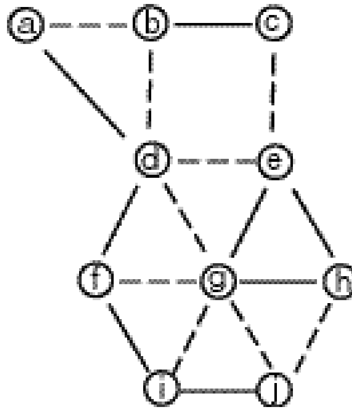



Figure 1.5 A graph with spanning tree marked by dashes.

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 edges 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 edges selected.

Work through the edges in order of increasing weight.

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

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

**Algorithm B:**

Work with the complete node set.

Start with all the edges selected.

Work through the edges in order of decreasing weight.

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

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

In each case the algorithm maintains a set of edges 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 edges 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 edges until the graph is acyclic.

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

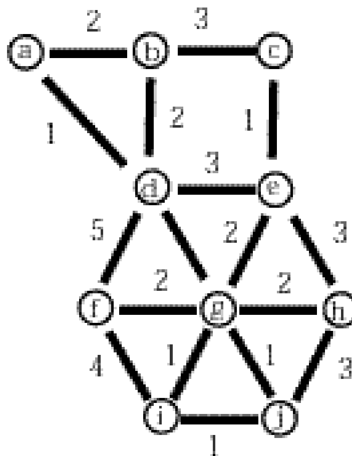



Figure 1.6 A graph with edge weights.

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

| edge | ce | ad | gi | gj | ab | eg | gh | fg | de |
|---|---|---|---|---|---|---|---|---|---|
| weight | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 |

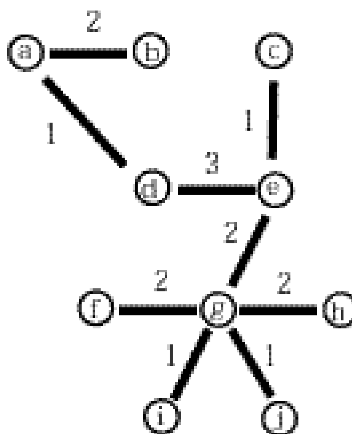



Figure 1.7 The minimum spanning tree found by Algorithm A.

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

| edge | df | fi | de | eh | hj | dg | ab | gj |
|---|---|---|---|---|---|---|---|---|
| weight | 5 | 4 | 3 | 3 | 3 | 3 | 2 | 1 |

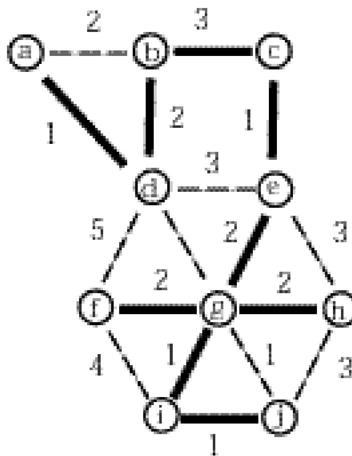



Figure 1.8 The minimum spanning tree found by Algorithm B.

Since the graph has the same weight on different edges there is a choice in the actual edge 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 edge $e = uv$ 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 edge $e$ to $\mathcal{T}$ contains a unique cycle formed by $e$ and the path $\mathcal{P}$. If we now choose an edge $f$ 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 $\mathcal{T}$. Also we will denote by $w(e)$ the weight on the edge $e$.

Theorem 1.5 *Algorithm A produces a MST*

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

First we show that the edge $e_1$ 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 edge 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 edge $f$ not in $\mathcal{T}$, the spanning tree produced by Algorithm A.

When $e_{k+1}$ was chosen by Algorithm A, the edge $f$ 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 edges, $\mathcal{T}$, 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 edges in order $e_1, e_2, \ldots, e_{n-1}$ to produce a spanning tree $\mathcal{T}$. We show that $\mathcal{T}$ is a MST by showing that the edges 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 edge 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 edges contain a MST, but when $e_{k+1}$ is removed there is no longer a MST in the remaining edges. Thus there is a MST, say $\mathcal{T}^*$ not using any of the edges in $\{e_1, \ldots, e_k\}$ but including the edge $e_{k+1}$. 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 edge 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 edges $\mathcal{T}$ remaining at the end are precisely a spanning tree containing a MST, so $\mathcal{T}$ is a MST. □

The 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 optimization duality. As algorithms though they are not efficient. For example, in a straightforward approach to Algorithm A, we would sort the edges ($|E| \log |E|$) into increasing weight and then process the edges 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 $n-1$ edges. This is much less than the potential number of edges in $\mathcal{G}$. Perhaps we can find a refinement of Algorithm A that avoids looking at substantial numbers of edges.

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

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

- Find $u$ with minimum $d(u)$.
- Add $u$ and the edge 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 $\mathcal{G}$.

# Chapter 2

# Maximum Flow in a Network

## 2.1 Flows in Networks

The model we use for a network consists of a set of nodes joined by directed edges as in Figure 2.1. Each edge $e$ of the network carries some amount of flow denoted $f_e$. The flow commodity could be transfer rate through a communications network, or merchandise moving from producer to consumer or simply water flowing in a pipe. Network models can be used in a vast array of applications.

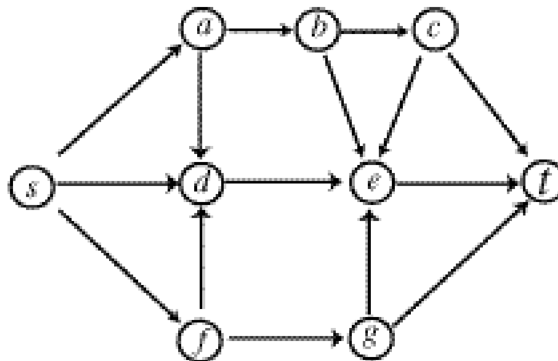



Figure 2.1 An Example of a Network

In the network there are two nodes with special roles: the *source* $s$ and the *sink* $t$. The network itself is a model of ways to arrange flow moving from node $s$ to node $t$. (This does not stop us from having edges directed into the source or out of the sink, though they may not be very helpful.)

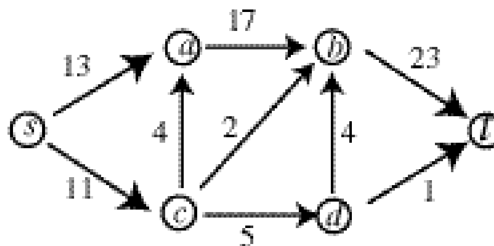



Figure 2.2 A Flow in a Network

There is one key condition that we place on any flow $F = \{f_e\}$ in a network. It is natural to assume that, except for the source and the sink, the nodes themselves neither create nor consume flow; they just pass it on. Thus we require "flow conservation" which means that, at each node $v \neq s, t$ the total flow into $v$ equals the total flow out of $v$. Any flow $F$ that satisfies this condition is called a *feasible flow*.

When discussing networks we will use the following notation. The network itself will be $N = (V, E)$ with node set $V$ and edge set $E$. A *flow* in $N$ will be denoted $F = \{f_e\}$ with $f_e$ being the flow in edge $e$. Each edge $e$ has a *tail* $t(e)$ and a *head* $h(e)$. The flow conservation requirement can now be written

$$\sum_{t(e)=v} f_e = \sum_{h(e)=v} f_e$$

for all nodes $v \in V$ with $v \neq s, t$. In any feasible flow the net flow out of the source is equal to the net flow into the sink (Exercise 2.2). The *value* of the flow is defined to be this common value

$$val(F) = \sum_{t(e)=s} f_e - \sum_{h(e)=s} f_e.$$

Note that in the examples of this and the next section, there are no edges leading into the source or out of the sink so the second sum in the formula for $val(F)$ is empty in these cases.

In any real network there will be an upper bound on the flow in an edge. With this in mind, we specify for each edge $e$ a *maximum capacity* $c_e$. In addition, we will assume for the moment that the flows are not negative. There are situations where a negative flow makes sense so we will eventually have to deal with this possibility, but for now $f_e \geq 0$. Therefore, for each edge $e$

$$0 \leq f_e \leq c_e.$$

When drawing pictures of networks with flows and capacities, the label on edge $e$ will have the form $(f_e, c_e)$ with the flow preceding the capacity. The edge $a \to b$ in Figure 2.3 has a flow of 6 with a capacity of 10. The value of the flow is 22. (Note that other authors use different conventions). Finally, we will assume that the edge capacities are rational numbers. This is not a serious constraint since in almost all practical problems any irrational number that arises can be replaced with a rational number. This assumption will simplify some of the analysis.

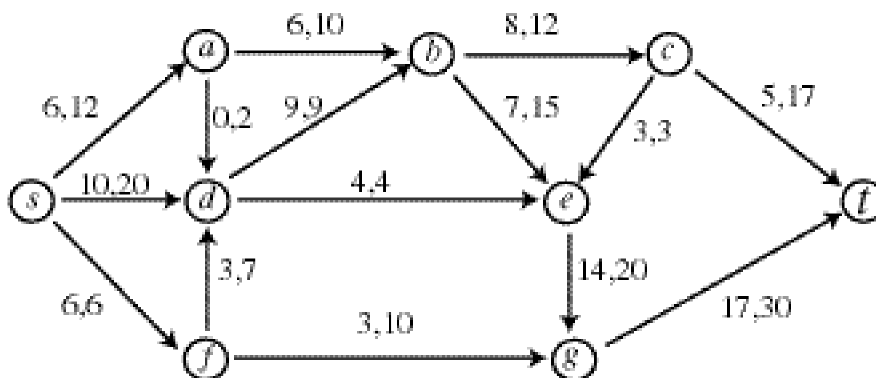



Figure 2.3 A Flow in a Network with edge Capacities

Exercise

2.1 In the network in Figure 2.4, the edge labels are part of a flow $F$ with the flow in some of the edges left blank. Specify the flows for these edges so that the flow is feasible and then determine the flow value $val(F)$.

2.2 Show that in any (feasible) flow in a network, $\sum_{t(e)=s} f_e - \sum_{h(e)=s} f_e = \sum_{h(e)=t} f_e - \sum_{t(e)=t} f_e$.

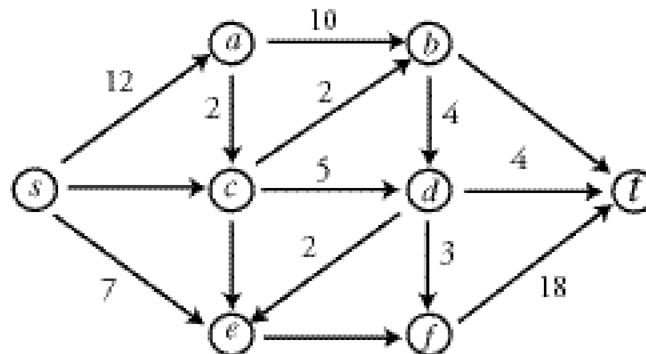



Figure 2.4 Network for Exercise 2.1

We are now in a position to formulate the central problem of this chapter. Given a network with edge capacities, the **Maximum Flow Problem** asks for an algorithm to determine a feasible flow of maximum value subject to the edge capacity constraints.

