Optimization Methods
Draft of August 26, 2005

IV.
Solving Network Problems

Robert Fourer

Department of Industrial Engineering and Management Sciences
Northwestern University
Evanston, Illinois 60208-3119, U.S.A.

(847) 491-3151

4er@iems.northwestern.edu
http://www.iems.northwestern.edu/~4er/

Introduction

You have seen how networks motivate many kinds of linear programming models. In fact, the influence of networks on operations research models is much broader than just linear programming. Some network problems cannot be solved as linear programs, and in fact are much harder to solve. Others are so easy that solving them as linear programs is more work than necessary. Still others are most efficiently solved by a network simplex method that is specialized to be much faster than the general-purpose method that you have learned.

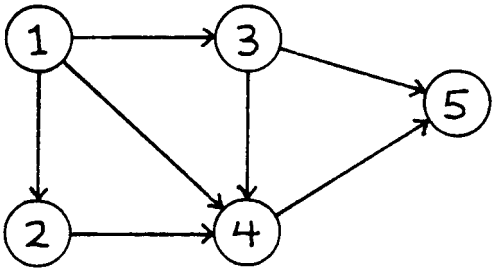
This part begins with a survey of some of the best-known network models. Then it considers the solution and analysis of different models in greater detail.

12. Network Optimization Examples

A network is defined by a set $\mathcal{N}$ of ***nodes***, and a set $\mathcal{A}$ of ***arcs*** connecting the nodes. We write $(i, j) \in \mathcal{A}$ to say that there is an arc between nodes $i \in \mathcal{N}$ and $j \in \mathcal{N}$. Where necessary, we will represent the numbers of nodes and arcs by $|\mathcal{N}|$ and $|\mathcal{A}|$.

In a ***directed*** network, the arc (i, j) is regarded as extending from node i to node j. Typically, a directed network model involves a flow or transportation of something along the arcs, in the specified directions. In an ***undirected*** network, the arc (i, j) just represents a connection between nodes i and j. An undirected network model may allow flows in either direction along an arc, or may not involve explicit flows at all.

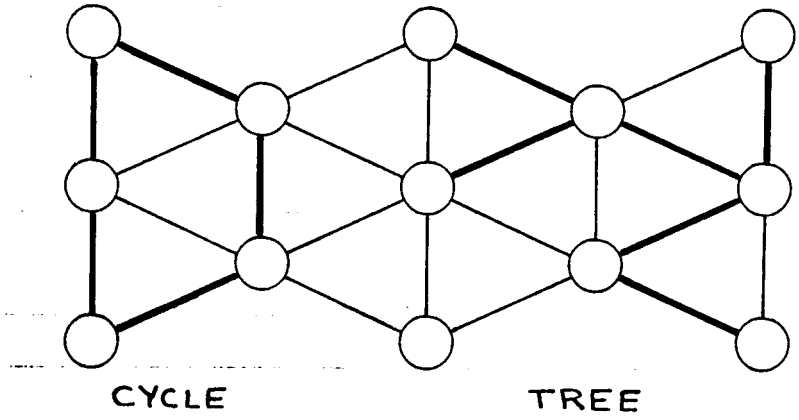
A network can be visualized by drawing the nodes as circles, and the arcs as lines between them. For a directed network, the lines are arrows pointing in the appropriate directions:



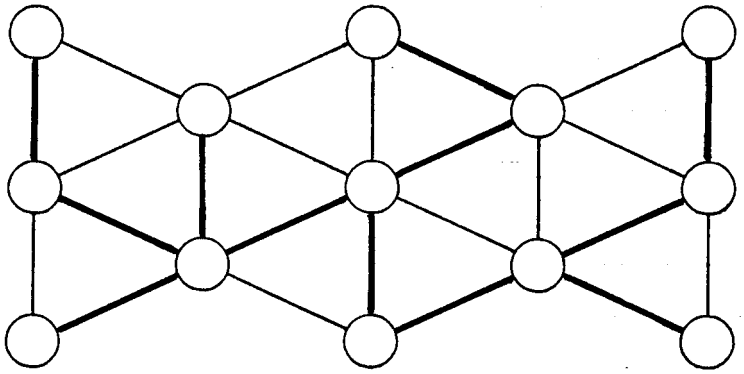
The node set here is obviously $\{1, 2, 3, 4, 5\}$. The set of arcs connecting the nodes is $\{(1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (3, 5), (4, 5)\}$.

12.1 Minimum spanning trees

A ***circuit*** in a network is a collection of arcs that are connected together in a circle, while a ***tree*** is a collection of arcs that are connected together without containing any circuits:



A ***spanning tree*** in a network is a collection of arcs that form a tree and that connect to every node. Here is an example of a spanning tree in the network above:



A spanning tree is a useful pattern for cheaply interconnecting all the nodes in a network. The number of arcs in the spanning tree equals the number of nodes minus one, and between any two nodes there is a unique path along the tree.

How cheaply can a spanning tree interconnect the nodes? Suppose that there is a cost or distance c_{ij} associated with each arc $(i, j) \in \mathcal{A}$. Then you can look for the spanning tree whose arcs have the lowest total cost or distance. In more formal algebraic terms, you want to

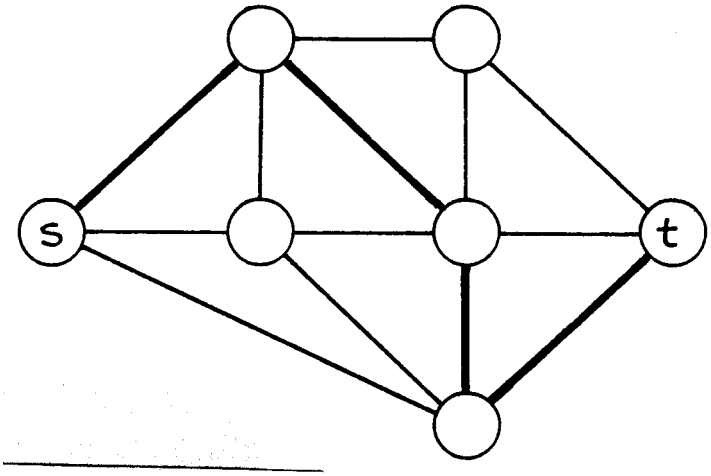
$$\underset{\mathcal{T} \subset \mathcal{A}}{\text{Minimize}} \sum_{(i,j) \in \mathcal{T}} c_{ij}$$

where the subsets $\mathcal{T}$ are limited to those that represent spanning trees within the network.

Since there are only finitely many spanning trees within a network, this minimization is well defined. However, a large network may contain a very large number of spanning trees, so it is not immediately clear how much work may be involved in finding a minimum one.

12.2 Shortest paths

A ***path*** from node $s \in \mathcal{N}$ to node $t \in \mathcal{N}$ is a sequence of arcs that lead from s to t: $(s, i_1), (i_1, i_2), (i_2, i_3), \ldots, (i_{k-1}, i_k), (i_k, t)$. A path within a network looks like this:



When there are many paths from s to t, you would naturally want to take the cheapest or shortest one. If c_{ij} represents the cost or distance of travel along arc $(i, j) \in \mathcal{A}$, then you want to solve

$$\underset{\mathcal{P} \subset \mathcal{A}}{\text{Minimize}} \sum_{(i,j) \in \mathcal{P}} c_{ij}$$

where the subsets $\mathcal{P}$ are limited to those that represent paths within the network.

Clearly, if the network is undirected, then the shortest path problem is much the same as the minimum spanning tree problem, except that it minimizes over paths rather than spanning trees. You can also imagine the problem on a directed network, however. In the directed shortest path problem, you must be able to travel a path from s to t without going "backwards" along any arc.

12.3 Travel problems

There are many problems like the shortest path problem, but minimizing costs or distances over different kinds of paths. We mention here two of the best known. Both can be defined on either an undirected or a directed network.

