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.

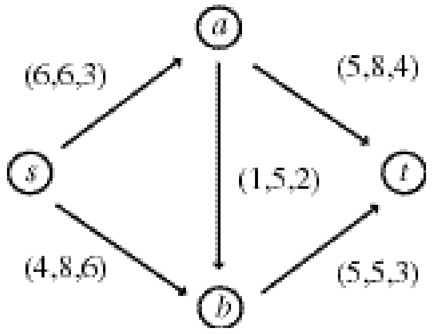

Figure 4.1 A network with arc capacities and costs

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.

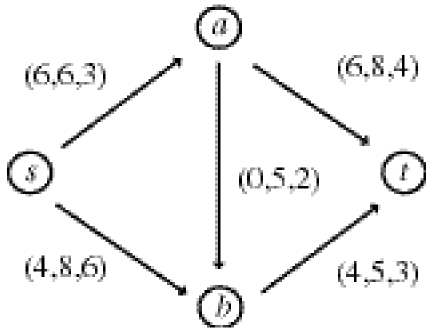

Figure 4.2 The same network with an alteration in the flow pattern

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.

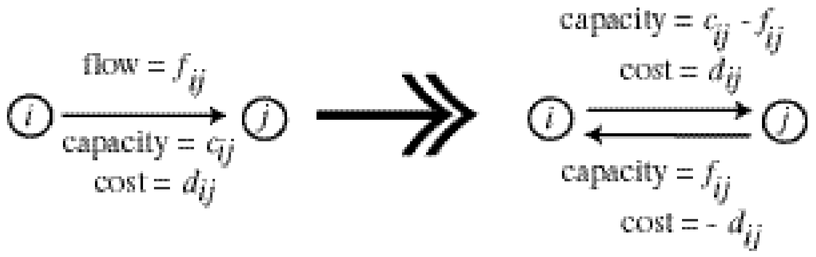

Figure 4.3 Specification of the incremental network

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

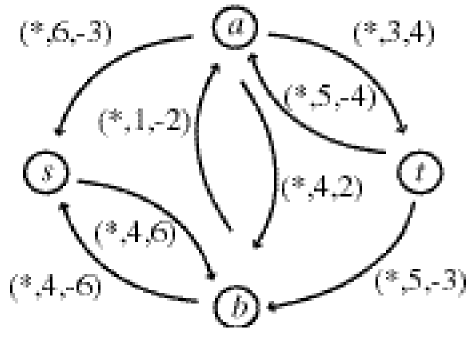

Figure 4.4 The incremental network of Figure 4.1

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.

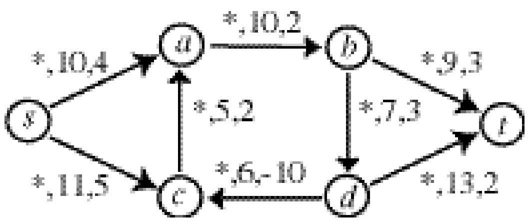

Figure 4.5 A negative cost cycle

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).

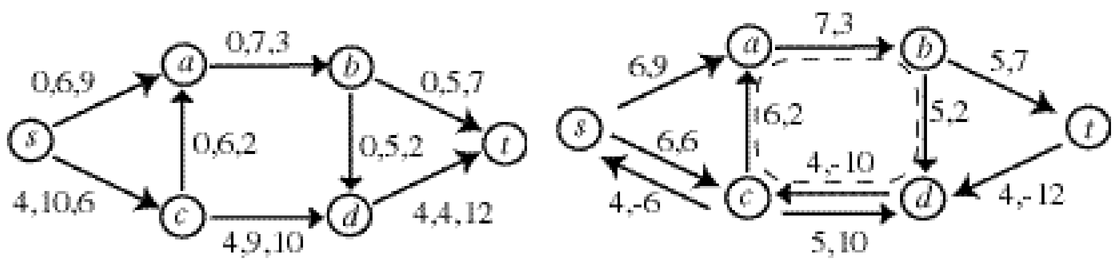

Figure 4.6 An example network and associated incremental network

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.

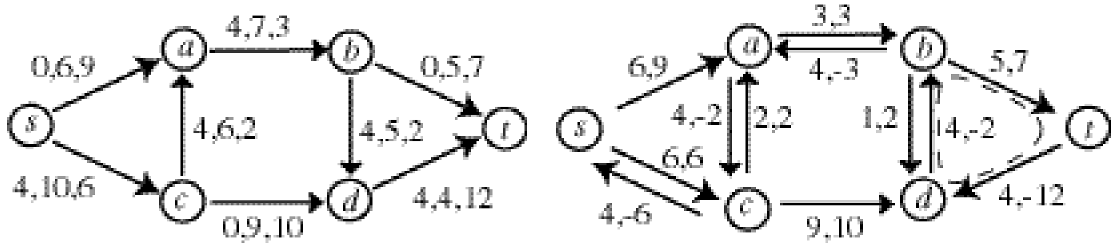

Figure 4.7 After one iteration.