## 2.2 Augmenting Paths

The flow in Figure 2.3 has value 22. Is there a feasible flow in this network with a larger value? In Figure 2.5, a particular directed path from $s$ to $t$ has been highlighted: $s \to a \to b \to e \to g \to t$. Along this path each of the edges has flow strictly less than capacity, $f_e < c_e$. Clearly there is room for more flow from $s$ to $t$ along this path. moreover, if we increse the flow in each edge by the same increment $\delta$ then we will still have a feasible flow since, at each internal node (i.e. not the source or sink), $\delta$ additional units flow in and $\delta$ additional units flow out.

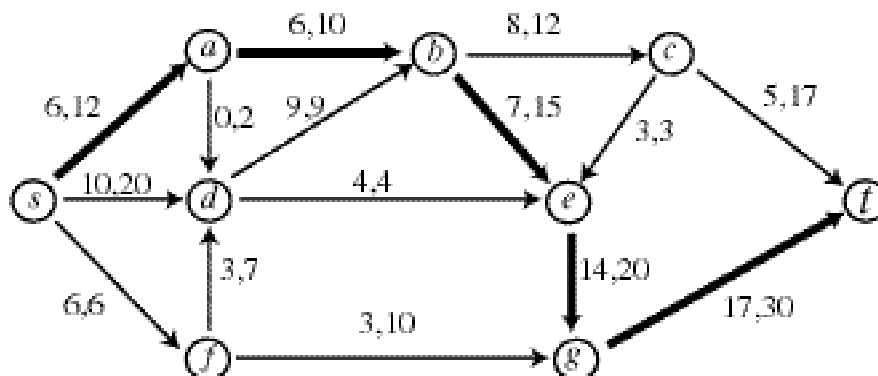



Figure 2.5 An Augmenting Path in a Network

There is a limit to the size of increment we can make. Along an edge $e$ of the path, the flow before was $f_e$ and after is $f_e + \delta$. We must still have $f_e + \delta \leq c_e$ so we need $\delta \leq c_e - f_e$. Thus the largest increment we can make is

$$\delta = min\{c_e - f_e | e \text{ is an edge of the path}\}$$

In Figure 2.5, the values of $c_e - f_e$ along the distinguished path are 6,4,8,6,13. Thus we can increase all flows along this path by the minumum which is 4 units. Notice that once we have augmented the flow along this path by 4 units the flow in edge $a \to b$ is at capacity; such an edge is called *saturated*. Flow cannot be increased in a saturated edge.

So far we have seen that, if there is a directed path from $s$ to $t$ with $f_e < c_e$ on every edge of the path (so no saturated edges), then the flow is not a maximum flow – it can be increased.

Consider now the simple network in Figure 2.6. The flow shown has value 1. Only two edges are not satured: $s \to b$ and $a \to t$. These edges do not form a directed path from $s$ to $t$ so the procedure just developed won't work. In fact,the given flow is not optimal as is clear from Figure 2.7 which exhibits a flow of value 2 in the same network. In order to move from the flow of value 1 to the flow of value 2, it is necessary to decrease the flow in edge $a \to b$. In other words, the flow in Figure 2.7 has made an unwise choice in sending flow from $a$ to $b$. If this unit of flow is redirected from $a$ to $t$, an additional unit of flow can move across the bottom of the network. To make general use of this redirection idea, we will allow some or all of the edges in our distinguished path to point backwards, that is, in the general direction of sink to source.

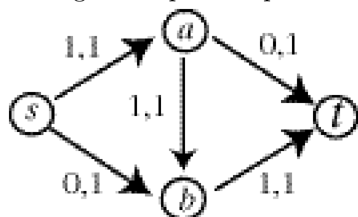
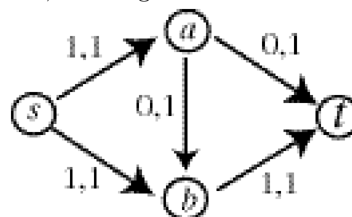



Figure 2.6 A Simple Network




An Optimal Flow in the same Network

An *augmenting path* in a network is a path from source $s$ to sink $t$ consisting of some *forward* edges directed source to sink and some *reverse* edges directed from sink to source along the path. Along the augmenting path we insist that on each forward edge $f_e < c_e$ and on each reverse edge $f_e > 0$.

The highlighted path in Figure 2.8 is an augmenting path: it goes from source $s$ to sink $t$ and all forward edges are non-saturated while the reverse edge $f \to d$ is non-empty. If $\delta$ is chosen so that $\delta \leq c_e - f_e$ for each forward edge and $\delta \leq f_e$ for each reverse edge then we can increase the flow value by $\delta$. The procedure is to add $\delta$ units of flow to the forward edges and subtract $\delta$ units of flow from the reverse edges. In the example, the largest increment possible along this augmenting path is 3 units.

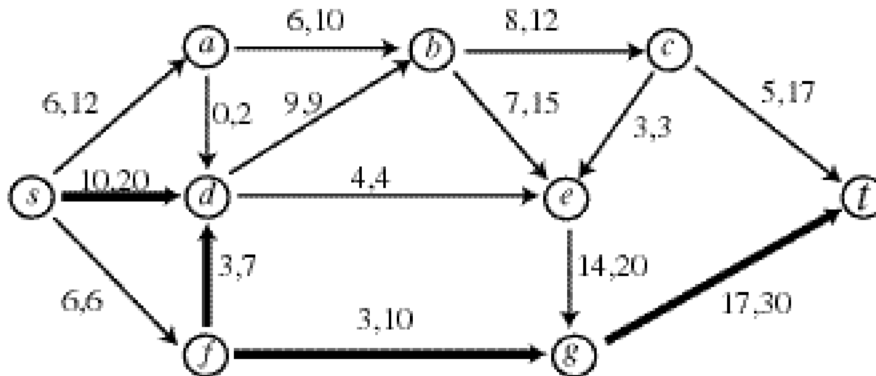



Figure 2.8 An augmenting path with a reverse edge

Staying with the same example, perform both of the augmentations described above. the result is displayed in Figure 2.9.

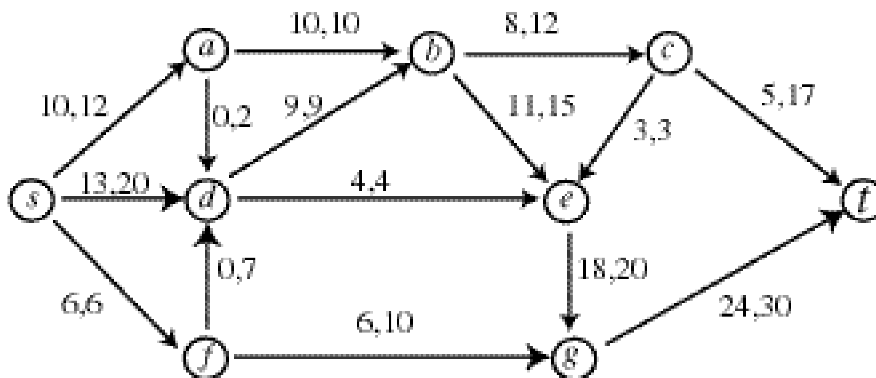



Figure 2.9 The network after both augmentations are performed

There don't appear to be any further augmenting paths in this network. Does this tell us that the flow is optimal? In any optimization situation, one should be on guard for local maxima. Our technique of constructing augmenting paths cannot be used to increase this flow, but does that mean that even if we made more drastic changes in the flow pattern, we would not find a flow of bigger value. This is the case; the next section develops the tool needed to prove this is true.

## 2.3 Cuts in a Network

In the search for augmenting paths in our current flow, we could start at the source $s$ and try to push flow as far as possible toward the sink. If flow has reached a node $x$ then from here flow can be moved forward along non-saturated edges $e$ with $t(e) = x$ and backwards along non-empty edges with $h(e) = x$. There are several ways to organize such a process into an algorithm but for now just let $S$ be the set of nodes such that, with the current flow in place, flow can be pushed to that node. Thus $x \in S$ if and only if there is a (partial) augmenting path from $s$ to $x$. Let $T$ be the set of nodes not already placed in $S$. In Figure 2.10, the set $S = \{s, a, d\}$ is surrounded by a dotted ellipse.

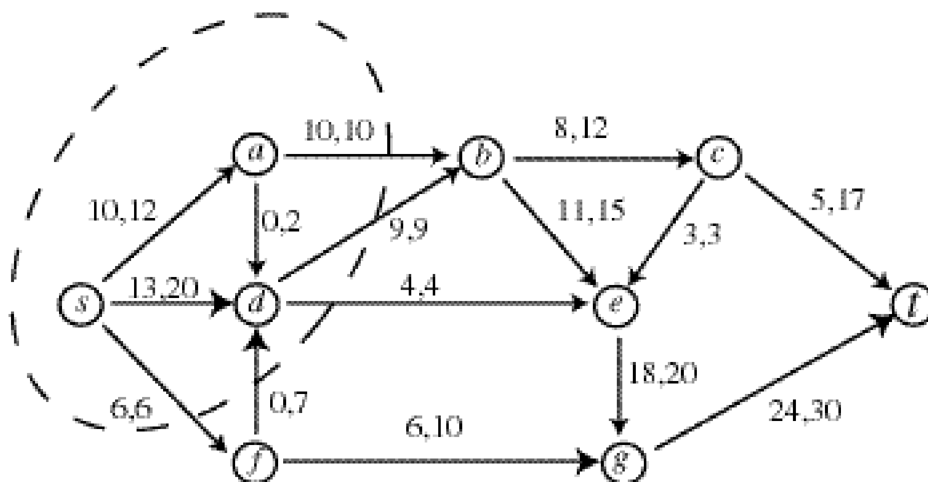



Figure 2.10 The network after both augmentations are performed

The sets $S$ and $T$ have several important features. These are disjoint sets with their union equal to the set of all nodes, so they partition the set of nodes. The source is in $S$ and unless there is an complete augmenting path from $s$ to $t$, the sink $t$ will be an element of $T$. Finally, from our construction of these sets it is clear that the edges coming out of $S$ leading into $T$ must be saturated while the edges leading out of $T$ and into $S$ must be empty. The first two properties are those we use to make our next definition.

A *cut* $(S, T)$ in a network $N = (V, E)$ is a partition of the node set $V = S \cup T$ into disjoint sets $S$ and $T$ such that the source $s \in S$ and the sink $t \in T$. The *capacity of the cut* is defined as

$$cap(S, T) = \sum_{t(e) \in S, h(e) \in T} c_e.$$

Thus the capacity of a cut is the sum of all the capacities of the edges leading out of $S$ and into $T$. This is quite a general definition. In particular, we are not assuming that the cut was determined by an attempt to define an augmenting path. A cut separates the network into two parts and the capacity of the cut sets an upper bound on moving flow from one part to the other. This is the content of the following theorem.

THEOREM 2.1 *Suppose that $N$ is a network with feasible flow $F = \{f_e\}$ and cut $(S, T)$ then*

(i) $val(F) = \sum_{t(e) \in S, h(e) \in T} f_e - \sum_{t(e) \in T, h(e) \in S} f_e$,

(ii) $val(F) \leq cap(S,T)$.