A circuit, as defined above, is a path from some node s back to itself. A circuit that makes one visit to every node in the network is called a *Hamiltonian* circuit. If you think of the network nodes as cities, then the problem of finding a shortest Hamiltonian circuit amounts to finding the shortest trip that visits all the cities. For this reason, it is universally known as the ***traveling salesman problem.***

A related problem is to find a circuit that does not merely visit every node, but that in fact contains every arc (some possibly more than once). Finding a shortest circuit of this kind is naturally known as the ***postman problem.***

All of the problems described so far in this section involve finding the shortest or cheapest subset of arcs from among a finite, though perhaps very large, collection. You will see later than some of them are reasonably easy to solve, while others are—in a certain sense—impossibly hard.

12.4 Maximum flows

In many kinds of directed network models, there is a flow x_{ij} associated with each arc $(i, j) \in \mathcal{A}$. A collection of flows is considered to be feasible if, for each node $j \in \mathcal{N}$, the total flow on all arcs out of j minus the total flow on all arcs into j satisfies some appropriate equation or inequality. Formally,

$$\sum_{k:(j,k) \in \mathcal{A}} x_{jk} - \sum_{i:(i,j) \in \mathcal{A}} x_{ij} \begin{Bmatrix} \leq \\ = \\ \geq \end{Bmatrix} b_j.$$

If this constraint is an equality and $b_j = 0$, then it says that flow out equals flow in; in other words, there is *conservation of flow* at node j. When b_j is positive, the flow out is greater than the flow in, so the node is a source of b_j units of flow; on the other hand, when b_j is negative, the flow in is greater than the flow out, and the node is a sink for b_j units. Interpretation of inequalities is similar.

Often there are additional constraints in the form of bounds on the flows. In general these are $l_{ij} \le x_{ij} \le u_{ij}$, where l_{ij} is a lower bound and u_{ij} is an upper bound on the flow from i to j.

A well-known simple flow problem is to maximize the total flow out of a designated source s and into a designated sink t, subject to conservation of flow at all the intermediate nodes, and upper bounds on all flows. The solution to this ***maximum flow*** problem gives the capacity of the whole network in a certain sense.

12.5 Minimum cost flows

In many models each arc (i, j) also has an associated cost c_{ij} per unit of flow. If linearity of costs can be assumed, then the total cost of flow along the arc is $c_{ij}x_{ij}$.

A minimum cost network flow model seeks a feasible flow that has the lowest total cost of flows. A simple version can be written as

$$\begin{array}{lll} \text{Minimize} & \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} & \\ \text{Subject to} & \sum_{k:(j,k)\in\mathcal{A}} x_{jk} - \sum_{i:(i,j)\in\mathcal{A}} x_{ij} = b_j, & j \in \mathcal{N} \\ & x_{ij} \ge 0, & (i,j) \in \mathcal{A} \end{array}$$

This is just a simple version of the network linear programs that you have already seen. More elaborate versions were formulated in Part I of these notes.

Some of the other network problems above are easily viewed as special cases of the minimum cost flow problem. To find the shortest path from s to t, for example, we can interpret c_{ij} as the distance from i to j; we take $b_s = 1$, $b_t = -1$, and $b_j = 0$ otherwise. Thus there is a source of one unit at s, and a sink of one unit at t, with conservation of flow elsewhere. The integrality property of network linear programs, as explained in Part I, guarantees in this particular case that the optimal value of each flow will be either $x^*_{ij} = 0$ or $x^*_{ij} = 1$. Moreover, the arcs (i, j) such that $x^*_{ij} = 1$ always form a path, while the objective value reduces to

$$\sum_{(i,j):x^*_{ij}=1} c_{ij}$$

which is the length of the shortest path.

For the maximum flow problem, no flow constraints are imposed at nodes s and t; rather, the objective is to maximize the net flow sent out of s, which equals the flow out minus any flow in:

$$\begin{array}{lll} \text{Maximize} & \sum_{k:(s,k)\in\mathcal{A}} x_{sk} - \sum_{i:(i,s)\in\mathcal{A}} x_{is} & \\ \text{Subject to} & \sum_{k:(j,k)\in\mathcal{A}} x_{jk} - \sum_{i:(i,j)\in\mathcal{A}} x_{ij} = b_j, & j \in \mathcal{N}\setminus\{s,t\} \\ & 0 \le x_{ij} \le u_{ij}, & (i,j) \in \mathcal{A} \end{array}$$

Exactly the same results are achieved by maximizing the net flow into t, which equals the flow in minus any flow out at t. In many cases s has only outgoing arcs (or t has only incoming arcs) so that the objective is just to maximize the total flow out of s (or equivalently the total flow into t).

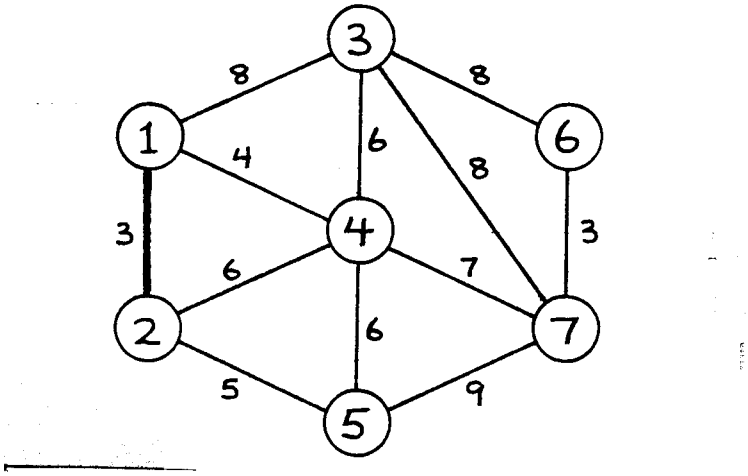
13. "Easy" Network Problems

Many network problems can be solved by specialized algorithms that are particularly simple or fast. To show the variety of these algorithms, we present examples here for the minimum spanning tree, shortest path and maximum flow problems.

13.1 Finding minimum spanning trees

An intuitively attractive way to find a "good" subset of arcs—like a spanning tree of low cost—is to build up the subset by adding one "good" arc at a time. This approach turns out to work very well for the minimum spanning tree problem.

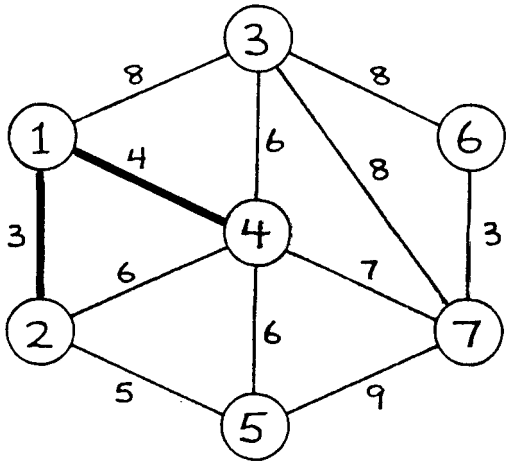
As an example, the diagram below shows an undirected network, with the labels on the arcs indicating their costs. To start, we pick one of the cheapest arcs; both (1,2) and (6,7) have a cost of 3, and we arbitrarily decide to take (1,2):



To extend the tree, we next look for the cheapest arc that is connected at either node 1 or node 2. The possibilities are:

arc	*cost*
(1,3)	8
(1,4)	4
(2,4)	6
(2,5)	5

Clearly the cheapest is (1,4), so we add it to the tree:



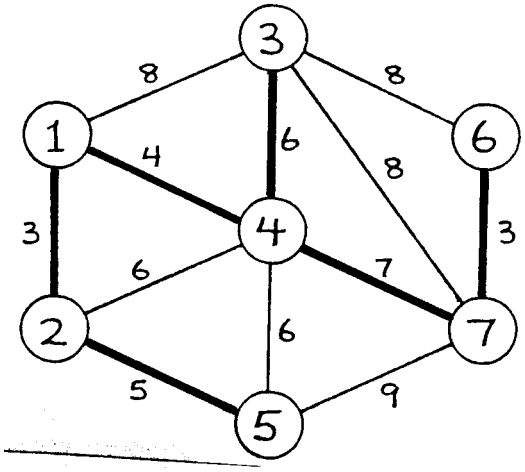
To further extend the tree, we now naturally look for the cheapest arc that is connected at nodes 1, 2 or 4:

arc	*cost*
(1,3)	8
(2,4)	$*$
(2,5)	5
(4,3)	6
(4,5)	6
(4,7)	7

Although (2,4) is connected to the tree, we cannot consider it. The problem is that it connects to the tree at both node 2 and node 4, so that its addition would form a circuit; recall that a tree must be a connected subset of arcs that contains no circuits. Thus we choose to add the cheapest of the other connected arcs, (2,5), to the tree.

