Notes on Network Optimization
CE 4920
Stephen Boyles
Fall 2009

1 Introduction and Preliminaries

From this point on, the remaining course material takes on a more tangible form than nonlinear or linear programming by considering optimization problems with a special structure. One broad class of optimization problems takes place in *networks*. Intuitively, a network consists of a set of objects which are connected to each other in some fashion, as is very common in transportation: travel origins and destinations are joined by roadways; bus stops are joined by bus routes; factories are connected by supply chains; and so forth.

Mathematically, a network (or graph) is often written $G = (N, A)$ to indicate that it consists of two parts: a set of *nodes* N representing the basic objects, and a set of *arcs* A connecting them. We write $n = |N|$ to indicate the number of nodes, and $m = |A|$ to indicate the number of arcs. (This terminology is not universal; some authors call nodes vertices, and some call arcs links or edges.)

Figure 1 shows a network representation of part of the transportation infrastructure. Some of the nodes represent intersections, while others represent bus stops. The solid lines connecting the intersections are arcs representing roadways, while the dashed lines connecting the bus stops represents a transit route. Also note the thick line connecting one of the intersection nodes to a nearby bus stop node, representing a transfer from driving to the bus at the park-and-ride.

Usually the nodes are numbered from 1 to n, and the arcs are written representing the nodes they connect. For example, in Figure 2, the five arcs are written as $(1, 2)$, $(1, 3)$, $(2, 3)$, $(2, 4)$, and $(3, 4)$. The arrow indicates that these are *directed* arcs, meaning that the order is important: one may travel from node 1 to node 2, but not the other way around. By contrast, Figure 1 shows a graph where travel is allowed in both directions. Such arcs are called *undirected*, and indicated either by an arrow on both ends, or by no arrows at all (if no arrows appear, the default is to assume that travel in both directions is allowed).

The *forward star* $FS(i)$ of a node i is all of the arcs which leave that node. In Figure 2, $FS(1) = \{(1, 2), (1, 3)\}$, $FS(3) = \{(3, 4)\}$, and $FS(4) = \emptyset$. The reverse star $RS(i)$ is defined in a similar manner, and is the set of all arcs which enter that node. In the same example, $RS(1) = \emptyset$ and $RS(4) = \{(2, 4), (3, 4)\}$. For an undirected graph, the forward and reverse stars are identical.

In transportation applications, further data is usually associated with each arc and node. For instance, traveling on an arc (i, j) may cost a certain amount c_{ij} (which may include both monetary costs, as well as the cost of travel time). We may also be interested in the number of people using the arc x_{ij}, called the *flow* on an arc. The flow may also represent freight or other goods which travel on an arc. There may also be an upper limit on the amount of flow which can use an arc; this is the capacity u_{ij}. Finally, flow may be "created" or "absorbed" at nodes (for instance,

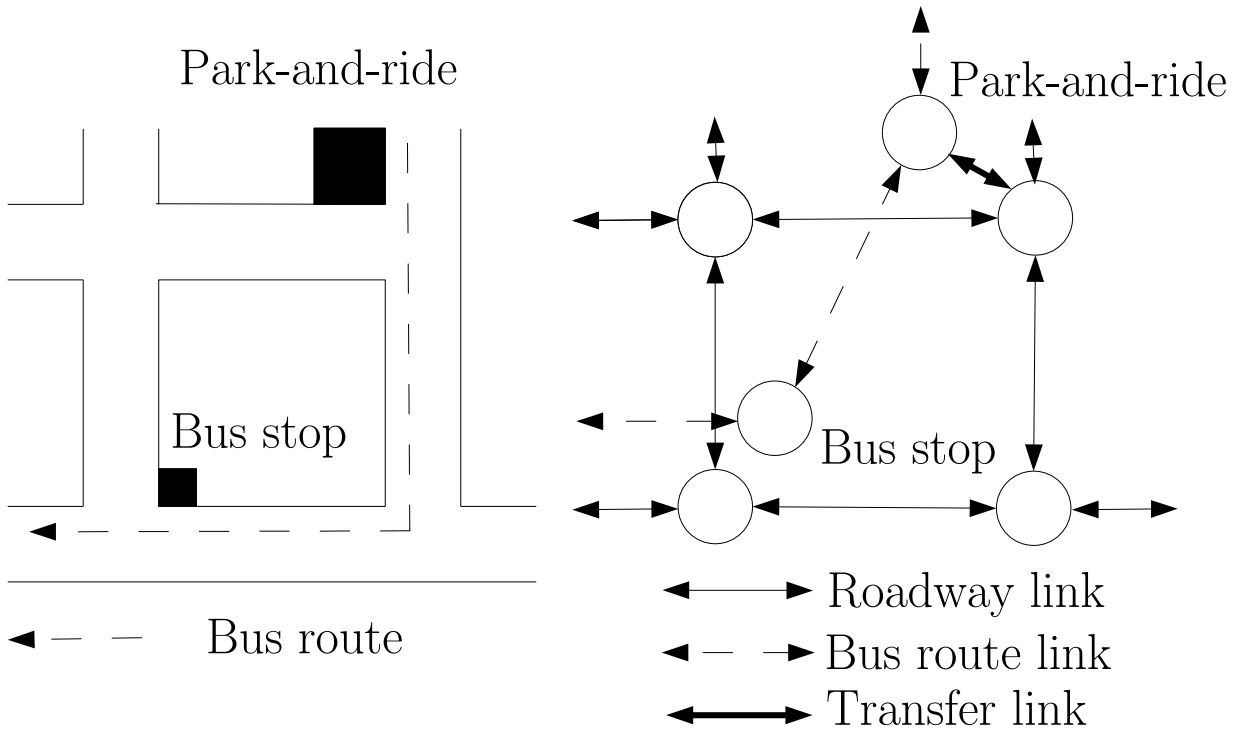


Figure 1: Example of a network representing road and bus lines.