*Proof.* Using the flow conservation property, the difference

$$\sum_{t(e)=v} f_e - \sum_{h(e)=v} f_e$$

equals the flow value if $v = s$ the source and is zero for all other nodes in $S$. Thus we can write

$$val(F) = \sum_{v \in S} \left( \sum_{t(e)=v} f_e - \sum_{h(e)=v} f_e \right) = \sum_{t(e) \in S} f_e - \sum_{h(e) \in S} f_e.$$

But an edge $e$ with both ends in $S$ contributes $+f_e$ through the first sum and $-f_e$ through the second sum; these cancel. All that remains is the terms $+f_e$ for edges with tail in $S$ and head in $T$ and terms $-f_e$ for edges with tail in $T$ and head in $S$. Thus

$$val(F) = \sum_{t(e) \in S, h(e) \in T} f_e - \sum_{t(e) \in T, h(e) \in S} f_e$$

as required for Part (i). To prove Part (ii), simply observe that the first sum is, by definition $cap(S,T)$ and we are subtracting something nonnegative. □

This theorem says that the value of any flow in a newtork is at most the capacity of an arbitrary cut.

## 2.4 The Augmenting Path Algorithm for Max-Flow

We are now in a position to describe the augmenting path algorithm for the Max-Flow problem. We will first describe the algorithm informally then discuss the correctness proof, efficiency and implementation issues.

As input we have a network $N = (V,E)$ with edge capacities $c_e \geq 0$. Our aim is to produce a feasible flow in $N$ of maximum value.

---

**Maximum Flow Algorithm**

(1) Initialize the flow $F$ with $f_e = 0$ for all edges $e \in E$.

(2) Construct the set $S$ of all nodes $x$ such that there is a (partial) augmenting path from the source $s$ to $x$ relative to the current flow $F$.

(3) If the sink $t \in S$ then an augmenting path from $s$ to $t$ has been found. Perform the maximum allowed augmentation along this path. Return to (2).

(4) If the sink $t \notin S$ then stop; the flow is maximal.

---

The first thing to establish is that if the algorithm ever stops then the flow is indeed maximal. What we learn from Theorem 2.1 is that for any flow $F$ and cut $(S,T)$ we have $val(F) \leq cap(S,T)$. In particular the maximum value of a flow is at most the minimum value of any cut. Indeed, if we ever find a flow and cut with equality, $val(F) = cap(S,T)$ then this must be a maximum flow and a minimal cut.

Suppose the algorithm stops. It must have reached Step (4) so the set $S$ of nodes to which flow can be moved from the source does not include the sink $t$. Setting $T = V \setminus S$, we have specified a cut $(S, T)$. Since we cannot extend any of the augmenting paths out of $S$ into $T$, every edge with its tail in $S$ and head in $T$ is saturated and every edge with its head in $S$ and tail in $T$ is empty. Using Theorem 2.1 Part (i), we deduce that for this particular cut, $val(F) = cap(S, T)$. Therefore $F$ is a maximum flow and $(S, T)$ is a minimum cut.

We have shown that if the algorithm ever terminates then it has found a maximum flow (and a minimum cut). Can the algorithm run forever? Each augmentation in Step (3) actually increases the flow by some non-zero amount, so the flow values are a monotonically increasing sequence. By Theorem 2.1, this sequence is bounded above by the capacity of any cut. If the capacities are integers then all flows constructed by the algorithm have integer flows in each edge hence have an integer value. Since there is no infinte, increasing bounded sequence of integers the algorithm must stop at some finite point. Recall that we have assumed that the capacities are rational numbers. A similar argument can be used (there will be an upper bound on the denominators); again the algorithm is sure to stop (see Exercise 2.5). Examples do exist of networks with irrational capacities and particular sequences of augmenting paths that result in an infinite sequence of flows ([Lawler]); the algorithm does not stop in these cases. This is not a very realistic example. In any case, if in Step (3) we are careful about which augmenting path we choose then the algorithm will always terminate. Therefore for any network some sequence of augmentations will lead the Augmenting Path Algorithm to a maximal flow. We summarize these results in the following theorem.

THEOREM 2.2 *Suppose that $N = (V, E)$ is a network with rational edge capacities $c_e \geq 0$. Then*

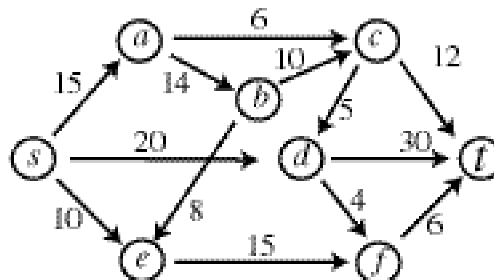
(i) *the augmenting path algorithm determines a maximal flow and a minimum capacity cut in a finite number of augmentations,*

(ii) *if the capacities are integers then there is a maximum flow with integer flows in every edge.*

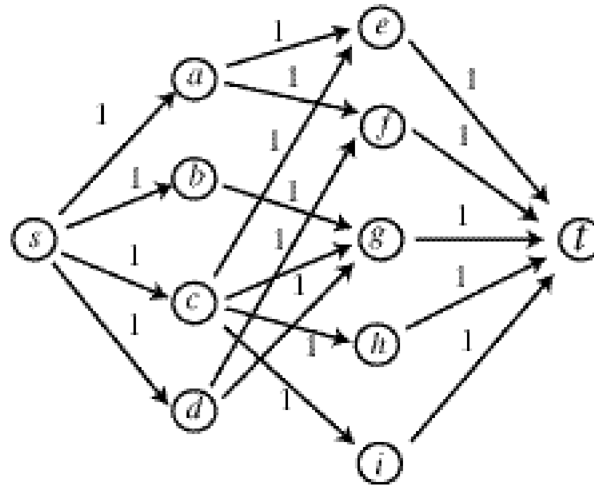
EXERCISES

2.3 In the three networks below the number beside each edge is the capacity of that edge. In each case, determine a maximal flow. Justify your claim that it is maximal.

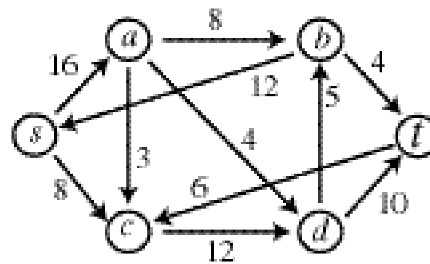
(a)



(b)




(c)




2.4 Give an example of a network which has at least two distinct maximal flows (of equal value, of course!)

2.5 Give an example of a network with at least two distinct minimal cuts.

2.6 Show that the Augmenting Path Algorithm only makes finitely many augmentations in the case that all capcities are rational $c_e \in \mathbf{Q}$.

2.7 Suppose that we have a network where in addition to the edge flow constraints there is a maximum flow $c_v$ that can pass through any internal node $v$. Thus $\sum_{t(e)=v} f_e \leq c_v$. Describe how to modify the network so that this new problem is simply the standard Max Flow problem in the modified network. [Hint: split each internal node in two.]

## 2.5 Implementation and Performance

Breadth first search

$O(nm^2)$.

## 2.6 Applications

Bipartite matching

## Interlude: Linear Program Formulations for Some Combinatorial Optimization Problems

Linear Programming is a commonly taught and used form of optimization. In this area, one tries to maximize or minimize a linear objective function subject to linear inequality contraints. In fact some Combinatorial Optimization problems can be recast as LP problems which can then be solved by standard LP algorithms. In some cases this is a fruitful approach, in others the number of constraints required is so large that solving the problem as an LP is impractical. In this brief interlude we will discuss how two combinatorial optimization problems can be solved through an associated linear program.

Let $G = (V, E)$ be a digraph where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices, and $E = \{e_1, e_2, \ldots, e_m\}$ is the set of arcs. The *incidence matrix* $M$ of $G$ is defined as follows. The rows of $M$ are indexed by the vertices, and the columns of $M$ are indexed by the arcs. The entries of $M$ are all $\pm 1$ or 0. If $e_k = (ij)$ joins $v_i$ to $v_j$, then $M_{ik} = -1$ and $M_{jk} = 1$. In the following, each arc $(ij)$ is assigned a weight $w_{ij}$.

**Potentials and the single source Shortest Path problem**

Consider the problem of finding a minimum weight dipath from a source vertex $s$ to other vertices of $G$.

A vector $\vec{g} = [g_1, g_2, \ldots, g_n]$ is called a *potential* with respect to the weight $\vec{w}$ if

$$g_j - g_i \leq w_{ij} \tag{1}$$

Note that the above system of inequalities (1) can be rewritten using the incidence matrix $M$ as

$$\vec{g}M \leq \vec{w}$$

For each $v \in V$, let $y_v$ be the weight of a minimum dipath from the source node $s$ to a vertex $v$. Suppose $P\colon s = v_0, v_1, \ldots, v_k = v$ is a dipath with weight $y_v$. Then we have

$$y_v = \sum_{j=0}^{k-1} w_{v_j v_{j+1}} \geq \sum_{j=0}^{k-1} \left(g_{v_{j+1}} - g_{v_j}\right) = g_v - g_s$$

On the other hand, $\vec{y}$ itself is a potential (Exercise) and trivially $y_s = 0$, so

$$y_v = y_v - y_s \geq g_v - g_s$$

for any potential $\vec{g}$. Hence the value of $y_v$ is given by the maximum potential.

Define a vector $\vec{c}$ indexed by vertices with $c_v = 1$, $c_s = -1$, and $c_u = 0$ for all $u \in V - \{v, s\}$. So $y_v - y_s = \vec{y}\vec{c}$ and $g_v - g_s = \vec{g}\vec{c}$. Then the following LP:

$$\text{Maximize } \vec{g}\vec{c} \text{ subject to } \vec{g}M \leq \vec{w} \tag{2}$$

has as its value the length $y_v$ of a minimal dipath from $s$ to $v$. In the LP there is one constraint for each arc. If the graph is reasonably sparse then this approach may well lead to an efficient solution to the minimal dipath problem.

The dual of the above LP is, with variable vector $\vec{h}$ indexed by arcs:

$$\text{Minimize } \vec{w}\vec{h} \text{ subject to } M\vec{h} = \vec{c}, \quad \vec{h} \geq 0. \tag{3}$$

Each dipath $P$ from $s$ to $v$ gives a feasible solution $\vec{h}_P$ to this dual LP, which is called the *characteristic vector* of $P$. It is defined by $h_e = 1$ for each arc $e \in P$, and $= 0$ otherwise. Then $\vec{w}\vec{h}_P$ is the weight of the dipath $P$. What LP Duality Theory tells us is that the minimum value of the dual LP is equal to the maximal value of the primal LP and this value is the weight of a minimal dipath from $s$ to $v$. The minimal value of the dual LP is achieved by the characteristic vector of a minimal dipath. There may be other optimal solutions of the dual that are not characteristic vectors of dipaths.

We note though that an arbitrary vector $\vec{h}$ with components 0 or 1 which is dual feasible is a characteristic vector of an $s - v$ dipath. Indeed the matrix product $M\vec{h} = \vec{c}$ can be considered a set of equations – one for each vertex $u$. The equation for this vertex (using the definition of $M$) adds the components of $\vec{h}$ on arcs with $u$ as head and subtracts the components with $u$ as tail. The definition of the vector $\vec{c}$ (value $+1$ at $v$ , $-1$ at $s$ and 0 elsewhere) ensures that if $\vec{h}$ is a $0-1$ vector feasible for the dual LP, then $\vec{h}$ is a characteristic vector of a dipath from $s$ to $v$.

This discussion shows how the minimum path problem and the maximum potential problem can be formulated as linear programs that are dual to each other.