You can now see how this algorithm proceeds in general. At the kth step, it has already found a tree of k arcs connecting $k+1$ nodes. It then examines all arcs (i, j) such that *one* of i and j is in the tree, and one is out, and it adds any one of the cheapest such arcs to the tree. The tree is consequently extended by one arc and one node, and the next step can begin. After $|\mathcal{N}| - 1$ steps all nodes have been connected, and by definition the tree must be a spanning tree.

In the example above, you can verify that the remaining three steps add the arcs (4,3), (4,7) and (7,6), giving the following spanning tree:



The total cost of the tree is 28.

It can be shown that this algorithm always leads to a spanning tree that has minimum cost. If there is a tie for cheapest arc at any step, it can be broken arbitrarily; in such a case, there may be more than one spanning tree that achieves the minimum for the network.

A procedure of this kind is called a *greedy* or *myopic* algorithm, because it just makes the obviously most attractive move at every iteration, without ever going back to make any adjustments. Unfortunately, greedy algorithms are seldom guaranteed to be optimal for problems of interest; the minimum spanning tree problem is one of the best known exceptions.

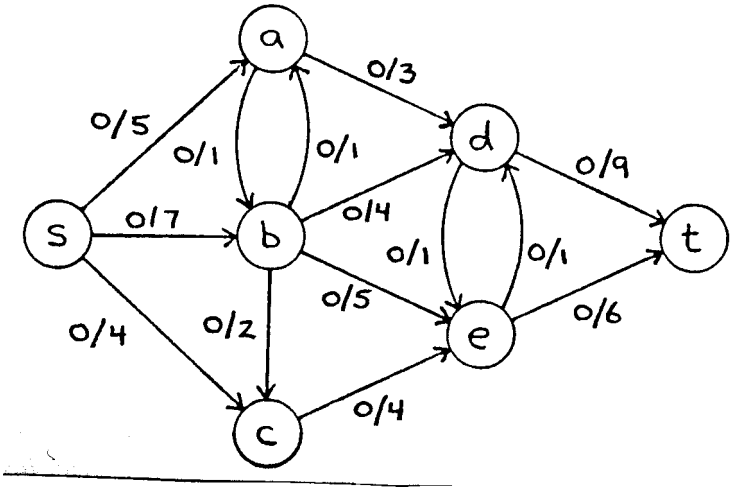
This is an easy algorithm to implement for a computer. You just need to be able to find, for every node in the tree, all arcs connected to it. This can be done efficiently by setting up an appropriate data structure. For example, for each node you might keep a list of all other nodes that are connected to it by an arc.

13.2 Finding maximum flows

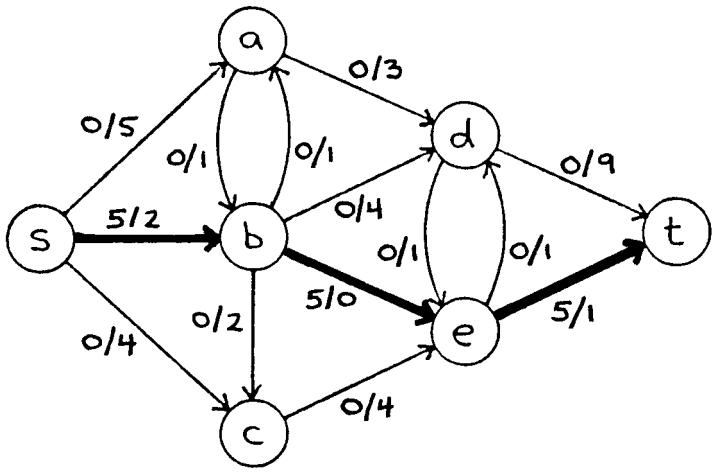
A similar kind of incremental approach can be used to try to find maximum flows between two nodes in a network. At each step some additional flow is added on a path from the origin s to the destination t, until all possible paths have reached their capacity.

In the example shown below[1] there are two labels on each arc, the first representing the amount of flow and the second the amount of unused capacity. Initially, all flows are zero, and all unused capacities equal the flow upper bounds:

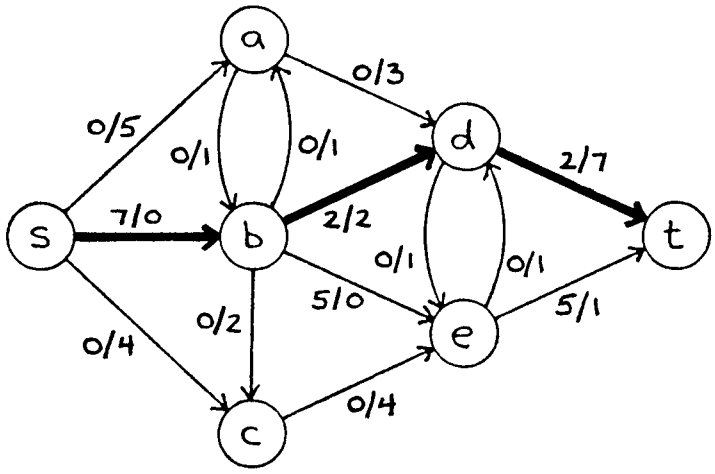
[1]Adapted from F.S. Hillier and G.J. Lieberman, *Introduction to Operations Research,* 3rd edition, pp. 241-246.



Clearly some of the flow can be handled along the path $s\ b\ e\ t$. The greatest flow that can be handled is 5, since any more will overload the arc (b, e). Suppose that we indeed put 5 units of flow on this path. All flows along the path increase by 5, and all of the unused capacities decrease by 5:



We can also send some flow along the path $s\ b\ d\ t$. Because only two units of capacity are left along the arc (s, b), the largest possible flow on the path at this point is just two units. We thus increase the flows along the path by 2, and decrease the unused capacities by the same amount:

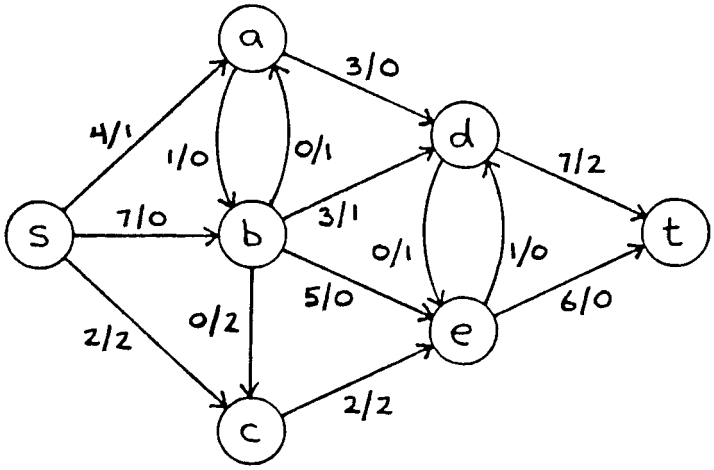


At this stage there is a total flow of 7 out of s and into t.