goods are created and factories and absorbed at warehouses; commuting trips are created at nodes representing homes, and absorbed at trips representing workplaces); this is written by d_i, with the understanding that flow is created at node i if $d_i > 0$, and absorbed if $d_i < 0$.

Some other terminology is useful. A *path* is a collection of adjacent links; in Figure 2, $\{(1,2),(2,3),(3,4)\}$ is a path representing travel from node 1 to node 2 to node 3 to node 4. $\{(1,2),(2,4)\}$ is a path representing travel from node 1 to node 2, then directly to node 4. A *cycle* is a path which starts and ends at the same nodes. A *tree* is a special type of graph in which no cycles exist (even if we change all the arcs to be undirected). A graph is *connected* if a path exists between every pair of nodes (again ignoring the direction on the arcs), and *strongly connected* if such a path exists taking arc direction into account. A tree is called a *spanning tree* if it is connected.

2 Minimum Spanning Trees

The *minimum spanning tree problem* is a basic problem in network optimization. Technically, it can be stated as follows: *given a graph $G(N,A)$ where each arc has a cost c_{ij}, find a spanning tree $T(N,A')$ where $A' \subseteq A$ and the sum of the costs of the arcs in A' is minimal.*

One application of minimum spanning trees is construction of a road network in a rural area,

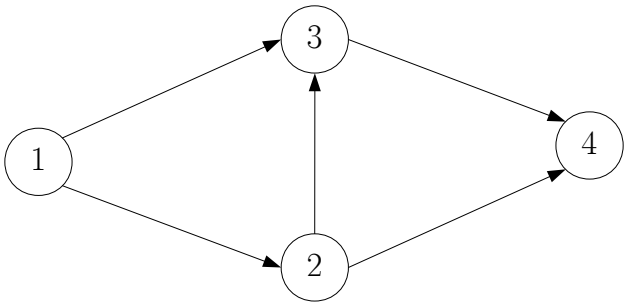


Figure 2: Labeling of nodes and arcs.

where c_{ij} represents the cost of constructing a road between towns i and j. In this network, the minimum spanning tree would show the cheapest way to build roadways ensuring that every town is connected to every other. Another application is in deployment of utilities, such as power lines or water mains: the minimum spanning tree shows the cheapest way to provide service to the public.

How can we find a minimum spanning tree? First, it should be clear that every spanning tree contains exactly $n-1$ arcs. If it contained fewer than this, the graph would not be connected and some nodes would go unserved. If it contained more than this, the graph would contain a cycle.[1] Likewise, every collection of $n-1$ arcs which has no cycle is a spanning tree.

So, the question is how to choose $n-1$ arcs in such a way that no cycle is formed. It turns out that the minimum spanning tree can be found through a *greedy algorithm*, a special solution technique where we can build the solution one step at a time, only considering what's best at the present step. (That is, we don't have to think ahead and worry about the impact of our current choice on the future.) Two simple, greedy algorithms are commonly used: Prim's algorithm, and Kruskal's algorithm.

Prim's algorithm builds the tree one arc at a time, starting from a "root" node s, and works as follows:

1. Create the set $N_T = \{s\}$ to represent the nodes in the tree.
2. Consider each arc (i, j) that has the property that i is in the tree, but j is not. Of these, pick the arc (u, v) with the lowest cost.
3. Add (u, v) to the set of tree arcs A', and add v to the set of tree nodes N_T.
4. Repeat until $N_T = N$, that is, until all nodes are in the tree

Prim's algorithm is greedy because at each point we are picking the cheapest arc to add to the tree. We never have to worry about creating a cycle in the tree, because the new arc never connects two nodes which are already in the tree. Thus, Prim's algorithim requires exactly $n-1$ steps to find the minimum spanning tree. We can write this more formally, using the pseudocode of Algorithm 1,

[1] Formal proof of this can be found in Bertsimas & Tsitsiklis, and any introductory text on graph theory