**Maxflow and Mincut problems**

A network consists of a digraph $G$, a source node $s$ and a sink node $t$ in $G$, and a capacity function $c = \{c_e : e \in E\}$. A vector $f = \{f_e : e \in E\}$ is called a flow in $G$ if

$$\sum_{h(e)=v} f_e - \sum_{t(e)=v} f_e = 0, \quad v \in V - \{s, t\} \tag{4}$$

where $t(e)$ and $h(e)$ denote the tail and head of the arc $e$, respectively. Condition (4) is called the *flow conservation law*. Note that the flow conservation law can be expressed as

$$\hat{M}f = 0$$

where $\hat{M}$ is the matrix obtained from the incidence matrix $M$ by deleting rows corresponding to $s$ and $t$. The maximum flow problem can be formulated as the following linear program.

$$\text{Maximize } \sum_{h(e)=t} f_e - \sum_{t(e)=t} f_e \text{ subject to } \hat{M}f = 0, \quad 0 \leq f \leq c$$

The augmenting path algorithm is a primal algorithm, since at each step it maintains a feasible flow and improves towards getting a dual feasible solution (a saturated cut).

The preflow-push algorithm can be viewed as a dual algorithm, since at each step it mains a dual feasible solution (saturated cut), and improves towards getting a primal feasible solution (feasible flow).

The LP formulation of transportation and assignment problems is straightforward. One usually uses a primal (simplex) algorithm to solve the transportation problem, and uses a dual algorithm to solve the assignment problem (such as Kuhn's Hungarian method).

# Chapter 3

# Maximum Flow by Preflow-Push

## 3.1 Pre-flows and Distance Labellings

The Augmenting Path Algorithm of Chapter 2 maintains a feasible flow at each stage. The value of the flow is steadily increased until a saturated cut has been formed. This is the signal that the flow has a maximum value. An alternate approach is used by the preflow-push algorithms. These algorithms maintain at all stages a feasible pre-flow (a "flow" without flow conservation) that has a saturated cut. The pre-flow is changed step by step until it does satisfy flow conservation. The resulting flow then has a saturated cut so is a maximum flow.

Suppose that $N$ is a network with edge capacities $c_{ab}$ and $\mathcal{P} = \{p_{ab}\}$ is an assignment of flow to the edges of $N$. Then $\mathcal{P}$ is a *preflow* if at each internal node $v \neq s, t$, we have $\sum_{h(e)=v} p_e \geq \sum_{t(e)=v} p_e$. Thus we assume that, at each internal node, the flow in is at least as large as the flow out. The *excess flow* at $v$ is defined by

$$ex(v) = \sum_{h(e)=v} p_e - \sum_{t(e)=v} p_e = \sum_w p_{wv} - \sum_u p_{vu}.$$

Of course, if $ex(v) = 0$ for all internal nodes $v$ then $\mathcal{P}$ is a flow. We call $\mathcal{P}$ a *feasible preflow* if, in addition, $0 \leq p_e \leq c_e$. Our general strategy is to start with an initial preflow then successively modify the flows on various edges so that at all times the preflow is feasible and moves steadily toward satisfying flow conservation.

Suppose we have a network $N$ with a preflow $\mathcal{P} = \{p_{ab}\}$. If there is an internal node with positive excess $ex(a) > 0$ then we would like to push this flow away from $a$ and toward the sink $t$. If we can somehow push all the excess flows to the sink then we will have a maximum flow. If we want to push excess flow from $a$ to $b$ then we can do it forward along $ab$ if $p_{ab} < c_{ab}$ or backwards along $ba$ if $p_{ba} > 0$ or both. A *push on* $ab$ is performed by moving up to $ex(a)$ units of flow from $a$ to $b$ with up to $c_{ab} - p_{ab}$ units moving forward on $ab$ and up to $p_{ba}$ units moving backward on $ba$. Once a push along $ab$ has been performed, the excess $ex(a)$ at a will decrease while $ex(b)$ will increase.

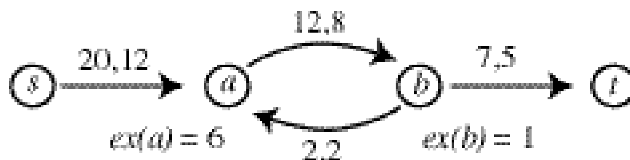



Figure 3.1 A simple preflow

As an example, consider the simple network in Figure 3.1. There are 6 excess units at $a$. We can push

all 6 units along $ab$ with 4 units moving foward along $ab$ and 2 units moving backward along $ba$. The result is shown in Figure 3.2. The preflow now satisfies flow conservation at $a$ but not at $b$.

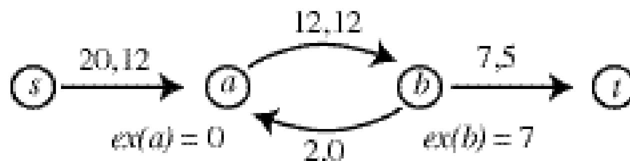



Figure 3.2 The preflow after pushing on $ab$

There might be several different ways to perform a push along $ab$. If, in the example, we were only pushing away an excess of 3 units there would not be enough new flow both to saturate $ab$ and to empty $ba$; there would be a choice to make. Moreover, the push on $ab$ might not exhaust the excess at $a$ so that further pushes are needed at $a$. The simple network fragment in Figure 3.3 illustrates the point. To move all 7 excess units away from $a$ will require pushes on several edges. An actual implementation will have to include rules that decide which edges to use and in what order and also which nodes to work on.

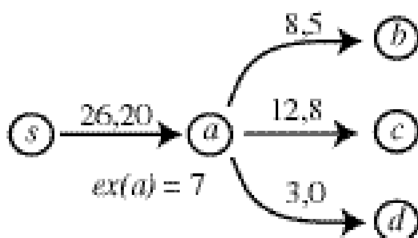



Figure 3.3 A preflow in a network fragment

Suppose we start with a preflow in our network and randomly push excess flow away from nodes. So far there is nothing to stop us from pushing the excess flow around in a cycle or even straight back where it came from. Pushing flow is an entirely local event; we need a global tool to direct our progress from preflow to flow.

A *distance labelling* $\lambda$ assigns to each node $v$ an integer $\lambda(v)$. The conditions that makes $\lambda(v)$ a distance labelling are that: $\lambda(s) = n = |V|$, $\lambda(t) = 0$ and if edge $ab$ is available for a push then $\lambda(a) \le \lambda(b) + 1$. So we insist that, whenever $p_{ab} < c_{ab}$ or $p_{ba} > 0$ we have $\lambda(a) \le \lambda(b) + 1$.

For any network $N$ it is possible to define a feasible preflow $\mathcal{P}$ and corresponding distance labelling $\lambda$ as follows. For every edge $sa$ leading out of the source, set $p_{sa} = c_{sa}$. For all other edges set $p_{ab} = 0$. Certainly $0 \le p_e \le c_e$ for all edges $e$. At an internal node $v$ that is not the head of an edge from the source we have all preflows in and out equal to zero, so $ex(v) = 0$. If, on the other hand, there is an edge $sv$, then $ex(v) = c_{sv} \ge 0$. Therefore $\mathcal{P}$ is indeed a feasible preflow. Using this preflow, a valid labelling is obtained by setting $\lambda(s) = |V|$ and $\lambda(v) = 0$ for all other vertices. With this definition, $\lambda(a) = \lambda(b)$ unless $a$ or $b$ is the source $s$. The edges $sv$ are not available for a push since $p_{sv} = c_{sv}$ and $p_{vs} = 0$. Therefore the definition of distance labelling is satisfied by $\lambda$.

Figure 3.4 illustrates these ideas. The edge labels are in the order $c_e, p_e$ and the numbers at the vertices form the distance labelling.

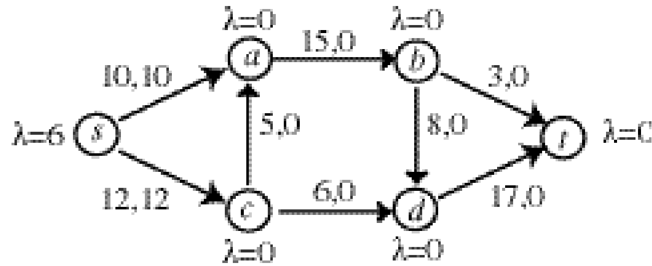



Figure 3.4 The Initial Preflow and Labelling of a Network

In Figure 3.4, the partition $S = \{s\}, T = \{a, b, c, d, t\}$ defines a saturated cut since all the edges coming out of the source are saturated and had there been an edge directed into the source it would be empty. As mentioned earlier, the existence of such a saturated cut is a general phenomenon. We complete this section with a proof of this fact.

THEOREM 3.1 *Suppose that network $N$ has a feasible preflow $\mathcal{P}$ and a distance labelling $\lambda$. Then there is a cut $(S, T)$ with every edge of $\delta(S)$ saturated and every edge of $\delta(T)$ empty*

*Proof.* Since there are $n = |V|$ nodes in the network then amongst the $n + 1$ integers $0 \ldots n$, there is at least one integer $k$ which is not used in the labelling. Let $S = \{v | \lambda(v) > k\}$ and $T = V \setminus S$. So $s \in S$ and $t \in T$. Also, for any edge $e = ab$ with $a \in S$ and $b \in T$, we have $\lambda(a) > k > \lambda(b)$. But then $\lambda(a) > \lambda(b) + 1$. By the definition of a distance labelling, this edge must be not admissible. Hence $ab$ is saturated and $ba$ is empty as required. □

This theorem says that any feasible preflow with a distance labelling has a saturated cut. For the special case of a flow, the Max-Flow Min-Cut Theorem then gives the following corollary which establishes the stopping condition for the preflow-push algorithms.

THEOREM 3.2 *If a feasible flow $F$ has a distance labelling then $F$ is a maximum flow.*

## 3.2 Preflow Push Algorithms

In this section, the ideas of preflow and distance labelling on a network will be fashioned into an algorithm for the maximum flow problem called the Preflow Push Algorithm. In fact, several of the details required for implementing Preflow Push can be specified in different ways so this can be considered a class of algorithms based on a central theme that we develop here.

We have a network $N$ with a preflow $\mathcal{P}$ and a distance labelling $\lambda$. At each internal node $v$ there is some excess flow $ex(v)$. As before we denote the capacity of the arc $ab$ by $c_{ab}$ while the preflow on this edge is $p_{ab}$.

We will call an edge $ab$ an *available edge* if either $ab$ or $ba$ is an arc of $N$ and either $p_{ab} < c_{ab}$ or $p_{ba} > 0$ or both. Thus an available edge is a candidate for a push operation. Note that the availability of an edge depends on the preflow; as the preflow is changed a given edge $ab$ may gain or loose this status. The maximum amount that can be pushed along an available edge $ab$ is $\tilde{c}_{ab} = c_{ab} - p_{ab} + p_{ba}$. The key to controlling the pushing of flow is to define an available edge $ab$ to be *admissable* if and only if $\lambda(a) = \lambda(b) + 1$. The Preflow Push Algorithm allows pushes only on admissable edges.In Figure 3.5, edges $va, vb, vc$ and $vd$ are available but only $vb$ and $vc$ are admissable. Note also that if we shift attention to $a$ the edge $av$ is admissible. From the definition, if an edge $uv$ is admissible then edge $vu$ is definitely not. Thus the labelling tells us a direction to push flow.