Additional *augmenting paths* like the ones above can be found to carry positive flows subject to the remaining capacities. We can successively add all of the following:

path	*flow*
$s\ a\ d\ t$	3
$s\ a\ b\ d\ t$	1
$s\ c\ e\ t$	1
$s\ c\ e\ d\ t$	1

The resulting flows and unused capacities are then as follows:

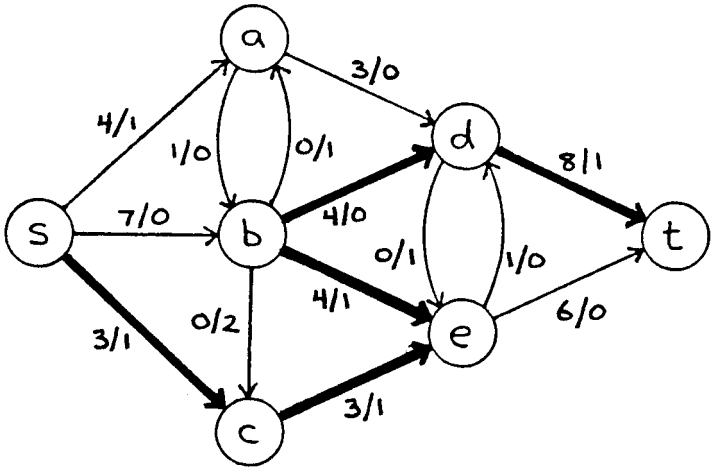


The total flow has increased to 13.

There are only a finite number of paths from s to t. If you check all of them now, you will find that every one includes an arc with no unused capacity. No further paths of improvement are available.

Even so, the flow is not a maximum! We can take one of the units of flow routed $b\ e\ t$, and send it along $b\ d\ t$ instead. Then capacity will be opened up to send one more unit on $s\ c\ e\ t$, and the flow will be increased to 14. For this problem, the obvious myopic approach has failed.

Fortunately, in this case the algorithm can be fixed up without too much trouble. You can think of the adjustment that allows the increase to 14 as adding one unit of "flow" along the "path" $s\ c\ e \leftarrow b\ d\ t$. On each forward arc, a unit of flow is added (and the unused capacity is reduced) as before; but on the backward arc $e \leftarrow b$, a unit of flow is subtracted (and the unused capacity is *increased*):



The result is an adjusted flow that is still feasible for the network, and that takes one more unit of material from s to t.

To make the algorithm work, then, we must consider all paths from s to t that contain any combination of forward and backward arcs. In particular, each step looks for a path of this kind that is augmenting in the following sense: every forward arc has a positive capacity (so that it can accept an increase in flow) and every backward arc has a positive flow (so that it can accept a decrease in flow). The amount of augmentation is equal to the smallest capacity on any forward arc, or the smallest flow on any backward arc, whichever is less. The step is completed by increasing the flow (and decreasing the unused capacity) on each forward arc by the amount of augmentation, and by decreasing the flow (and increasing the unused capacity) on each backward arc by the amount of augmentation.

It can be shown that, if no augmenting flow in this expanded sense is possible, then the flow has indeed been maximized. This condition holds in the example above, so that the maximum flow is in fact 14.

Another way to see that the maximum must be 14 in this case is to consider all the arcs from the nodes $\{s, a, b, c, e\}$ to the nodes $\{d, t\}$. Because the total *capacity* of these arcs — (a, d), (b, d), (e, d) and (e, t) — is only 14, there is no way that a flow of more than 14 could possibly get from s to t. Arcs that partition the network in this way are known as a ***cut.*** The fact that smallest capacity of any cut is 14 in this case is not an accident, for the following can be proved:

> *Max-flow min-cut theorem:* For any maximum flow problem, the capacity of the minimum cut is equal to the maximum flow.

Actually, this is just a special case of linear programming duality. If you take the dual of the linear program that is equivalent to the maximum flow problem, as described in the previous section, then you get a linear program that is equivalent to the minimum cut problem. The above theorem follows immediately from the fact that the two linear programs have the same optimal objective value.

In implementing the augmenting path method on a computer, there are two crucial considerations: to find augmenting paths quickly, and to avoid having to look for them too many times. Very efficient and clever schemes for both these purposes have been discovered. There exist versions that require an effort only proportional to $|\mathcal{N}|^3$, regardless of how many arcs there are. Other methods can be even more efficient when the network is sufficiently *sparse* that $|\mathcal{A}|$ is much less than $|\mathcal{N}|^2$ (where $|\mathcal{N}|^2$ is approximately the largest number of arcs that a network of $|\mathcal{N}|$ nodes can have).

13.3 Finding shortest paths

Shortest path problem is another network problem that can be solved by linear programming, but that has other faster methods of solution. The idea is to first find the shortest path from s to the closest of all nodes, then the shortest path to the next-closest node, and so forth until the shortest path to t is discovered.

To explain more precisely how this approach works, we'll use the following terminology pertaining to the kth-closest node:

s_k the kth closest node to s
S_k the set containing s and all the k closest nodes to s

We'll also use

c_{ij} the length of the arc (i, j)
ℓ_i the length of the shortest path from s to node $i \in \mathcal{N}$

Naturally $c_{ij} \geq 0$ for all arcs (i, j), and the shortest path from node s to itself is $\ell_s = 0$. Furthermore, if we write shortest distance from s to any other node as

$$c_{sq} = \min_{(s,j)\in\mathcal{A}} c_{sj},$$

then q is the closest node to s; so we easily find $s_1 = q$, $S_1 = \{s, q\}$ and $\ell_q = c_{sq}$. The challenge is to now continue by finding s_2, s_3, and so forth. In general, we want a way to find s_{k+1} and the shortest path to it, once we have already found the shortest paths to all the nodes in S_k; then $S_{k+1} = S_k \cup \{s_{k+1}\}$.

To see how this is done, observe first that all the nodes along the path to s_{k+1}, the $(k+1)$st closest node, are even closer to s. Thus

> All nodes on the on the shortest path from s to s_{k+1} are in S_k, except for s_{k+1} itself.

What then does the shortest path to the $(k+1)$st closest node look like? It might be just one arc from s to s_{k+1}. In general, however, it consists of a path from s through some nodes in S_k, plus one more arc from a node in S_k to s_{k+1}.

Thus, in looking for the $(k+1)$st closest node, we can confine our search to the set of all arcs that connect a node in S_k to a node not in S_k. We'll call the set of all such arcs $\mathcal{T}_k$:

$$\mathcal{T}_k \quad \text{the set of all arcs } (i,j) \in \mathcal{A} \text{ such that } i \in S_k \text{ but } j \notin S_k$$

What is the length of the shortest path that ends with some particular arc $(i,j) \in \mathcal{T}_k$? You can find it by taking the length of the shortest path to i, which you already know is ℓ_i since $i \in S_k$, and then adding on the length of (i,j), which is c_{ij}. Hence you know that the shortest path ending with $(i,j) \in \mathcal{T}_k$ has length $\ell_i + c_{ij}$.

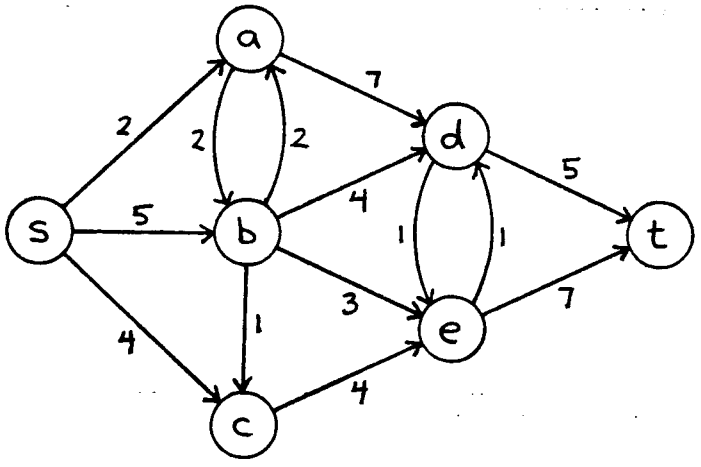
Now we are prepared to claim that the $(k+1)$st closest node can be found by solving:

$$\min_{(i,j)\in\mathcal{T}_k} \ell_i + c_{ij}.$$

Suppose the minimum occurs for a node $(q,r) \in \mathcal{T}_k$. Then the $(k+1)$st closest node $s_{k+1} = r$, and the length of the path to it is $\ell_q + c_{qr}$. The actual shortest path is found by just taking the shortest path from s to q, which is known, and appending (q,r).