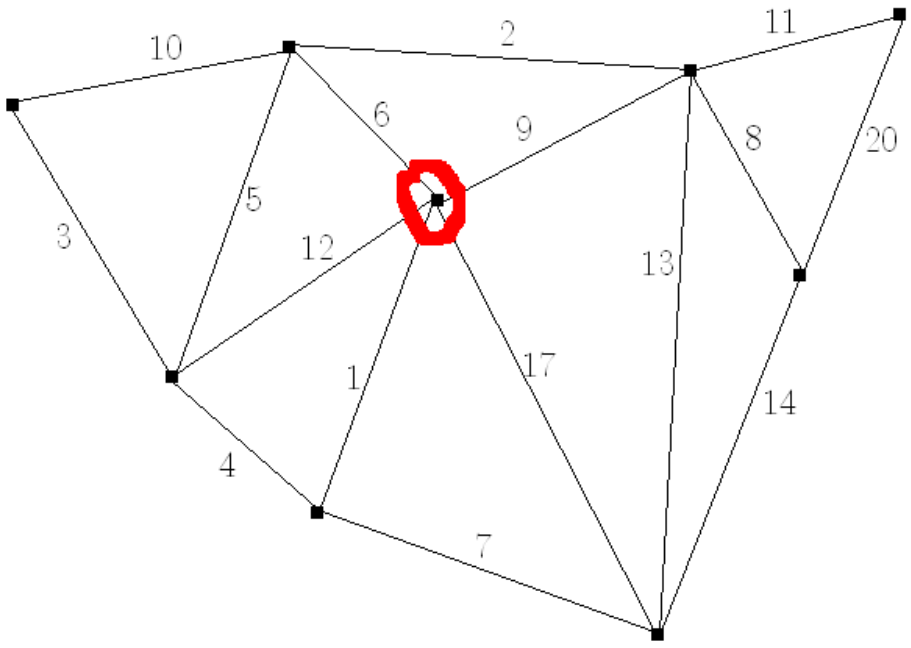


Figure 3: Network for demonstrating Prim's algorithm.

where E defines the set of "eligible" arcs which can enter while maintaining a connected, cycle-free tree.

For an example, consider the network in Figure 3, where the arc costs are as indicated and the root note is circled in red (this is arbitrary; we could have chosen any other node to be the root as well). In the first step, we consider all of the arcs which leave the root node. The arc with a cost of 1 is cheapest, so we add it to the tree (Figure 4). The set N_T now consists of the two nodes circled in red. We now examine all of the arcs which connect a node in N_T to a node outside of N_T; the one with a cost of 4 is cheapest, so we add that to the tree (Figure 5). We continue in the same manner (Figures 6–11) until every node is connected. This spanning tree has a cost of $1 + 4 + 3 + 5 + 2 + 7 + 8 + 11 = 41$, which is the least among all possible spanning trees.

How do we know that Prim's algorithm works? Since it is supposed to find a minimum spanning tree, there are three ways it can go wrong: (1) it fails to find a tree; (2) it fails to find a *spanning* tree; (3) it fails to find a *minimum* spanning tree. We can check each of these conditions in turn. [2]

Clearly it finds a tree, because the only way a cycle can be formed is if an arc is added, where both its tail and head nodes are already in the tree. However, the definition of the eligible arc set E ensures that no such arcs will ever be added. Furthermore, it is clear that this tree is spanning because the algorithm only terminates when every node is included in the tree. So, the only way Prim's algorithm could go wrong is if the spanning tree it finds does not have minimum cost.

[2]The discussion provided here is not entirely rigorous. A more formal proof can be found at `http://math.wikia.com/wiki/Proof\_of\_Prim\%27s\_algorithm`

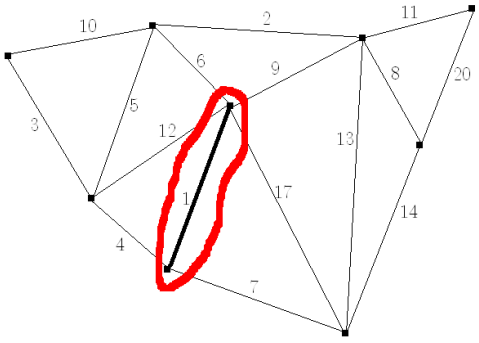


Figure 4: Prim's algorithm, step 1.

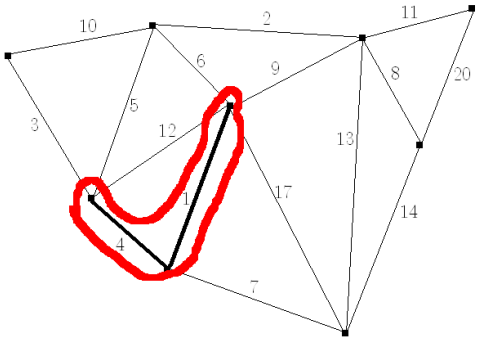


Figure 5: Prim's algorithm, step 2.

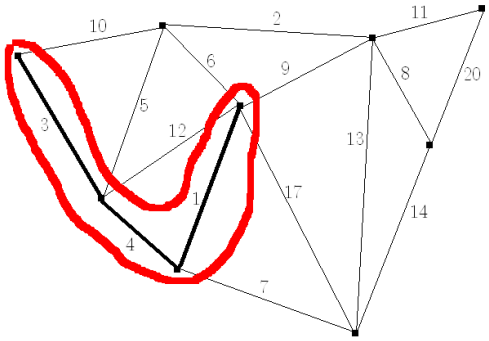


Figure 6: Prim's algorithm, step 3.