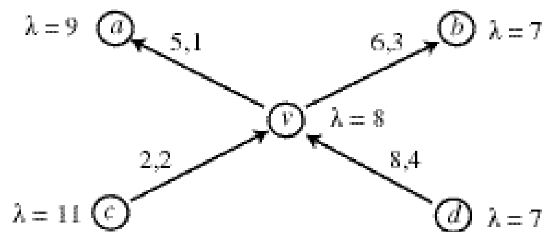



Figure 3.5 Available and admissible edges at $v$

The final ingredient for the algorthmn is *relabelling*. If, in Figure 3.5, we want to push excess flow from $v$ to $a$, this will not be possible unless the labels change so that $va$ is admissable, that is $\lambda(v) = \lambda(a) + 1$. Therefore some relabelling procedure is required.

The Preflow Push algorithm is based on the following local routine that operates on the excess flow at a single vertex.

---

**Process**($v$)

1. Start with an active node $v$ (so $ex(v) > 0$).
2. Pick an admissible edge $va$.
3. Push $min\{ex(v), \tilde{c}_{va}\}$ additional flow along $va$.
4. Repeat Steps 2 and 3 until either $ex(v) = 0$ or there are no more admissible edges.

5. If $ex(v) > 0$ and there are no admissable then relabel $v$ as follows. Set

$$\lambda(v) = min\{\lambda(a) + 1 | va \text{ is an available edge}\}$$

This will produce at least one admissible edge; return to Step 2. Note that since $ex(v) > 0$ this excess flow must have arrived on some arcs that are now available for a push.

6. If $ex(v) = 0$ then $v$ is no longer active and we are finished processing $v$.

---

With this routine in hand, the Preflow Push algorithm can be simply stated.

**Preflow Push Algorithm**

1. Initialize: define an initial preflow and distance labelling on $N$
2. While an active node $v$ remains: Process($v$).
3. The preflow is now a maximum flow.

EXAMPLE 3.1

The example in Figure 3.6 is an extremely simple, indeed linear, network. It is transparent that the maximum flow value is 3. Nevertheless, it is a useful first example of pushing and relabelling.

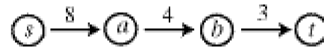


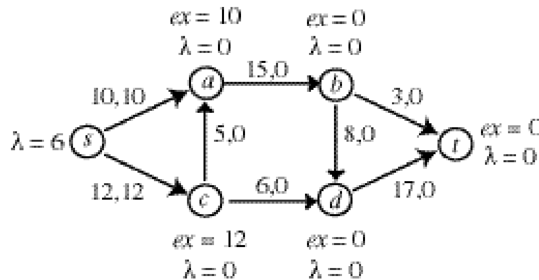

Figure 3.6 A simple example (Edge labels are capacities.)

| | | $\lambda(s)$ | $p_{sa}$ | $ex(a)$ | $\lambda(a)$ | $p_{ab}$ | $ex(b)$ | $\lambda(b)$ | $p_{bt}$ | $ex(t)$ | $\lambda(t)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| initialization | | 4 | 8 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| process($a$) | relabel $a$ | 4 | 8 | 8 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| | push on $ab$ | 4 | 8 | 4 | 1 | 4 | 4 | 0 | 0 | 0 | 0 |
| | relabel $a$ | 4 | 8 | 4 | 5 | 4 | 4 | 0 | 0 | 0 | 0 |
| | push on $as$ | 4 | 4 | 0 | 5 | 4 | 4 | 0 | 0 | 0 | 0 |
| process($b$) | relabel $b$ | 4 | 4 | 0 | 5 | 4 | 4 | 1 | 0 | 0 | 0 |
| | push on $bt$ | 4 | 4 | 0 | 5 | 4 | 1 | 1 | 3 | 3 | 0 |
| | relabel $b$ | 4 | 4 | 0 | 5 | 4 | 1 | 6 | 3 | 3 | 0 |
| | push on $ba$ | 4 | 4 | 1 | 5 | 3 | 0 | 6 | 3 | 3 | 0 |
| process(a) | push on $as$ | 4 | 3 | 0 | 5 | 3 | 0 | 6 | 3 | 3 | 0 |
| STOP: | maximum flow | | 3 | | | 3 | | | 3 | | |

EXAMPLE 3.2

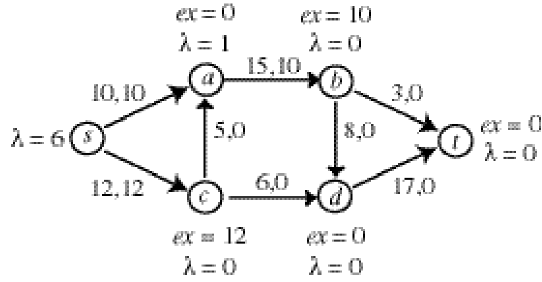
As a second example, we will use Preflow Push to determine a maximum flow in the network of Figure 3.4. In this example, the next node to be processed is choosen arbitrarily among the current active nodes.

1. Initialize the preflow and labelling.



2. Relabel $a$: $\lambda(a) = 1$

3. Push on $ab$: $p_{ab} = 10, ex(a) = 0$




4. Relabel $c$: $\lambda(c) = 1$

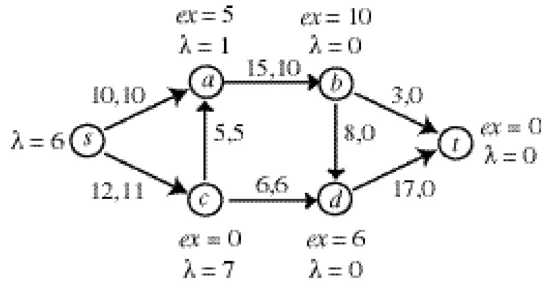
5. Push on $cd$: $p_{cd} = 6, ex(c) = 6$

6. Relabel $c$: $\lambda(c) = 2$

7. Push on $ca$: $p_{ca} = 5, ex(c) = 1$

8. Relabel $c$: $\lambda(c) = 7$

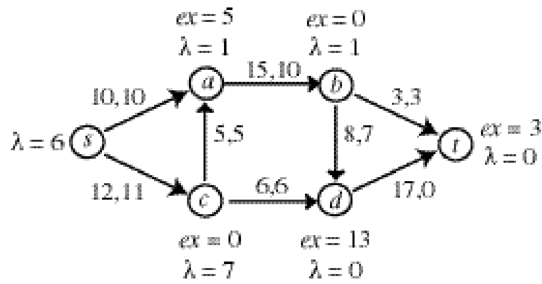
9. Push on $cs$: $p_{sc} = 11, ex(c) = 0$




10. Relabel $b$: $\lambda(b) = 1$

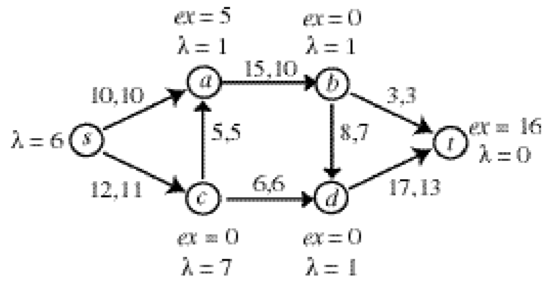
11. Push on $bt$: $p_{bt} = 3, ex(b) = 7$

12.Push on $bd$: $p_{bd} = 7, ex(b) = 0$



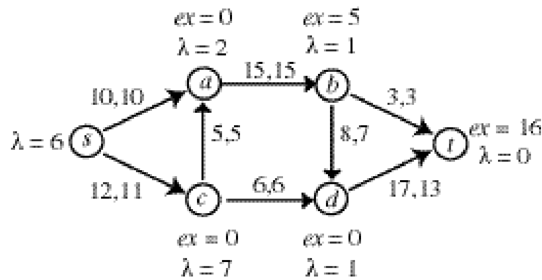

13. Relabel $d$: $\lambda(d) = 1$

14. Push on $dt$: $p_{dt} = 13, ex(d) = 0$

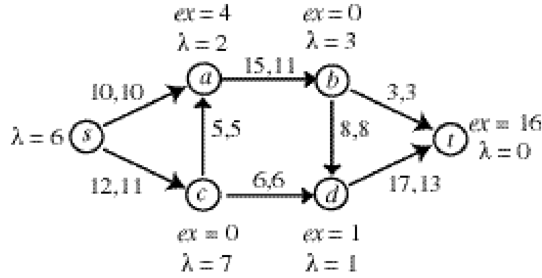



15. Relabel $a$: $\lambda(a) = 2$

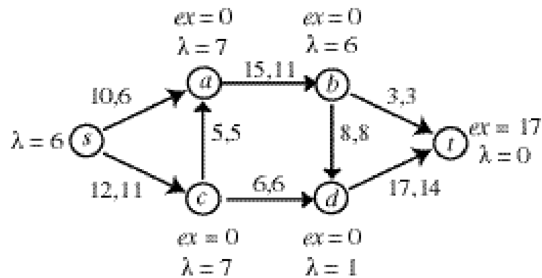
16. Push on $ab$: $p_{ab} = 15, ex(a) = 0$



17. Relabel $b$: $\lambda(b) = 2$
18. Push on $bd$: $p_{bd} = 8, ex(b) = 4$
19. Relabel $b$: $\lambda(b) = 3$
20. Push on $ba$: $p_{ab} = 11, ex(b) = 0$




21. Push on $dt$: $p_{dt} = 14, ex(d) = 0$
22. Relabel $a$: $\lambda(a) = 4$
23. Push on $ab$: $p_{ab} = 15, ex(a) = 0$
24. Relabel $b$: $\lambda(b) = 5$
25. Push on $ba$: $p_{ab} = 11, ex(b) = 0$
26. Relabel $a$: $\lambda(a) = 6$
27. Push on $ab$: $p_{ab} = 15, ex(a) = 0$
28. Relabel $b$: $\lambda(b) = 6$
29. Push on $ba$: $p_{ab} = 11, ex(b) = 0$
30. Relabel $a$: $\lambda(a) = 7$
31. Push on $as$: $p_{sa} = 6, ex(a) = 0$
32. STOP: No active vertices remain.




On such a small example it is easy to see ways the algorithm could have taken a short cut. For example, the relabelling loop between nodes $a$ and $b$ is a candidate. The problem is that the distance labels are our global control, but we are modifying them locally. One way of improving performance is to periodically calculate a new distance labelling for the entire network. This is discussed further in the next section.

Before leaving this example, locate in each of the displayed networks the saturated cut guaranteed by Theorem 3.1.

## 3.3 Performance

**Correctness and Worst-case Analysis**

In order to show that the Preflow-Push algorithm is correct, we must show that the algorithm stops in a finite number of steps with a maximum flow. Since we can estimate the number of steps used, the correctness proof also yields a worst-case estimate at the same time. To simplify the discussion, we will analyze the instance of Preflow-Push that always picks an active vertex with maximal label for processing.

There are several facts to be checked. First, we must show that the algorithm maintains both a valid feasible preflow and a valid distance labelling at each stage. The next component of the analysis is the proof that, if there are $n$ nodes, then the algorithm takes at most $2n^2$ relabel steps and at most $2n^3$ push steps. Finally, we show that the algorithm doesn't stop while there is an active node. Theorem 3.2 ensures that when the algorithm does stop it has determined a maximum flow.