The shortest path algorithm consists of up to $|\mathcal{N}|$ steps. At the kth step, S_{k+1} is determined from S_k by the minimization described above. If the shortest path from s to t is wanted, then the algorithm stops as soon as s_{k+1} happens to be t at some step. If the algorithm is continued for all $|\mathcal{N}|$ steps, then it finds a shortest path from s to all other nodes (assuming all can be reached over the network from s).

As an example, we'll use the following network similar to the one above:



It's obvious that the closest node to s is a, at a distance of 2, so for step one we take $s_1 = a$, $S_1 = \{s,a\}$ and $\ell_s = 0$, $\ell_a = 2$.

To begin step two, we must consider all arcs from nodes in $S_1 = \{s,a\}$ to nodes outside it:

$$\mathcal{T}_1 = \{(s,b),(s,c),(a,b),(a,d)\}.$$

We seek the minimum of $\ell_i + c_{ij}$ over all arcs $(i,j) \in \mathcal{T}_1$:

$$\begin{array}{ll} (s,b): & \ell_s + c_{sb} = 0 + 5 = 5 \\ (s,c): & \ell_s + c_{sc} = 0 + 4 = 4 \\ (a,b): & \ell_a + c_{ab} = 2 + 2 = 4 \\ (a,d): & \ell_a + c_{ad} = 2 + 7 = 9 \end{array}$$

The minimum is achieved by both (s,c) and (a,b), from which it follows that both b and c are second-closest. Thus we can combine steps two and three by setting $s_2 = b$, $s_3 = c$, with $\ell_b = \ell_c = 4$ and $S_3 = \{s,a,b,c\}$. The shortest path to b is given by the shortest path to a together with (a,b), or $s\ a\ b$; the shortest path to c is just $s\ c$.

Having combined steps two and three, we begin step four by considering all arcs from nodes in $S_3 = \{s,a,b,c\}$ to nodes outside it:

$$\mathcal{T}_3 = \{(a,d),(b,d),(b,e),(c,e)\}.$$

We seek the minimum of $\ell_i + c_{ij}$ over all arcs $(i,j) \in \mathcal{T}_3$:

$$\begin{array}{ll} (a,d): & \ell_a + c_{ad} = 2 + 7 = 9 \\ (b,d): & \ell_b + c_{bd} = 4 + 4 = 8 \\ (b,e): & \ell_b + c_{be} = 4 + 3 = 7 \\ (c,e): & \ell_c + c_{ce} = 4 + 4 = 8 \end{array}$$

The minimum is achieved by (c,e), so e is the fourth-closest node. We set $s_4 = e$, with $\ell_e = 7$ and $S_4 = \{s,a,b,c,e\}$. The shortest path to e is given by the shortest path to b together with (b,e), or $s\ a\ b\ e$.

Step five considers all arcs from nodes in $S_4 = \{s,a,b,c,e\}$ to nodes outside it:

$$\mathcal{T}_4 = \{(a,d),(b,d),(e,d),(e,t)\}.$$

We seek the minimum of $\ell_i + c_{ij}$ over all arcs $(i,j) \in \mathcal{T}_4$:

$$\begin{array}{ll} (a,d): & \ell_a + c_{ad} = 2 + 7 = 9 \\ (b,d): & \ell_b + c_{bd} = 4 + 4 = 8 \\ (e,d): & \ell_e + c_{ed} = 7 + 1 = 8 \\ (e,t): & \ell_e + c_{et} = 7 + 7 = 14 \end{array}$$

The minimum is achieved by both (b,d) and (e,d), so d is the fifth-closest node. We set $s_5 = d$, with $\ell_d = 8$ and $S_5 = \{s,a,b,c,d,e\}$. There is a tie for the shortest path to d: either the shortest path to b together with (b,d), giving $s\ a\ b\ d$; or the shortest path to e together with (e,d), giving $s\ a\ b\ e\ d$.

At the final step, only the shortest path to t remains to be determined. We consider all arcs from the other nodes (which are all in S_5 now) to t, the one node outside S_5:

$$\mathcal{T}_5 = \{(d,t),(e,t)\}.$$

We look at the minimum of $\ell_i + c_{ij}$ over all arcs $(i,j) \in \mathcal{T}_5$:

$$(d,t): \quad \ell_d + c_{dt} = 8 + 5 = 13$$
$$(e,t): \quad \ell_e + c_{et} = 7 + 7 = 14$$

The minimum is achieved by (d,t). The shortest path to t, of length 13, consists of the shortest path to d together with (d,t). Since there are two shortest paths to d, the solution is given by either *s a b d t* or *s a b e d t*.

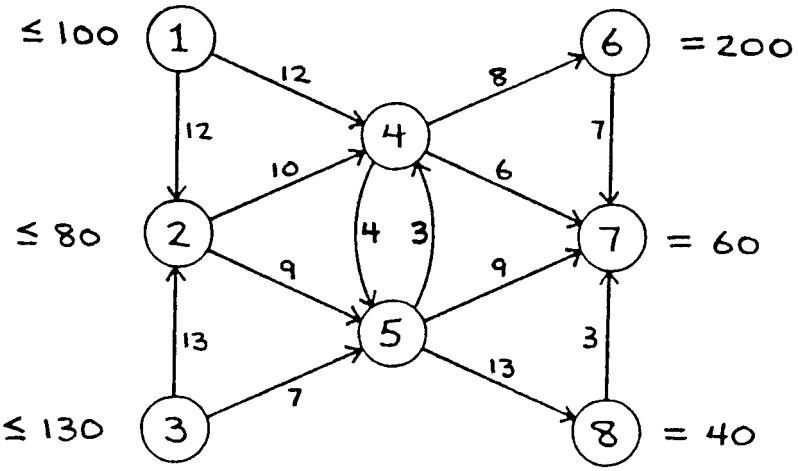
For this small example, $\mathcal{T}_k$ can be determined by observation at each step. Computer implementations can do this implicitly, in a way that keeps the amount of work proportional to only $|\mathcal{N}|^2$.

The idea of this algorithm, in which a series of optimal solutions is built up until the desired solution is found, is known as ***dynamic programming.*** A variety of other problems in deterministic and stochastic modeling can be handled by the same kind of approach.

14. The Network Simplex Method

You have seen how a variety of situations can be modeled as network flow linear programs. As you would expect, network linear programs can be solved readily by application of the simplex method. In fact, that the simplex method can be streamlined in several significant ways when it is applied to a problem that is known to have a network structure. The resulting network simplex method is so different that it is commonly studied as a separate algorithm.

We will use an example to show what an iteration of the network simplex method is like. The network for the example is as follows:



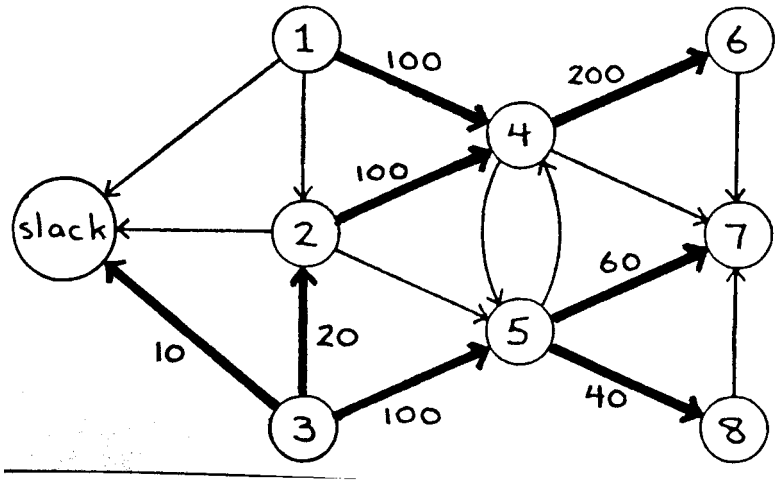
The unit cost of flow is indicated next to each arc. Material is available at nodes 1, 2 and 3, in the amounts shown at the left, and is required at nodes 6, 7 and 8 in the amounts shown at the right; nodes 4 and 5 are transshipment points.