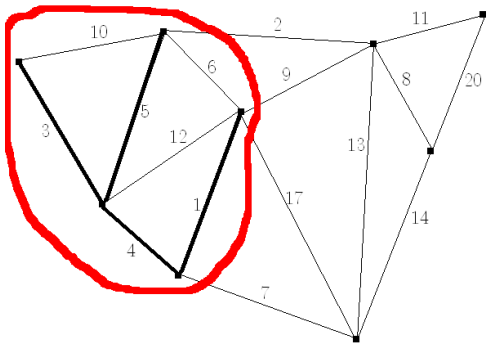


Figure 7: Prim's algorithm, step 4.

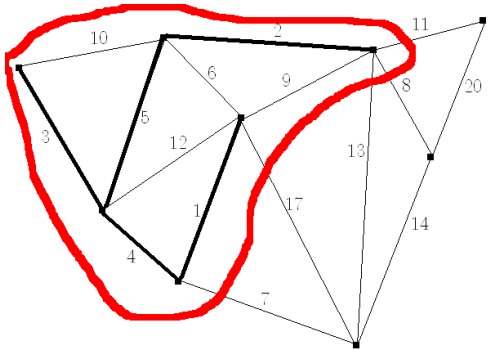


Figure 8: Prim's algorithm, step 5.

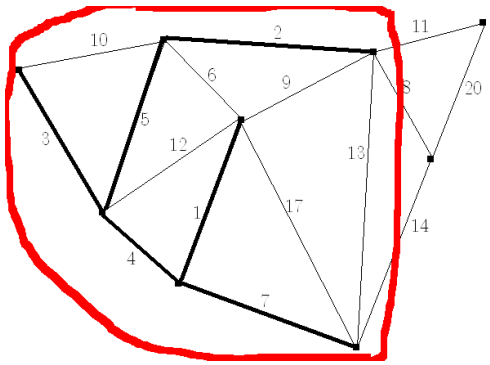


Figure 9: Prim's algorithm, step 6.

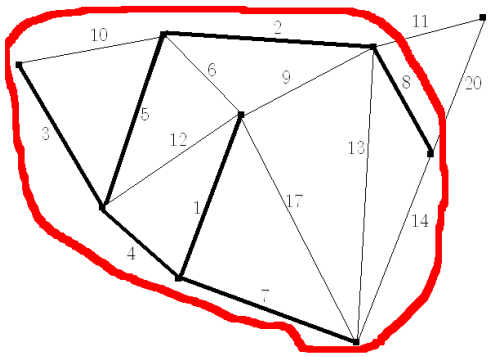


Figure 10: Prim's algorithm, step 7.

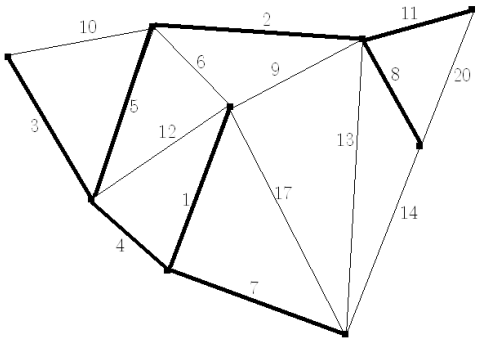


Figure 11: Prim's algorithm, final step.

Algorithm 1 Prim's algorithm

{Input: a network $G = (N, A)$}
{Output: a minimum spanning tree $T = (N^T, A^T)$}
{Initialization}
$N^T \leftarrow i$ for some $i \in N$
$A^T \leftarrow \emptyset$
while $N^T \neq N$ **do**
    $E \leftarrow \{(i, j) \in A : i \in N^T, j \notin N^T\}$
    Choose $(i^*, j^*) \in \arg\min_{(i,j)\in E}\{c_{ij}\}$
    $N^T \leftarrow N^T \cup \{j^*\}$
    $A^T \leftarrow A^T \cup \{(i^*, j^*)\}$
end while
return N^T, A^T

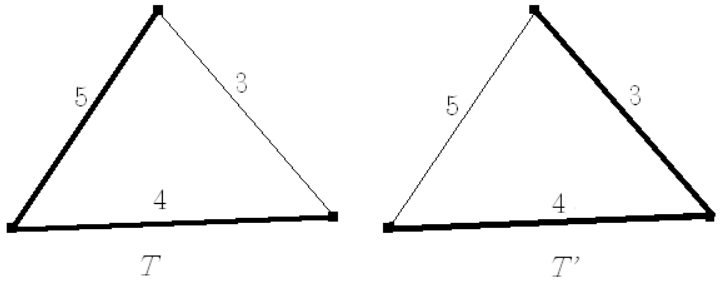


Figure 12: Why Prim's algorithm always works.

By contradiction, assume that this is the case. Then there must be some other tree T' with a lower cost than T. Thus, there is some arc (i, j) which is in T but not T', and some arc (i', j') which is in T' but not T. Consider the graph $T = (N, A^T \cap \{(i', j')\})$. This graph contains a cycle, in which some arc $(\hat{i}, \hat{j})$ (not (i', j')) has maximum cost. (see Figure 12). Thus, at some point, Prim's algorithm would have chosen to have (i', j') enter the tree rather than $(\hat{i}, \hat{j})$, which is the contradiction we needed.