The argument that the algorithm always maintains a valid feasible preflow is straightforward (Exercise 3.1). The key property of a distance labelling is that $\lambda(a) \leq \lambda(b) + 1$ for any available edge $ab$. A push step does not change the labels so the only risk to the distance labelling would be from an edge becoming available. Suppose that some push step changes the status of an edge $ab$ from unavailable to available. It must have been a push along $ba$ that did this so edge $ba$ was admissable before that push. Hence $\lambda(b) = \lambda(a) + 1$. Then $\lambda(a) = \lambda(b) - 1 < \lambda(b) + 1$ as required. Relabelling is specified by an assignment

$$\lambda(v) = min\{\lambda(a) + 1 | va \text{ is an available edge}\}$$

This ensures that after the relabelling $\lambda(v) \leq \lambda(a) + 1$ for any available edge $va$. The only other case to checkis an available edge $bv$. But before the relabel $\lambda(b) \leq \lambda(v) + 1$ which remains valid since the label on $v$ has increased (Exercise 3.2).

The following lemma bounds the number of relabel steps.

LEMMA 3.3 *If $v$ is an active node then $\lambda(v) \leq 2n - 1$*

*Proof.* The key is that if $v$ is an active node then there is a sequence

$$v = v_0 v_1 \cdots v_k = s$$

of nodes leading back to the source with each edge $v_i v_{i+1}$ available. Clearly $k$ can be taken less than $n$. The result follows from this since the definition of distance labelling says that $\lambda(v_i) \leq \lambda(v_{i+1}) + 1$ for all $i$ so that $\lambda(v) \leq \lambda(s) + k \leq n + (n - 1) = 2n - 1$.

Let $S$ be the set nodes for which such a sequence $v = v_0 v_1 \cdots v_k = s$ exists and let $T = V \setminus S$ be the complement. Our goal is to show that all active nodes are in $S$ or equivalently that there are no active nodes in $T$. Focus on the sum $\sum_{v \in T} ex(v) \geq 0$. As usual, we expand, change the order of summation and collapse to get

$$0 \leq \sum_{v \in T} ex(v) = \sum_{v \in T} \left( \sum_{a \in V} p_{av} - \sum_{b \in V} p_{vb} \right) = \sum_{a \in S, v \in T} p_{av} - \sum_{v \in T, b \in S} p_{vb} \leq \sum_{a \in S, v \in T} p_{av}.$$

By the definition of $S$ the preflows $p_{av}$ in the last sum are all 0. This forces $\sum_{v \in T} ex(v) = 0$ and therefore all nodes in $T$ are inactive. □

Since there are $n-2$ nodes that can be relabelled and each can be relabelled at most $2n-1$ times the number of relabels steps is at most $(n-1)(2n-1) < 2n^2$.

Having counted the maximum possible number of relabel steps, we next consider the push steps. If you look back at Example 3.2 you will notice that there are two types of pushes used by the algorithm. Some pushes at node $a$ reduce $\tilde{c}_{ab}$ to 0 either by making $p_{ab} = c_{ab}$ or by making $p_{ba} = 0$ while leaving $a$ active. These are called *saturated* pushes. If, on the other hand, $ex(a) < \tilde{c}_{ab}$ then only $ex(a)$ can be pushed, leaving $ab$ available. These are called *unsaturated* pushes. After a saturated push along $ab$, the edge $ab$ is unavailable; after an unsaturated push along $ab$, the node $a$ is inactive. It is convenient to count the number of saturated and unsaturated pushes separately.

LEMMA 3.4 *If the Preflow-Push algorithm always processes a node with maximal distance label then the number of unsaturated pushes is less than $n$ times the number of relabels*

*Proof.* Suppose that there is an unsaturated push at a node $v$. Then this node is now inactive and by assumption $\lambda(v) \geq \lambda(w)$ for any other active node $w$. Before we can process node $v$ again, more flow will have to be pushed to $v$. If this flow is pushed from node $w$, say, then $wv$ is admissible at that moment and node $w$ will have to be relabelled at some point to make $\lambda(w) = \lambda(v) + 1$. This shows that between two unsaturated pushes at node $v$ some node gets relabelled. Since there are $n$ choices for $v$ the result follows. □

LEMMA 3.5 *The total number of saturated pushes on a given edge $ab$ is at most $n$*

*Proof.* We show that between two saturated pushes on $ab$ the distance label on $a$ increases by at least 2. The result will then follow from Lemma 3.3. Suppose that the algorithm performs a saturated push on $ab$. Then, by definition, $ab$ is no longer available. Before $ab$ can be available again, there must be a push on $ba$. We use $\lambda$ for the distance labels at the time of the first push on $ab$, and $\lambda'$ for the labels at the time of the push on $ba$. Finally we denote by $\lambda''$ the distance labels at the time of the next saturated push on $ab$. Thus $\lambda(a) \leq \lambda'(a) \leq \lambda''(a)$ and $\lambda(b) \leq \lambda'(b) \leq \lambda''(b)$. Then using the definition of an admissible edge we have

$$\lambda''(a) = \lambda''(b) + 1 \geq \lambda'(b) + 1 = \lambda'(a) + 2 \geq \lambda(a) + 2.$$

□

Since there are $n(n-1)$ ordered pairs of nodes potentially defining edges for a push, the total number of saturated pushes is at most $n^2(n-1) < n^3$. Combining all of this analysis gives the following result.

THEOREM 3.6 *If the Preflow-Push algorithm is implemented so as to process next a node with maximal label then there are at most $2n^2$ relabel steps and at most $3n^3$ pushes. In particular, the algorithm stops with a maximum flow in a finite number of steps.*

It is only a bit harder to analyze the general Preflow-Push algorithm. See [BILLS] or [GOLDBERG1] for a full acount.

**Practical Performance**

EXERCISES

3.1 Show that, in the Preflow-Push algorithm each step maintains a valid feasible preflow.

3.2 Show that in the relabel step $\lambda(v) = min\{\lambda(a) + 1 | va$ is an available edge$\}$ the label $\lambda(v)$ increases.

3.3 In Example 3.2, identify both a saturated push and an unsaturated push.

# Chapter 4
# Min-cost Flows

## 4.1 Minimum Cost Flow Problems

In a typical network, the cost of using the various links varies from arc to arc. The cost of using a satellite, optical fiber or copper wire is not the same. This leads to the simple question: How can we find, in a network with arc capacities and arc costs, a minimum cost flow of a given flow value? For example, consider the network in Figure 4.1.

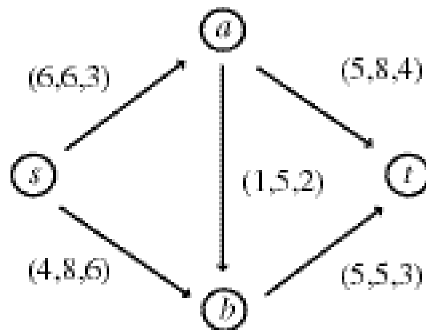



Figure 4.1 A network with arc capacities and costs

The triple on each arc $(f_{ij}, c_{ij}, d_{ij})$ denotes the flow in the arc, the maximum capacity of the arc and the cost per unit flow of using the arc. The flow in the example has value 10 and cost $6{\cdot}3+4{\cdot}6+5{\cdot}4+1{\cdot}2+5{\cdot}3 = 79$. Suppose now that the unit of flow down arc $ab$ is diverted into arc $at$ while the flow in arc $bt$ is reduced to 4 (see Figure 4.2). The result is again a flow of value 10, but the total cost is reduced to 78.

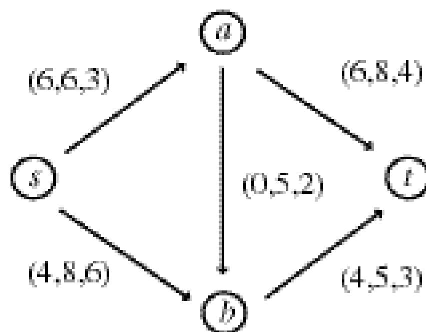



Figure 4.2 The same network with an alteration in the flow pattern

What is the minimum cost of a flow with value 10 in this network and how can we determine such a min-cost flow? This is the central question of this chapter.

The general problem of determining a minumum cost flow with a given value is more complex than the

max-flow problem. We now have two constraint parameters, capacity and cost, attached to each arc. These parameters constrain the problem in different ways. We will present two different approaches to the min-cost problem. In simple terms, one method works with a flow of given value and tries to modify it into a flow of the same value and lower cost. The second approach is to incrementally build up the flow from 0 in such a way that at each stage we have a min-cost flow of the current value. When the flow value reaches the target value, we have a min-cost flow.

Again we have the basic optimality duality appearing. We can either work with feasible instances and move toward the optimality condition or we can work with infeasible instances satisfying the optimality condition and move toward a feasible instance.

One straightforward approach to the min-cost flow problem is to build up flow from zero using the least cost augmenting path available at each stage. Define the cost $C(P)$ of an $s$–$t$ path $P$ as the net cost obtained by adding the costs of the forward arcs and subtracting the costs of the reverse arcs. Thus $C(P)$ is the increment in the flow cost when a unit augmentation follows $P$. If the network happens to contain a negative cost "augmenting" circuit, we can pass flow around this circuit lowering the cost of the flow without changing the value. This process will stop when some arc of the circuit becomes saturated or empty.

**Algorithm Build-Up**

1. Initialize with flow 0 in each arc.
2. Find a minimum cost augmenting path $P$ or a negative cost augmenting circuit $D$.
3. Augment the flow along $P$ or $D$ as much as possible . Stop if the flow value reaches $v$.
4. Go back to Step 2.

It turns out that this "greedy" approach works; for a proof see [POP & STIEG, page 142] for example. This is a theoretical success but actually just reduces the problem to finding minimum cost augmenting paths. One way to find such paths is to define a related network where the augmenting paths of the original network now become simple directed paths in the new *incremental network*. Once this is done, minimum cost augmenting paths become shortest d-paths and a standard algorithm can be used. We take up the discussion of the incremental network in the following Section.

## 4.2 The Incremental Network

We have seen that the search for an augmenting path must allow adges going both forward and backward. There is a straight forward transformation of the network that allows us to work with forward edges only. This *incremental* network is useful for the min-cost flow problem. It is defined as follows.

Suppose that $\mathcal{F}$ is a flow in a network $\mathcal{N}$ with arc capacities $c_{ij}$. If $\mathcal{F}$ places flow $f_{ij}$ on arc $ij$ then an augmenting path can add up to $c_{ij} - f_{ij}$ more units of flow to this edge when the arc is used as a forward edge and can add up to $f_{ij}$ units of flow if the arc is used as a backward edge.

Define a new network $\mathcal{N}(\mathcal{F})$ on the same nodes as $\mathcal{N}$ but with modified capacities. For each arc $ij$ define the *reduced capacities* as:

$$\tilde{c}_{ij} = c_{ij} - f_{ij}$$

$$\tilde{c}_{ji} = f_{ij}$$

The cost of arc $ij$ is $d_{ij}$ while the cost of using arc $ji$ in $\mathcal{N}(\mathcal{F})$ is $-d_{ij}$. Thus the incremental network is coding the backward edges as negative costs. The new network does not yet have any flow assigned to its arcs.