The minimum cost flow problem for this network can be represented by the following linear program:

$$
\begin{array}{lrcl}
\text{Minimize} & 12x_{12}+12x_{14}+10x_{24}+9x_{25}+13x_{32}+7x_{35}+4x_{45}+8x_{46}+6x_{47}+3x_{54}+9x_{57}+13x_{58}+7x_{67}+3x_{87} & & \\
\text{Subject to} & +x_{12}+x_{14}+s_1 & = & 100 \\
& -x_{12}+x_{24}+x_{25}-x_{32}+s_2 & = & 80 \\
& +x_{32}+x_{35}+s_3 & = & 130 \\
& -x_{14}-x_{24}+x_{45}+x_{46}+x_{47}-x_{54} & = & 0 \\
& -x_{25}-x_{35}-x_{45}+x_{54}+x_{57}+x_{58} & = & 0 \\
& -x_{46}+x_{67} & = & -200 \\
& -x_{47}-x_{57}-x_{67}-x_{87} & = & -60 \\
& -x_{58}+x_{87} & = & -40 \\
& x_{12},x_{14},x_{24},x_{25},x_{32},x_{35},x_{45},x_{46},x_{47},x_{54},x_{57},x_{58},x_{67},x_{87},s_1,s_2,s_3 & \geq & 0
\end{array}
$$

The first three constraints say that the net flows out of source nodes 1, 2 and 3 must be $\leq$ the amounts available; slack variables s_1, s_2 and s_3 have been added to produce equalities. The next two constraints insure that flow in equals flow out at nodes 4 and 5. Finally, the last three constraints (when multiplied through by -1) say that net flows into sink nodes 6, 7 and 8 must equal the amounts required.

To illustrate one iteration, we consider the situation in which the simplex method has reached the basic solution $\bar{x}_B = (\bar{x}_{14}, \bar{x}_{24}, \bar{x}_{32}, \bar{x}_{35}, \bar{x}_{46}, \bar{x}_{57}, \bar{x}_{58}, \bar{s}_3) = (100, 100, 20, 100, 200, 60, 40, 10)$. The positive flows along the arcs in this solution have the following appearance:



(An extra arc has been drawn from node 3 to an imaginary "slack" node, to represent the 10 units that are unused.) You can see that the arcs corresponding to the basic variables form a spanning tree in the network, as defined in Section 1 above. In fact, every basis for the linear program corresponds to some spanning tree in the network.

To begin the simplex method, we must solve the linear system $\pi B = c_B$. Extracting just the coefficients of the basic variables from the linear program, we have

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ -1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix}$$

and $c_B = (12, 10, 13, 7, 8, 9, 13, 0)$. Each column of B corresponding to a basic variable x_{ij} has only two nonzero elements, a $+1$ in row i and a -1 in row j; each column of B corresponding to a basic variable s_i has only a 1 in row i. The corresponding components in c_B are c_{ij} for a basic x_{ij}, and zero for a basic s_i (since slack variables have a coefficient of zero in the objective).

The equations $\pi B = c_B$ thus reduce to a system that has a particularly simple structure. In the case of our example, these equations are

$$\begin{aligned}
\pi_1 - \pi_4 &= 12 \\
\pi_2 - \pi_4 &= 10 \\
\pi_3 - \pi_2 &= 13 \\
\pi_3 - \pi_5 &= 7 \\
\pi_4 - \pi_6 &= 8 \\
\pi_5 - \pi_7 &= 9 \\
\pi_5 - \pi_8 &= 13 \\
\pi_3 &= 0
\end{aligned}$$

and a solution can be found immediately by substitution:

$$\begin{aligned}
\pi_3 &= 0 \quad \text{and} \quad \pi_3 - \pi_5 = 7 \quad \Rightarrow \quad \pi_5 = -7 \\
\pi_5 &= -7 \quad \text{and} \quad \pi_5 - \pi_8 = 13 \quad \Rightarrow \quad \pi_8 = -20 \\
\pi_5 &= -7 \quad \text{and} \quad \pi_5 - \pi_7 = 9 \quad \Rightarrow \quad \pi_7 = -16 \\
\pi_3 &= 0 \quad \text{and} \quad \pi_3 - \pi_2 = 13 \quad \Rightarrow \quad \pi_2 = -13 \\
\pi_2 &= -13 \quad \text{and} \quad \pi_2 - \pi_4 = 10 \quad \Rightarrow \quad \pi_4 = -23 \\
\pi_4 &= -23 \quad \text{and} \quad \pi_4 - \pi_6 = 8 \quad \Rightarrow \quad \pi_6 = -31 \\
\pi_4 &= -23 \quad \text{and} \quad \pi_1 - \pi_4 = 12 \quad \Rightarrow \quad \pi_1 = -11
\end{aligned}$$

Thus we have $\pi = (\pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_6, \pi_7, \pi_8) = (-11, -13, 0, -23, -7, -31, -16, -20)$. If you imagine "walking around the tree" starting from below node 3 and proceeding to the right, you will see that you come to the nodes in the order 3, 5, 8, 7, 2, 4, 6, 1. This is just the same as the order in which the π values were found by substitution above; or, looking at matters from the opposite perspective, the substitution order for computing the π values can be found by just walking around the tree.

Next we have to compute the reduced costs. Again, we take advantage of the simple structure of the constraints: the vector of coefficients a_{ij} for a nonbasic x_{ij} has only two nonzeros, a $+1$ as the ith component and a -1 as the jth component; the vector of coefficient a_i for a nonbasic s_i has only one nonzero, a 1 as the ith component. Thus the formulas for the reduced costs simplify to

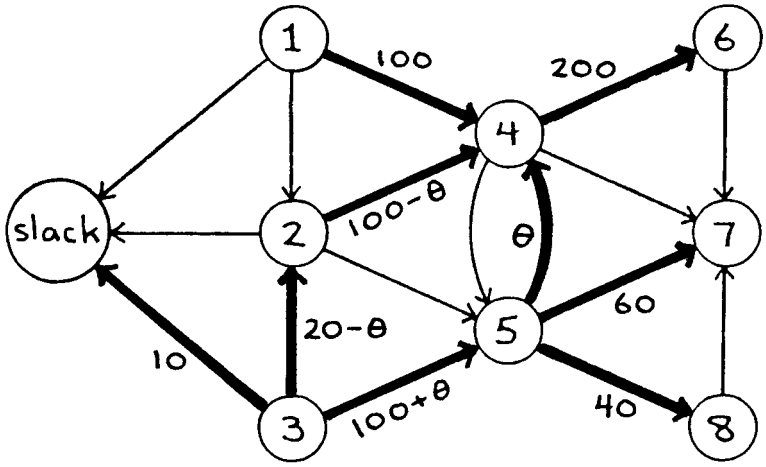
$$\begin{aligned}
d_{12} &= c_{12} - \pi_1 + \pi_2 = 12 - (-11) + (-13) = 10 \\
d_{25} &= c_{25} - \pi_2 + \pi_5 = 9 - (-13) + (-7) = 15 \\
d_{45} &= c_{45} - \pi_4 + \pi_5 = 4 - (-23) + (-7) = 20 \\
d_{54} &= c_{54} - \pi_5 + \pi_4 = 3 - (-7) + (-23) = -13 \\
d_{47} &= c_{47} - \pi_4 + \pi_7 = 6 - (-23) + (-16) = 13 \\
d_{67} &= c_{67} - \pi_6 + \pi_7 = 7 - (-31) + (-16) = 22 \\
d_{87} &= c_{87} - \pi_8 + \pi_7 = 3 - (-20) + (-16) = 7
\end{aligned}$$

$$\begin{aligned}
d_1 &= 0 - \pi_1 = -(-11) = 11 \\
d_2 &= 0 - \pi_2 = -(-13) = 13
\end{aligned}$$

Only $d_{54} < 0$, so only x_{54} is eligible to enter the basis at this iteration.