3 Shortest Path Algorithms

3.1 Preliminaries

The *shortest path problem* is another fundamental problem in network optimization. This problem can be stated as follows: *given a graph $G = (N, A)$ where each arc has a cost c_{ij}, an origin $r \in N$, and a destination $s \in N$, find the path P connecting r and s for which the sum of its arcs' costs is minimal.*

The shortest path problem is very widely applicable. In transportation and logistics problems, it can be used to find either the quickest or cheapest paths connecting two locations (if the costs represent travel time and monetary cost, respectively). A huge variety of other applications also exists, in construction management, geometric design, operations research, and many other areas. For instance, the fastest way to solve a Rubik's Cube from a given position can be solved using a shortest path algorithm, as can the fewest number of links needed to connect an actor to Kevin Bacon when playing Six Degrees of Separation. *Network Flows* by Ahuja, Magnanti, and Orlin provides a thorough overview of such applications.

For an example, consider the graph in Figure 2, with arc costs given in Table 1. Let's say we are trying to find the shortest path between node 1 (r) and node 4 (s). Three paths connecting these

Table 1: Costs for demonstration of shortest paths.

Arc	Cost
(1,2)	2
(1,3)	4
(2,3)	1
(2,4)	5
(3,4)	2

nodes exist in this network: $\{(1,2),(2,4)\}$, which has a cost of $2+5=7$; $\{(1,3),(3,4)\}$, which has a cost of $4+2=6$; and $\{(1,2),(2,3),(3,4)\}$, which has a cost of $2+1+2=5$. Thus, we see that the shortest path is $\{(1,2),(2,3),(3,4)\}$, even though it contains three arcs rather than two.

Most shortest path algorithms rely on *Bellman's principle*, which requires that any segment of a shortest path itself be a shortest path. For instance, the shortest path in this example was $\{(1,2),(2,3),(3,4)\}$. Bellman's principle requires that $\{(1,2),(2,3)\}$ be a shortest path from nodes 1 to 3, and that $\{(2,3),(3,4)\}$ be a shortest path from nodes 2 to 4. It further requires that $\{(1,2)\}$ be a shortest path from node 1 to node 2, $\{(2,3)\}$ be a shortest path from node 2 to node 3, and $\{(3,4)\}$ be a shortest path from node 3 to node 4. You can easily verify that this is true with the given arc costs.

To see why this must be the case, assume that Bellman's principle was violated. Continuing with the same example, if the cost on arc $(1,3)$ was reduced to 2, then $\{(1,2),(2,3)\}$ is no longer a shortest path from node 1 to node 3 (that path has a cost of 3, while the single-arc path $\{(1,3)\}$ has cost 2). Bellman's principle implies that $\{(1,2),(2,3),(3,4)\}$ is no longer the shortest path between nodes 1 and 4. Why? The first part of the path can be replaced by $(1,3)$ (the new shortest path between 1 and 3), reducing the cost of the path from 1 to 4: $\{(1,3),(3,4)\}$ now has a cost of 4.

In general, if a segment of a path does *not* form the shortest path between two nodes, we can replace it with the shortest path, and thus reduce the cost of the entire path. Thus, the shortest path must satisfy Bellman's principle for all of its segments. Bellman's principle also implies that the union of shortest paths from r to every other node can form a tree; this provides a compact representation which is very useful when finding shortest paths.

The two most common approaches for solving shortest paths are *label setting* and *label correcting*, both of which are described in the following subsections. Dijkstra's algorithm and the Bellman-Ford algorithm are the quintessential label setting and label correcting algorithims, respectively, and they are used to show the difference between these approaches.

Both of these algorithms actually find the shortest path from the origin r to *every* other node, not just the destination s; this exploits Bellman's principle, because the shortest path from r to s must also contain the shortest path from r to every node in that path. Further, both of them contain a set of labels L_i associated with each node, representing the shortest path from r to node i. With a label setting approach, each node's label is determined once and only once. On the other hand, with a label correcting approach, each node's label can be updated multiple times. An analogy can

be drawn with line minimization and the Armijo rule: the former usually requires fewer iterations, but each iteration requires more time. The latter may take more iterations, but less time is spent on each step.

3.2 Label Setting: Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from the origin r to every other node, *when every arc has a nonnegative cost*. Furthermore, at each step it finds the shortest path to at least one additional node. It uses the concept of *finalized* nodes, that is, nodes to which the shortest path has already been found. Furthermore, it stores an $n \times 1$ vector $\mathbf{q}$, representing the previous node in the shortest paths, taking advantage of the treelike sructrure of the shortest paths.

The vector $\mathbf{q}$ allows the shortest paths to be traced back to the origin; we use a -1 to indicates that there is no predecessor in the shortest path tree, either because that node is the origin, or because we haven't found a shortest path yet. For instance, let's say that $\mathbf{q} = \begin{bmatrix} -1 & 3 & 1 & 3 & 4 \end{bmatrix}$ and we are finding the shortest paths from node 1. Node 1 is the origin, so $q_1 = -1$. To find the shortest path to node 2, we see that $q_2 = 3$, so we reach node 2 through node 3. $q_3 = 1$, so we reach node 3 through node 1. This is the origin; so the shortest path to node 2 is thus $(1, 3, 2)$. To find the shortest path to node 5, we again trace the labels $\mathbf{q}$: $q_5 = 4$; $q_4 = 3$; $q_3 = 1$, so the shortest path is $(1, 3, 4, 5)$.