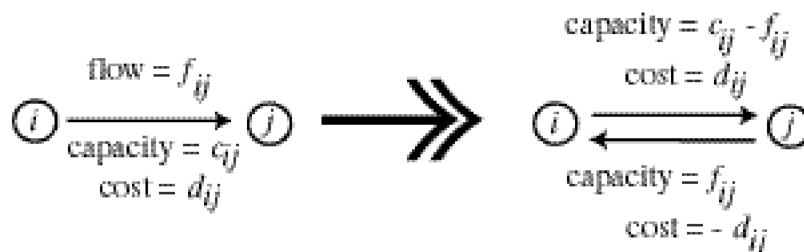



Figure 4.3 Specification of the incremental network

Figure 4,4 illustrates the incremental network derived from the first example, Figure 4.1.

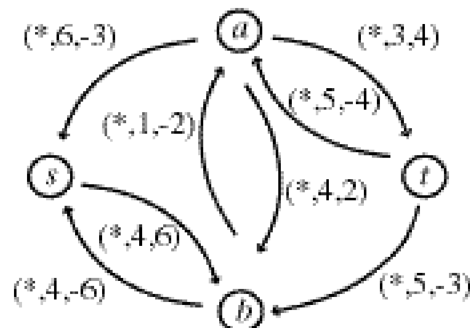



Figure 4.4 The incremental network of Figure 4.1

The definition of the incremental network implies that an $s-t$ di-path in $\mathcal{N}(\mathcal{F})$ corresponds to an $s-t$ augmenting path in the original network $\mathcal{N}$. If the cost of the $s-t$ di-path in $\mathcal{N}(\mathcal{F})$ is $x$ then making an augmentation of one unit along this path, in $\mathcal{N}$, will increase the flow by 1 unit and the cost by $x$. Another feature of incremental networks is that since they can have negative cost edges they can also have negative cost di-cycles. Consider the example of an incremental network $\mathcal{N}(\mathcal{F})$ in Figure 4.5. The cycle

$a \to b \to d \to c \to a$ has a total cost $2 + 3 - 10 + 2 = -3$. Moving flow around this cycle will preserve the flow conservation condition and hence keep the flow at the same value. At the same time each additional unit of flow around the cycle will reduce the cost by 3. Of course, eventually we will saturate or empty some arc and the cycle will disappear from the incremental network. This illustrates the importance of negative cost cycles in min-cost flow algorithms.

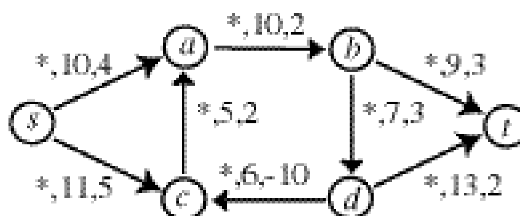



Figure 4.5 A negative cost cycle

## 4.3 An Algorithm for Min-cost Flow

With the concept of incremental network in hand, we can now specify an algorithm for the min-cost problem. At the end of the last section,the importance of negative cost cycles was emphasized. In fact, this is the key to our first algorithm.

We are to find a flow of value $v$ in a network $\mathcal{N}$ which has minimum cost. First we find some flow $\mathcal{F}$ of value $v$ using any max-flow algorithm. Next we construct the incremental network $\mathcal{N}(\mathcal{F})$ and search for a negative cost cycle. In Chapter 1, we studied an algorithm that can do this. Once we have a negative cost cycle we modify the flow $\mathcal{F}$ by passing flow around this cycle of $\mathcal{N}(\mathcal{F})$. Each change in the flow $\mathcal{F}$ produces a change in the incremental network. Eventually some arc of the cycle drops out of $\mathcal{N}(\mathcal{F})$ because it is now saturated or empty in $\mathcal{N}$. The process is repeated until a flow $\mathcal{F}$ has been reached such that there are no negative cost cycles in $\mathcal{N}(\mathcal{F})$. This is an optimal flow.

**Min-Cost Flow by Cycles**

1. Use a max-flow algorithm to determine a flow of value $v$.
2. While there is a negative cost cycle $\mathcal{C}$ in the incremental network $\mathcal{N}(\mathcal{F})$, augment flow around $\mathcal{C}$ until the cycle is no longer part of $\mathcal{N}(\mathcal{F})$.

The algorithm stops if and when it finds a flow $\mathcal{F}$ which produces an incremental network $\mathcal{N}(\mathcal{F})$ with no negative cost cycles. This leaves us with two basic questions. Does this final flow $\mathcal{F}$ indeed have minimum cost for a flow with value $v$? Is it possible that the algorithm can always find another negative cost cycle and runs on forever? These are the basic algorithmic questions of termination and correctness.

The answer to the first question is yes though we will postpone a proof until later. The answer to the second question depends on the original network. If the costs are all non-negative and the costs and

capacities are rational numbers (as we have assumed throughout) then there are only finitely many flows of any given value and each has a non-negative cost. Since the algorithm maintains the value of the flow and actually reduces the cost on each iteration, only a finite number of iterations can take place. So under these assumptions the algorithm terminates. We can generalize this argument to allow negative cost arcs as long as there is no negative cost cycle with infinite capacity.

---

EXAMPLE

As an example, let us use the Min-cost Flow by Cycles Algorithm to find a minimum cost flow of value 4 in the network of Figure 4.6. This network is already showing a feasible flow, that is, a flow of value 4. Its cost is 112. In the figures below the labels on the original network are (flow,capacity,cost) while on the incremental networks they are (capacity,cost).

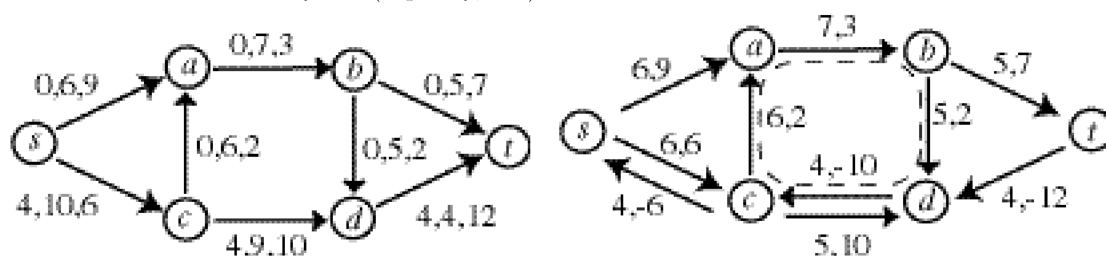



Figure 4.6 An example network and associated incremental network

The incremental network on the right has a displayed negative cost cycle that we use to modify the flow. Around this cycle, we move 4 units of flow resulting in the network in Figure 4.7. After this first iteration, the flow still has value 4 but now has cost 100.

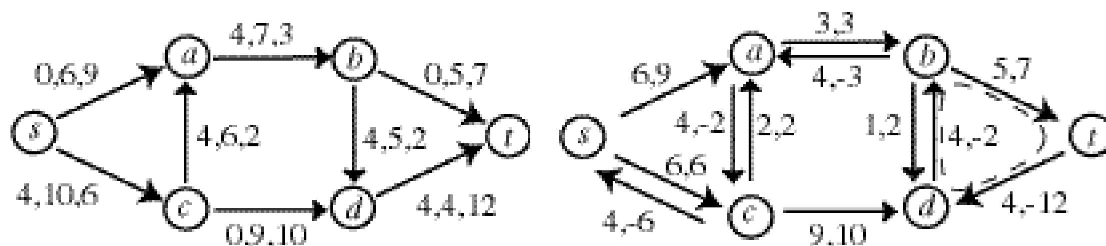



Figure 4.7 After one iteration.

Again the incremental network is constructed. A search discovers a negative cost cycle $b \to t \to d \to b$ and the flow is modified around this cycle. The network on the left in Figure 4.8 shows the resulting flow of value 4 and cost 72.

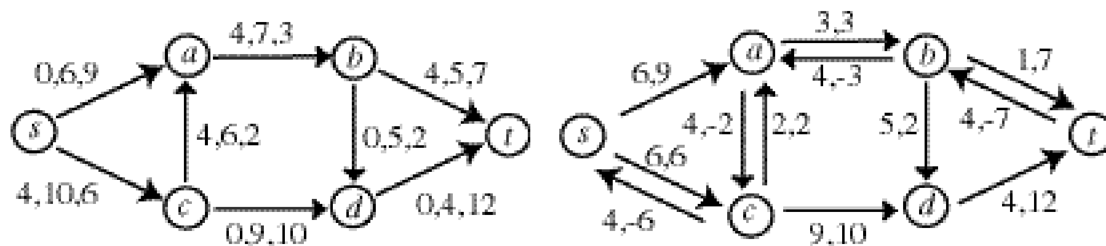



Figure 4.8 After two iterations.

The incremental network is produced (on the right in Figure 4.8) and this time there are no negative cost cycles. So after two iterations the algorithm stops. It has found a minimum cost flow of value 4. Note that if our routine that finds the negative cost cycles had been able to return a "most negative" cost cycle we would have used $a \to b \to t \to d \to c \to a$ and reached the answer in one iteration.

## 4.4 The Network Simplex Algorithm

The Cycle Algorithm for Min-cost Flows does not, in practice, have good performance. In this section we will study a different algorithm that also looks for negative cost cycles, but is able to restrict and control the search to produce a very efficient result. The Network Simplex algorithm can be viewed as a straightforward translation of the standard linear programming Simplex Algorithm to the network context. Our approach, however, is to build up a combinatorial description of the algorithm.

Consider the network in Figure 4.9. The arc labels are in the standard order $(flow, capacity, cost)$. The flows assigned to the arcs determine a feasible flow of value 10. In this network the solid arcs form a spanning tree. The arcs outside of this tree are each either empty or saturated. This is a special situation which we can exploit to efficiently move to an optimal flow. The key point is that if we add any one new arc to the tree, we create a unique cycle. By working through the arcs outside the tree, we can move through a set of cycles looking for a cycle with negative cost. If we find one then, as in the Cycle algorithm, we can pass flow around this cycle to reduce the overall cost up to the point where the cycle is broken. At that point the tree changes and we can start again.

There are several difficulties to overcome, of course. How do we find the initial tree solution? Why are we sure that a negative cost cycle of this apparently special type will occur whenever the flow is non-optimal? Before addressing this questions, though, we will give a formal presentation of the algorithm.

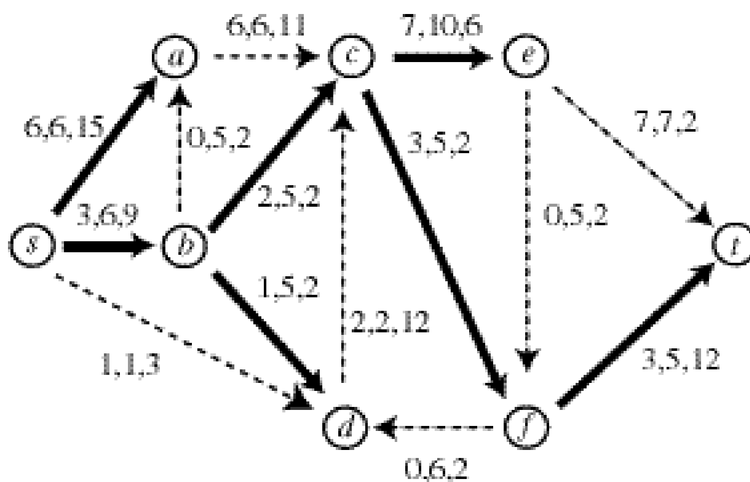



Figure 4.9 A Network with a Tree Solution

Suppose that $\mathcal{N}$ is a network with capacities and costs assigned to its arcs. Suppose also that a feasible

flow has been assigned to the arcs. A *tree solution* is a spanning tree $T$ on the arcs of $\mathcal{N}$ with the property that $f_e = 0$ or $f_e = c_e$ for each arc $e$ not in the tree.