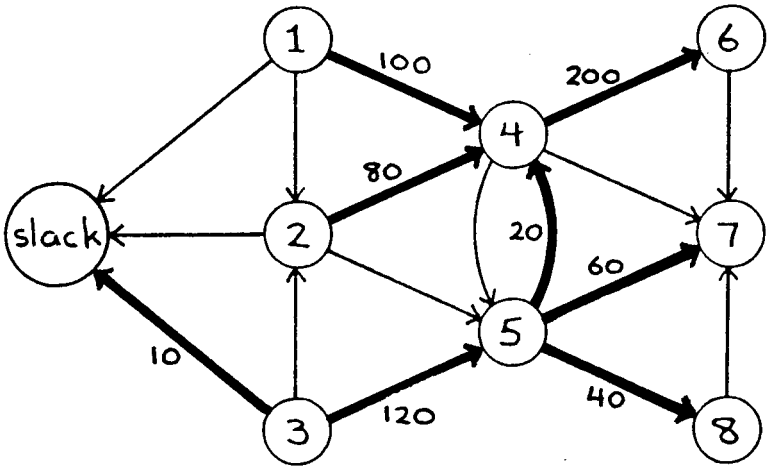
To determine the leaving variable, we could proceed to solve a linear system like $By_{54} = a_{54}$, and then to carry out a minimum ratio test. However, it is much easier to do the same thing by just looking at the network. When the entering variable x_{54} is increased from zero, the arc (5,4) is added to the basic tree, creating a unique circuit consisting of x_{54}, x_{24}, x_{32} and x_{35}. To maintain

feasibility of all flows when flow is added to (5,4), it suffices to adjust the flows along the resulting circuit as shown in this diagram:



Examining the circuit, we see that (3,5) is oriented in the same direction as the entering arc (5,4); as a result, when x_{54} increases from 0 to θ, x_{35} increases from 100 to $100 + \theta$. On the other hand, since (2,4) and (3,2) are oriented in the opposite direction from (5,4), x_{24} decreases from 100 to $100 - \theta$ and x_{32} decreases from 20 to $20 - \theta$.

How much can we increase the flow θ on (5,4)? As in the general simplex method, we look to see what flows might fall to zero. The flow on (3,5) only increases; the flow on (2,4) falls to zero when θ reaches 100, and the flow on (3,2) falls to zero when θ reaches 20. Thus we can increase x_{54} up to 20, at which point x_{32} becomes zero and drops out of the basis. Equivalently, the arc (3,2) drops out of the circuit, producing a new basis tree:



You can check that this new flow is less costly by exactly $-\theta d_{54} = 260$.

Every iteration of the network simplex method is just as simple as the one in our example. In general, the equations $\pi B = c_B$ reduce to

$$\begin{aligned} \pi_i - \pi_j &= c_{ij} \quad \text{for each basic } x_{ij} \\ \pi_i &= 0 \quad\quad\quad\ \ \text{for each basic } s_i \end{aligned}$$

It can be proved that, because the basis corresponds to a tree, these equations can always be solved by simple substitution. Moreover, the order of substitution can always be found by just "walking around the tree" as in the example. (Efficient implementations further reduce the cost of determining π by updating it as they walk around the tree, rather than computing it anew at each iteration.)

Once π is known, the general formula for the reduced costs is also very simple:

$$\begin{aligned} d_{ij} &= c_{ij} - \pi_i + \pi_j \quad \text{for each nonbasic } x_{ij} \\ d_i &= -\pi_i \qquad\qquad\quad \text{for each nonbasic } s_i \end{aligned}$$

There is a reduced cost for every nonbasic arc, whereas the rest of an iteration works mainly with basic arcs, which correspond in number to the nodes. Large networks commonly have many more arcs than nodes, as a result of which the reduced cost computation becomes the dominant cost of an iteration. To keep this cost low, only a limited subset of reduced costs is computed at most iterations, and the entering variable is chosen to correspond to the most negative reduced cost in the subset.

The rest of the work of an iteration consists of finding the circuit created in the tree by the entering arc, then tracing around the circuit to determine which arc should leave. The leaving arc is always the one that has minimum flow among all those in the circuit that are oriented opposite to the direction of the entering arc. The new flow along the entering arc is always equal to the old flow on the leaving arc (which falls to zero); each arc on the circuit has its flow either increased by this same amount (if it is oriented in the same direction as the entering arc) or decreased by this amount (if it is oriented oppositely). The new basis is always a new tree that gives a new feasible flow.

When the network simplex steps are to be carried out by a computer, it is not so obvious how the walk around the tree can be carried out, or how the circuit induced by the entering arc can be found. A few concise and clever data structures are used to represent the basis tree in a way that allows these operations to be efficient. The data structures can themselves be efficiently updated as the tree changes from iteration to iteration.

15. "Hard" Network Problems

Having looked at some ways in which classic network problems can be solved, we now turn to some situations in which solutions are very difficult to find. We first show how even an easy problem like minimum spanning tree can be challenging to solve as a linear program. Then we introduce a surprising class of problems that seem to resist solution by any means.

15.1 Intractable linear programs

Interest in network modeling increased greatly beginning about 1950, when computers started to become available. At first, it was hoped that some kind of linear programming formulation or technique could be used to efficiently solve any kind of network problem. Then although there might be some problems (like maximum flow or shortest path) that could be solved even faster by specialized algorithms, at least the simplex algorithm could be relied upon in every case.

It quickly became apparent, however, that even certain very easy network problems cannot be solved efficiently as linear programs. Consider the minimum spanning tree problem, which you saw can be solved by a simple greedy algorithm. To model this problem as a linear program, we can aim for a solution in which

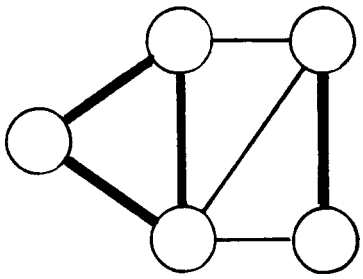
$$x_{ij} = \begin{cases} 1 & \text{if arc } (i,j) \in \mathcal{A} \text{ is in the spanning tree} \\ 0 & \text{if arc } (i,j) \in \mathcal{A} \text{ is } \textit{not} \text{ in the spanning tree} \end{cases}$$

The objective, equal to the cost of all arcs in the tree, could then be taken as the sum of $c_{ij}x_{ij}$ over all arcs $(i,j) \in \mathcal{A}$. The most obvious constraint is that the number of arcs must equal the number of nodes minus one:

$$\sum_{(i,j)\in\mathcal{A}} x_{ij} = |\mathcal{N}| - 1.$$

Any tree satisfying this property must be a spanning tree.

Further constraints are needed to ensure that the arcs (i,j) having $x_{ij} = 1$ actually form a tree, however, rather than some other structure:



The difficulty with such solutions is that, although they contain 4 arcs—one less than the number of nodes—they incorporate a circuit within some subset of the

nodes. To deal with this, we observe that if there is a circuit through exactly some subset C of the nodes, then there are $|C|$ arcs connecting the nodes of C, whereas in a spanning tree there could be at most $|C| - 1$ arcs connecting the nodes of C. Thus we can rule out the undesirable circuit by requiring that less than $|C|$ arcs be used among all those that connect nodes from C. To say this in algebra, let $\mathcal{A}_C \subset \mathcal{A}$ be the subset of connecting arcs:

$$\mathcal{A}_C = \{(i,j) \in \mathcal{A} : i \in C \text{ and } j \in C\}.$$

Then what we want to require is that

$$\sum_{(i,j)\in\mathcal{A}_C} x_{ij} \le |C| - 1$$

for every subset $C \subset \mathcal{N}$.