With this in mind, Dijkstra's algorithm can be stated as follows:

1. Initialize every label L_i to ∞, except for the origin, where $L_r \leftarrow 0$.
2. Initialize the set of finalized nodes $F = \{r\}$, and the path vector $\mathbf{q} \leftarrow \mathbf{-1}$.
3. Find the set of eligible arcs $E = \{(i, j) \in A : i \in F, j \notin F\}$
4. For each arc in E, calculate the temporary labels $L_{ij}^{temp} = L_i + c_{ij}$
5. Find the arc (i^*, j^*) for which L_{ij}^{temp} is minimal.
6. Update $L_j = L_{ij}^{temp}$, add j^* to F, and set $q_{j^*} = i^*$.
7. If all nodes have been finalized ($F = N$), terminate. Otherwise, return to step 3.

This algorithm is considered "label setting," because once a node's label is updated, the node is finalized and never revisited again. We consider how this algorithm can be applied to the example in Figure 2 and Table 1.

Initialization. We set $\mathbf{L} = \begin{bmatrix} 0 & \infty & \infty & \infty \end{bmatrix}$, $F = \{1\}$, and $\mathbf{q} = \begin{bmatrix} -1 & -1 & -1 & -1 \end{bmatrix}$.

Iteration 1. The set of eligible arcs is $\{(1, 2), (1, 3)\}$, from which we calculate the temporary labels $L_{12}^{temp} = 0 + 2 = 2$ and $L_{13}^{temp} = 0 + 4 = 4$. The temporary label is smallest for $(1, 2)$, so

we update $\mathbf{L} = \begin{bmatrix} 0 & 2 & \infty & \infty \end{bmatrix}$, $F = \{1, 2\}$, and $\mathbf{q} = \begin{bmatrix} -1 & 1 & -1 & -1 \end{bmatrix}$. We have found the shortest path to node 2, which has a cost of 2, and consists of the arc $\{(1, 2)\}$ as seen from the vector $\mathbf{q}$.

Iteration 2. The set of eligible arcs is $\{(1, 3), (2, 3), (2, 4)\}$, from which we calculate temporary labels $L_{13}^{temp} = 4$, $L_{23}^{temp} = 2 + 1 = 3$, and $L_{24}^{temp} = 2 + 5 = 7$. The smallest label corresponds to $(2, 3)$, so we update $\mathbf{L} = \begin{bmatrix} 0 & 2 & 3 & \infty \end{bmatrix}$, $F = \{1, 2, 3\}$, and $\mathbf{q} = \begin{bmatrix} -1 & 1 & 2 & -1 \end{bmatrix}$. We have found the shortest path to node 3, which has a cost of 3. To find the actual path itself, we trace the nodes through $\mathbf{q}$. $q_3 = 2$, and $q_2 = 1$, so the shortest path travels from node 1, to node 2, to node 3.

Iteration 3. The set of eligible arcs is $\{(2, 4), (3, 4)\}$, from which we calculate temporary labels $L_{24}^{temp} = 7$ and $L_{34}^{temp} = 5$. The smallest corresponds to $(3, 4)$, so $\mathbf{L} = \begin{bmatrix} 0 & 2 & 3 & 5 \end{bmatrix}$, $F = \{1, 2, 3, 4\}$, and $\mathbf{q} = \begin{bmatrix} -1 & 1 & 2 & 3 \end{bmatrix}$. $F = N$, so the algorithm terminates.

Why does Dijkstra's algorithm work? Put another way, is there any way it can fail? At each iteration, it fixes the shortest path from the origin to one more node; let's see if it's possible that the algorithm is wrong in doing so. Let's say that at iteration k, the temporary label L_{ij}^{temp} is least, so j is added to the set of finalized nodes, and the algorithm says that the shortest path from r to j consists of the shortest path from r to i, plus the arc (i, j).

If this was wrong, there is some faster way to get from r to j, say, from node r to node h, plus the arc (h, j). If node h were finalized, then we already would have considered this option by calculating the temporary label L_{hj}^{temp}, which would have been smaller and thus this path would have been chosen, rather than the one through i. Thus, we conclude that h has not yet been finalized. This means that the cost of the shortest path from r to h is at least as large as L_{ij}^{temp}. Since the cost of each arc is nonnegative, this means that the path from r to i through node h must have a cost at least as large as L_{ij}^{temp}, contradicting the assumption that this path was shorter than going through i.

3.3 Label Correcting: Bellman-Ford Algorithm

As shown in the previous section, Dijkstra's algorithm finds the shortest path from the origin to a new node at each iteration. Two potential problems with this approach are (1) it only works when arc costs are nonnegative, and (2) finding the minimum temporary label can be time-consuming. For this reason, an alternate class of shortest path algorithms have been developed, in which the node labels can change multiple times. This approach typically requires more iterations to converge, but each iteration is faster.