Since $T$ is a tree there is a unique path in $T$ from $s$ to each of the nodes of $\mathcal{N}$; let $y_v$ denote the cost of the path in $T$ from $s$ to $v$. Now for an arc $e = vw$ define its *reduced cost* $\overline{c}_e$ by

$$\overline{c}_e = c_e + y_v - y_w.$$

Referring to Figure 4.9, the paths in $T$ from $s$ to $v$ and $w$ will overlap for some initial segment and then split into distinct paths (at node $u$ in the figure). When the arc $e$ is added to the tree, a cycle is formed, passing through $u$ by dropping out this common segment and the bits of the tree beyond $v$ and $w$. Denote by $C(T, e)$ the cycle formed in this way. As the algorithm proceeds, we can use $C(T, e)$ to rearrange flows with $e$ forward if this arc is empty and with $e$ reversed if it is saturated. The reduce cost $\overline{c}_e$ is the cost of $C(T, e)$ interpreted as a cycle using arc $e$ in a forward direction. If $C(T, e)$ is used in the opposite direction, with arc $e$ reversed, then the cost is $-\overline{c}_e$. Thus our search for a negative cost cycle will be successful if we ever find an arc outside the tree $T$ with reduced cost $\overline{c}_e < 0$ if $e$ is empty or with $\overline{c}_e > 0$ if $e$ is saturated.

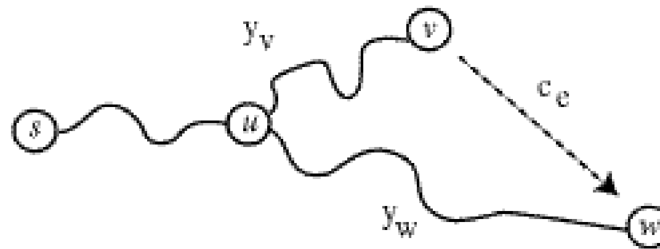



Figure 4.10 Definition of the Reduced Cost

**Min-Cost Flow by Network Simplex**

1. Determine an initial tree solution $T$ of value $v$.
2. Calculate the function $y(w)$ (cost of path from source to $w$ in $T$).
3. Look for an arc $e$ not in $T$ such that either $e$ is empty and $\overline{c}_e < 0$ or $e$ is saturated and $\overline{c}_e > 0$. If there are no such arcs then stop. $T$ is optimal.
4. Otherwise, calculate the maximum flow that can be passed around the cycle $C(T, e)$ in a forward direction if $e$ is empty or in the reverse direction if $e$ is saturated. Make this augmentation around the cycle. Some arc $h$ will become empty or saturated in the process. (This arc $h$ may be in the tree or it may be that $h = e$.)
5. If $h$ is an arc of the tree then construct a new tree by adding $e$ and deleting $h$. Update the function $y(w)$. (If $h = e$ then it is not a candidate for selection in Step 3 at the next iteration.)
6. Return to Step 3.

Those familiar with the linear programming Simplex algorithm will hear echos in the Network Simplex Algorithm. Steps 1 and 2 are the initialization step corresponding to finding an initial feasible basis for the LP problem. Then Step 3 corresponds to determining an entering variable while Steps 4 and 5 are the analogues of fixing a leaving variable and pivoting.

Next we look at why this algorithm works. It is clear from the earlier discussion that the augmentation carried out in Step 4 will result in a feasible flow of the same value but lower cost. So once we get going each iteration will move us closer to an optimal solution. But how do we get started? In the proof below a *critical arc* is an arc that is either empty or saturated.

THEOREM 4.2 *If a network has a feasible flow then it has a tree solution. If a network has a minumum cost feasible flow then there exists a tree solution which is optimal.*

*Proof.*

Suppose that $\mathcal{F}$ is a feasible flow. If there is a cycle $C$ in which every arc is non-critical, so $0 < f_e < u_e$, then an augmentation in one direction or the other is possible that will not change the flow value, will not increase the cost and will result in at least one arc becoming critical. (One says "not increase" rather than "decrease" since the cycle may have cost zero.) After this augmentation, the number of cycles without critical arcs has decreased. Eventually we will have a situation where all cycles in the network contain at least one critical arc, the flow has not increased in cost and the flow has retained the same value.

So suppose now that all cycles of the network contain at least one critical arc. We want to identify a spanning tree $T$ such that all edges not in $T$ are critical. As in one of the standard spanning tree algorithms, while there is still a cycle $C$, we delete a critical arc fron $C$. When no cycles remain, the arcs left behind form the required spanning tree.

If we started with a minimum cost flow, this process would produce a flow with the same cost. Hence an optimal tree solution exists. □

In some sense, the proof of the theorem is an algorithm for initializing the Network Simplex algorithm. There is actually quite a bit to say about implementation. See [CHVATAL].

---

**Example**

Lets return to the network at the beginning of this section and search for a minimum flow of value 10.

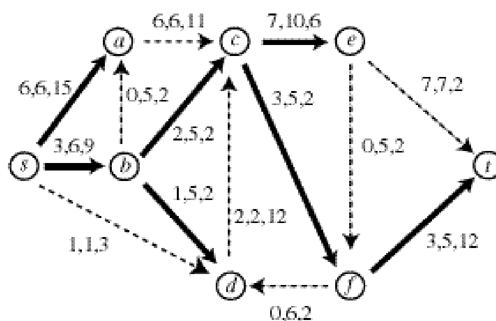


Figure 4.11 Network Simplex Example: start

| node | s | a | b | c | d | e | f | t |
|---|---|---|---|---|---|---|---|---|
| $y$ = | 0 | 15 | 9 | 11 | 11 | 17 | 13 | 25 |

In the first iteration the arc $ba$ is chosen; it is empty with reduced cost $\overline{c}_{ba} = 2 + 9 - 15 = -4$. This determines the cycle $b - a - s - b$. Then 3 units of flow are passed around this cycle reducing the cost of the flow by 12. This saturates arc $sb$ which moves out of the tree with he result as shown in Figure 4.12.

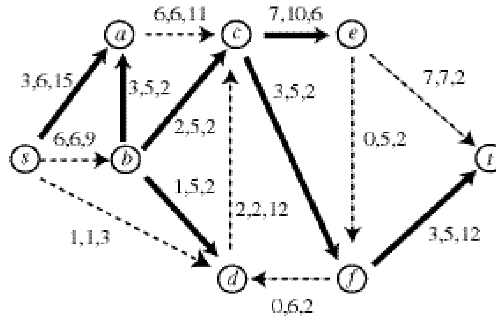



Figure 4.12 Network Simplex Example: first iteration

| node | s | a | b | c | d | e | f | t |
|---|---|---|---|---|---|---|---|---|
| $y$ = | 0 | 15 | 13 | 15 | 15 | 21 | 17 | 29 |

At the second iteration, the saturated arc $dc$ is selected; the reduced cost here is $\overline{c}_{dc} = 12 + 15 - 15 = 12$. Using the arc $dc$ backwards we pass 1 unit of flow around the cycle $d - b - c - d$. This reduces the cost of the flow by 12. The arc $dc$ enters the tree while arc $bd$ leaves. The result is Figure 4.13.

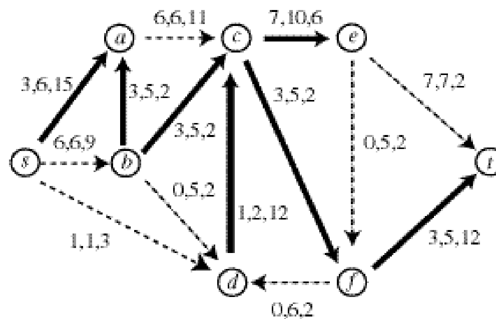



Figure 4.13 Network Simplex Example: second iteration

| node | s | a | b | c | d | e | f | t |
|---|---|---|---|---|---|---|---|---|
| $y$ = | 0 | 15 | 13 | 15 | 3 | 21 | 17 | 29 |

Is this an optimal flow of value 10? We leave that to an exercise.

EXERCISE

4.1 Complete the example. Are we done or is there another arc that can enter the tree?

This is the basic Network Simplex algorithm. There are several issues left unresolved. We need an efficient way to find an initial tree solution. One way is to add arcs that form a spanning tree but have such a high cost that the solution cannot contain any of them. We start with flow zero in all the original arcs, and the desired flow along the artifical arcs forming the path from $s$ to $t$. Now start the algorithm which will toss out all the artifical arcs on the way to an optimal solution.

We should also be concerned, at some level, about cycling. This is the phenomenon where we end up in an endless round of iterations on zero cost cycles that keep bringing us back to the same non-optimal tree solution. This is theoretically possible and there are ways of implementing the algorithm that avoid cycling. It rarely occurs in practice. See for example [BILLS] and [CHVATAL].

References:

W. Cook, W. Cunningham, W. Pulleyblank, A. Schrijver, Combinatorial Optimization, Wiley, 1998. ISBN 0-471-55894-X QA402.5.C54523

J.R. Evens and E. Minieka, Optimization Algorithms for Networks and Graphs, Marcel Dekker, 1992. ISBN 0-8247-8602-5 QA166.E92

L.R. Foulds, Combinatorial Optimization for Undergraduates, Undergraduate Texts in Mathematics, Springer, 1984. QA164.F68

W. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, 1976 (Out-of print)

C.H. Papadimitriou and K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Dover.1998 (Originally by Prentice-Hall 1982).

** There is also relevant information in some general algorithm books such as:

T.H.Cormen, Introduction to Algorithms, MIT Press 1990 (on reserve in the library)

** Other Related References also held by the MacOdrum Library:

S. Even, Graph Algorithms, Computer Science Press, 1979. ISBN 0-914894-21-8 QA166.E94

S. Even, Algorithmic Combinatorics, Macmillan, 1973. QA164.E88

A. Gibbons, Algorithmic Graph Theory, Cambridge Uni. Press, 1985. ISBN 0-521-28881-9 QA166.G53

M. Syslo, N.Deo, J.Kowalik, Discrete Optimization Algorithms, Prentice-Hall 1983. ISBN 0-13-215509-5 QA402.5 S94.

C.R. Reeves (ed.), Modern Heuristic Techniques for Combinatorial Problems, Halstead Press, 1983. ISBN 0-470-22079-1 QA402.5 M62

** Research Papers

Andrew Goldberg and Satish Rao, "Beyond the Flow Decomposition Barrier", J. A.C.M. 45 (1998) 783-797.

Dan Gusfield, "Very Simple Methods for All Pairs Network Flow Analysis", SIAM J. Comput 19 (1990) 143-155.

A. Ephremides, and S. Verdu, “Control and Optimization Methods in Communiation Network Problems”, IEEE Trans Automatic Control, 34 (1989)

Ash, Cardwell and Murray, "Design and Optimization of Networks with Dynamic Routing", Bell System Tech J. 60 (1981) 1787–1820.