Putting all this together, it would seem that a linear program for the minimum spanning tree problem might be as follows:

$$\begin{array}{lll} \text{Minimize} & \sum_{(i,j)\in\mathcal{A}} c_{ij}x_{ij} & \\ \text{Subject to} & \sum_{(i,j)\in\mathcal{A}} x_{ij} = |\mathcal{N}| - 1 & \\ & \sum_{(i,j)\in\mathcal{A}_C} x_{ij} \le |C| - 1, & \text{for all } C \subset \mathcal{N} \\ & 0 \le x_{ij} \le 1, & \text{for all } (i,j) \in \mathcal{A} \end{array}$$

In fact, it can be shown that these constraints are sufficient. In principle, you can solve the minimum spanning tree problem by solving this linear program.

There is a catch, however. How many subsets $C \subset \mathcal{N}$ are there? For a network of $|\mathcal{N}|$ nodes, there are at most $\frac{1}{2}|\mathcal{N}|(|\mathcal{N}| - 1)$ arcs (in the case where every pair of nodes is connected) but there are $2^{|\mathcal{N}|}$ subsets C of the nodes. For $|\mathcal{N}| = 100$, the number of arcs is at most 4950, which is manageable for running the greedy algorithm of Section 2. However, the number of subsets C is $2^{100} \approx 1.27 \cdot 10^{30}$, so that it makes no sense to even think about solving the linear program.

We can save some constraints by considering only subsets of 2 or more nodes for which $\mathcal{A}_C$ is *connected,* in the sense that there is a path between any two of its nodes. Unless there are very few arcs, however, the number of constraints in the linear program will still be impossibly large for even moderate-sized networks.

There are other linear programming formulations of the minimum spanning tree problem that avoid an exponential blowup in the number of variables or constraints. It is far from obvious how to construct these formulations, however; and even if one is given to you, it is hard to see, at first, that its optimal solution is guaranteed to be a minimum spanning tree.

15.2 Intractable network problems

Even if linear programming will not solve every network problem, it is still possible to believe that every problem can be solved by some efficient algorithm—if only someone is clever enough to find it. There is surprisingly strong evidence, however, that many problems admit no generally efficient method of solution.

What should be considered an efficient algorithm? The most widely used criterion was initially suggested in a 1965 paper (entitled "Paths, Trees and Flowers") by Jack Edmunds:

> The amount of work required by a "good" algorithm is bounded by a *polynomial* function of the problem size.

For example, an algorithm that requires at most $|\mathcal{N}|$ or $12|\mathcal{N}|^2$ or even $|\mathcal{N}|^4|\mathcal{A}|$ operations is considered a relatively efficient, polynomial algorithm. On the other hand, a method that requires even $\frac{1}{100}2^{|\mathcal{N}|}$ steps in some cases is regarded as an inefficient, *exponential* algorithm.

As $|\mathcal{N}|$ and $|\mathcal{A}|$ grow larger, any exponential function of them grows much faster than any polynomial function. The following table[2] illustrates the trend; for $n = 10, \ldots, 60$, it compares the time needed to execute a polynomial number of operations (n, n^2, n^3 or n^5) and an exponential number of operations (2^n or 3^n) on the assumption that one operation takes a microsecond:

Time complexity function	Size n					
	10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

The sudden rapid growth of the exponential functions is evident. In more general terms, n^5 increases by a factor of 32 when n is doubled; but 2^n increases by the same factor when n merely increases by 5.

To see the same phenomenon another way, imagine that you had an algorithm that could solve some 1000-node network problem in one hour. If it were an $|\mathcal{N}|^5$ algorithm, the computer would need 32 hours to solve a 2000-node version of the problem. But if it were a $2^{|\mathcal{N}|}$ algorithm, the computer would need 32 hours just to solve a 1005-node version — and $2^{10} = 1024$ hours to

[2] M.R. Garey and D.S. Johnson, *Computers and Intractability,* p. 7.

solve a 1010-node version. (It would need $2^{1000} \approx 1.2 \times 10^{295}$ centuries to solve a 2000-node problem!)

There are many network problems (and others, too) for which polynomially bounded algorithms are known to exist. The "easy" problems in Section 2 are examples.

More generally, what problems are candidates for having polynomial algorithms? At least, they must have the property that, given a solution, you can check the objective value with only a polynomially bounded amount of work. Problems that satisfy this requirement are said to be ***nondeterministic polynomial,*** or ***NP.*** (Actually the definition of NP is not quite so simple, but the fine points need not concern us here.)

The subject of NP problems was introduced in a 1971 paper by Stephen Cook. Most importantly, he showed that there exists an ***NP-complete*** problem that is at least as hard as all other NP problems. More precisely, any other NP problem can be transformed to an equivalent problem of this one particular type, with only a polynomially bounded amount of work; so if you can solve problems of this one type, you can solve all NP problems with comparable effort.

Cook's original NP-complete problem was not especially relevant to operations research. Soon thereafter, however, Richard Karp and others were able to show that a huge number of more familiar and natural problems also fall into the NP-complete class. The traveling salesman problem described in Section 1 is NP-complete. The quite similar postman problem has an easy polynomial algorithm, if all arcs are directed or all are undirected; but it is NP-complete for networks that have some directed and some undirected arcs. This is a common pattern; small changes in a problem can make all the difference to whether it is NP-complete.

Here is a small sampling of other NP-complete network problems:

- Finding a spanning tree that minimizes the maximum distance between any two nodes, via the unique path through the tree.
- Finding a maximum-capacity cut through a network.
- Finding a longest Hamiltonian circuit or longest path; finding a Hamiltonian circuit whose longest arc is as short as possible.
- Finding a maximum flow that uses only certain specified paths.
- Finding the best integer solution to a multicommodity network flow linear program.

NP-complete problems also arise in other kinds of optimization, and in numerous other fields of mathematics and computer science.

Are there polynomially bounded algorithms for NP-complete problems? Probably not. Practical experience with various NP-complete problems suggests that they really are very hard to solve. Moreover, mathematicians have found reasons to suspect that an exponentially growing amount of work will be required by any algorithm that solves an NP-complete problem—although no proof of this fact has been devised as yet.

If all algorithms for NP-complete problems are "exponential", does this mean that they are impossible to solve? Not necessarily. First of all, for an algorithm to be exponential, it need only take an exponentially growing number of operations *in the worst case.* For example, an algorithm is considered exponential if, for every value of m, there exists at least one network of m nodes that takes 2^m iterations to solve. The same algorithm might, however, require only an average of $2m^3$ iterations to solve the networks that you are really interested in. Traveling salesman problems in hundreds of nodes have been efficiently solved, for instance, through a combination of ingenious techniques that seem to work well for a range of networks that arise in applications.

Furthermore, it is not always necessary to find an optimal solution to an NP-complete problem. Often a "close enough" solution will suffice. For some problems it is possible to devise efficient *heuristic* algorithms that come close to the optimum with acceptable consistency. In the case of the traveling salesman problem, for example, one obvious heuristic is to first pick a starting city, and then keep traveling to the closest unvisited city; when finally all cities have been visited, you return to the start. This particular idea often does not work too well, but more elaborate heuristics have been observed to do much better.

What about the problem of linear programming? There exist specially constructed linear programs in m constraints that can require 2^m simplex iterations when the entering variables are picked in seemingly logical ways, but typical linear programs require far fewer iterations. Thus the simplex algorithm can be viewed as an example of a worthwhile exponential algorithm. The existence of a polynomial algorithm for linear programming turns out to be a difficult question; it was resolved favorably only after the simplex method had been studied for three decades.

Almost another decade went by before algorithms that were both polynomial in theory and efficient in practice became widely available. These are the ones that were developed in Part III of these notes. They do not step from one basic solution to another, and are in most other respects unlike the simplex algorithm; they can be superior to the simplex algorithm, particularly for very large or degenerate problems.