The Bellman-Ford algorithm maintains a *scan eligible list* SEL, containing nodes which need to be "scanned" by considering new shortest paths that pass through these nodes.

1. Initialize every label L_i to ∞, except for the origin, where $L_r \leftarrow 0$.

2. Initialize the scan eligible list $SEL = \{r\}$, and the path vector $\mathbf{q} \leftarrow \mathbf{-1}$.

3. Choose a node $i \in SEL$ and remove it from that list.

4. Scan node i as follows: for each arc $(i, j) \in A$, compare L_j to $L_i + c_{ij}$. If $L_i + c_{ij} < L_j$, update $L_j = L_i + c_{ij}$, $q_j = i$, and add j to SEL.

5. If SEL is empty, then terminate. Otherwise, return to step 3.

Repeating the same example, this algorithm works as follows:

Initialization. We set $\mathbf{L} = \begin{bmatrix} 0 & \infty & \infty & \infty \end{bmatrix}$, $SEL = \{1\}$, and $\mathbf{q} = \begin{bmatrix} -1 & -1 & -1 & -1 \end{bmatrix}$.

Iteration 1. We remove node 1 from SEL to scan. We consider the two arcs $(1, 2)$ and $(1, 3)$ emanating from node 1. We see that $L_1 + c_{12} = 2 < \infty = L_2$ and $L_1 + c_{13} = 4 < \infty = L_3$, so we update the values of $\mathbf{L}$ and $\mathbf{q}$ for these nodes and add them to SEL. Thus $\mathbf{L} = \begin{bmatrix} 0 & 2 & 4 & \infty \end{bmatrix}$, $\mathbf{q} = \begin{bmatrix} -1 & 1 & 1 & -1 \end{bmatrix}$, and $SEL = \{2, 3\}$. Note that, unlike Dijkstra's algorithm, we are not claiming that these are the shortest paths to these nodes. We are simply claiming that these are the shortest paths found thus far.

Iteration 2. Remove node 3 from SEL. The only arc emanating from node 3 is $(3, 4)$, and $L_3 + c_{34} = 6 < \infty = L_4$, so we update $\mathbf{L}$, $\mathbf{q}$, and SEL: $\mathbf{L} = \begin{bmatrix} 0 & 2 & 4 & 6 \end{bmatrix}$, $\mathbf{q} = \begin{bmatrix} -1 & 1 & 1 & 3 \end{bmatrix}$, and $SEL = \{2, 4\}$. Even though we've found a path to the destination (node 4), we don't know if it's shortest until we continue scanning every element in SEL.

Iteration 3. Remove node 2 from SEL. There are two arcs adjacent to this node. For $(2, 3)$, $L_2 + c_{23} = 3 < 4 = L_3$, so we update $L_3 = 3$, $q_3 = 2$, and add node 3 into SEL a second time. For $(2, 4)$, $L_2 + c_{24} = 7 > 6 = L_4$, so we do not perform any update step for this arc. Thus $\mathbf{L} = \begin{bmatrix} 0 & 2 & 3 & 6 \end{bmatrix}$, $\mathbf{q} = \begin{bmatrix} -1 & 1 & 2 & 3 \end{bmatrix}$, and $SEL = \{3, 4\}$

Iteration 4. Remove node 3 from SEL for the second time. Checking arc $(3, 4)$ we see that $L_3 + c_{34} = 5 < 6 = L_4$ (we are "correcting" the labels found the first time around, because we have found a shorter way to get to node 3) and $\mathbf{L} = \begin{bmatrix} 0 & 2 & 3 & 5 \end{bmatrix}$, $\mathbf{q} = \begin{bmatrix} -1 & 1 & 2 & 3 \end{bmatrix}$, and $SEL = \{4\}$.

Iteration 5. Remove node 4 from SEL. No arcs leave node 4, so there is no scanning to be done. At this point SEL is empty, and the algorithm terminates.

4 Maximum Flow Problem

The shortest path problem made use of arc costs, but ignored capacity. The maximum flow problem, on the other hand, makes use of arc capacities, but ignores arc costs. We are given an origin r and a destination s as before, and wish to ship as much flow as possible from r to s while obeying arc capacities u_{ij}. This has applications in evacuation (trying to get as many people to safety, given capacity limitations on expressways and other routes to safety), scheduling, and finding the number

of redundant paths through a network. (Again, *Network Flows* by Ahuja et al. does an excellent job of providing surprising applications for these problems. Best textbook ever.)