Again the incremental network is constructed. A search discovers a negative cost cycle $b \rightarrow t \rightarrow d \rightarrow b$ 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.

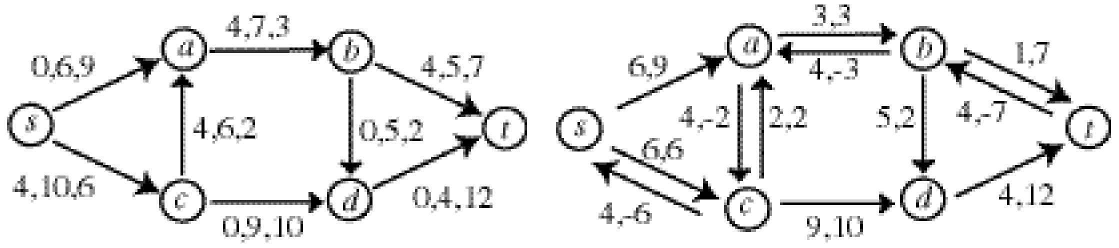

Figure 4.8 After two iterations.

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.

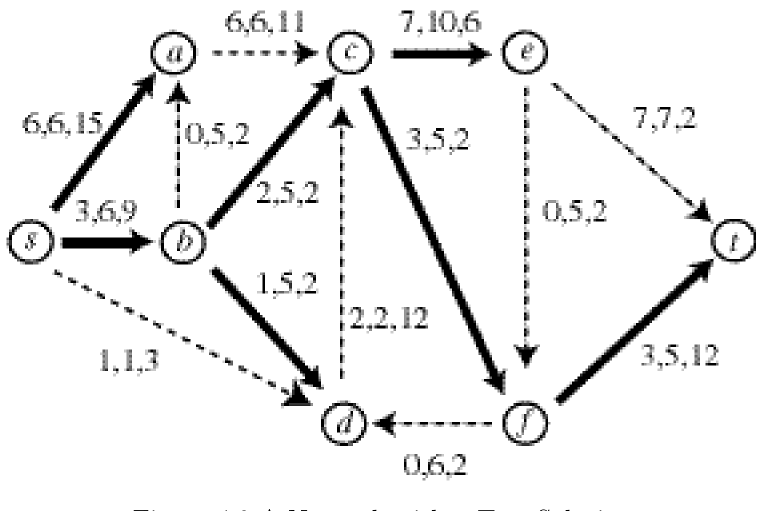

Figure 4.9 A Network with a Tree Solution

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.

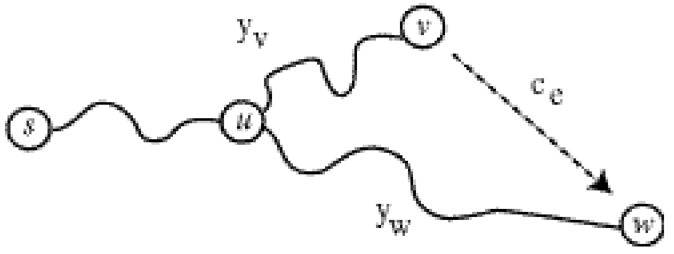

Figure 4.10 Definition of the Reduced Cost

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.

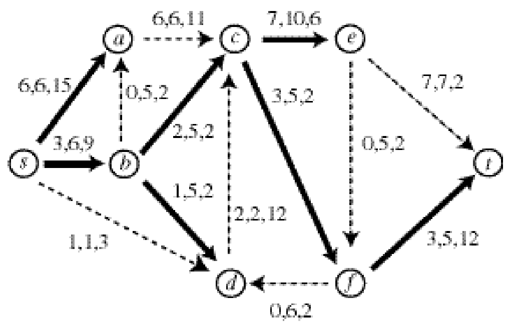

Figure 4.11 Network Simplex Example: start

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.

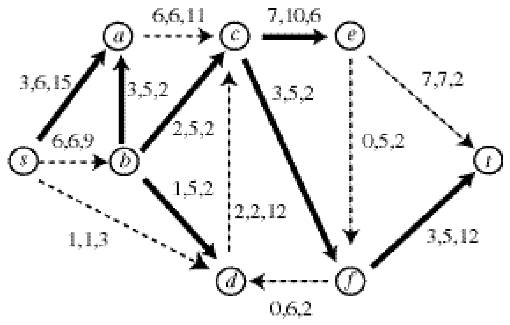

Figure 4.12 Network Simplex Example: first iteration

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.

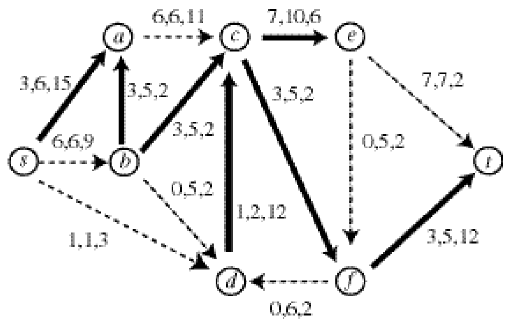

Figure 4.13 Network Simplex Example: second iteration

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].