We can formulate this as a formal optimization problem. Let the decision variable b represent the amount of flow shipped from r to s, and let x_{ij} be the amount of flow on a given link. Clearly, the objective function is to maximize b, and as a constraint we must have $0 \leq x_{ij} \leq u_{ij}$. We also have to make sure that flow conservation is enforced, that is, flow doesn't appear or disappear at any nodes except the origin and the destination. Following the AMPL tutorial for the shortest path problem, we can write this as $\sum_{(h,i)\in A} x_{hi} - \sum_{(i,j)\in A} x_{ij} = 0$ for all nodes i not the origin or destination; $\sum_{(h,r)\in A} x_{hr} - \sum_{(r,j)\in A} x_{rj} = -b$ for the origin r, and $\sum_{(h,s)\in A} x_{hs} - \sum_{(s,j)\in A} x_{sj} = 0$ for the destination s. That is, the maximum flow problem solves

$$\begin{aligned} \max_{b,\mathbf{x}} \quad & b & (MAXFLOW) \\ \text{s.t.} \quad & \sum_{(h,i)\in A} x_{hi} - \sum_{(i,j)\in A} x_{ij} = \begin{cases} -b & i = r \\ +b & i = s \\ 0 & \text{otherwise} \end{cases} \\ & 0 \leq x_{ij} \leq u_{ij} & \forall (i,j) \in A \end{aligned}$$

This is a linear program which we could solve using the simplex method. However, faster approaches are available that specifically take advantage of the network structure. One of the simplest to understand is the *augmenting path algorithm*. This makes use of something called the *residual network*, which is defined for a given flow. For flows $\mathbf{x}$, the residual network $R(\mathbf{x})$ consists of

1. The same set of nodes as the original network.
2. *Two* arcs for every arc (i,j) in the original network: a *forward arc* (i,j), with capacity $u_{ij} - xij$, and a *reverse arc* (j,i) with capacity x_{ij}.

For examples of residual networks, see Figures 13–17.

The augmenting path algorithm works in the following fashion:

1. Set the initial flows $\mathbf{x^0} = \mathbf{0}$, $b^k = 0$ and the iteration counter $k = 0$.
2. Construct the residual network $R(\mathbf{x})$ corresponding to flows $\mathbf{x^k}$.
3. Identify a path π^k in R connecting s and t which has positive capacity on each of its arcs. If no such path exists, terminate: $\mathbf{x^k}$ and b^k are optimal.
4. Find the minimum capacity u_k of all of the arcs in π^k (whether forward or reverse).
5. For each forward arc (i,j) in π^k, $x_{ij}^{k+1} = x_{ij}^k + u_k$. For each reverse arc (j,i) in π^k, $x_{ij}^{k+1} = x_{ij}^k - u_k$. For all other arcs, $x_{ij}^{k+1} = x_{ij}^k$.
6. Increase b^k by u_k, increase the iteration counter k by one, and return to step 2.

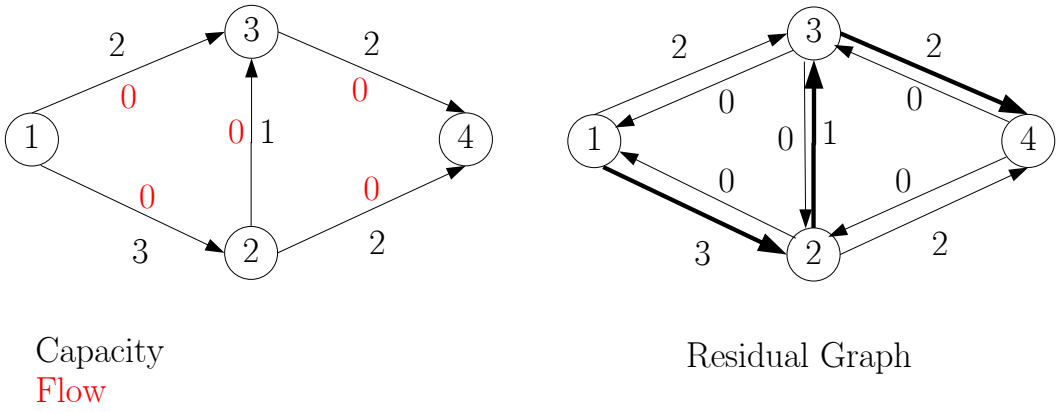


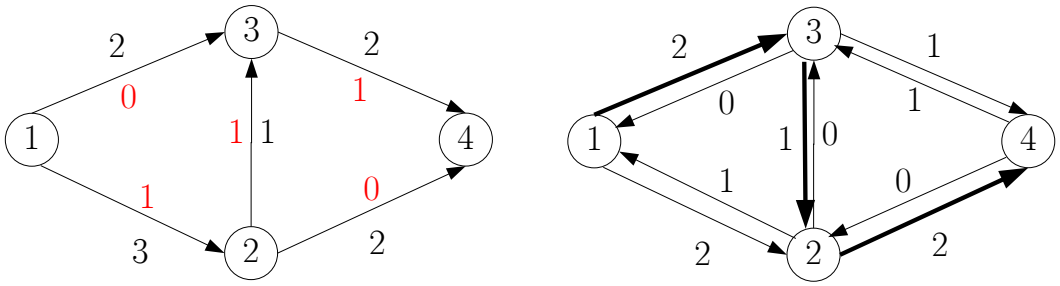
Figure 13: Augmenting path algorithm, initialization.

For instance, consider the network in Figure 13, with the initial (zero) flow shown at left and the corresponding residual network to the right. Let the origin r be node 1, and the destination s be node 4. Currently no flow is traveling from the origin to the destination ($b = 0$) and $x_{ij} = 0$ for all arcs. In the residual network, we find that the path $1 - 2 - 3 - 4$ connects the origin and destination, and has positive capacity on each arc. The capacity of this "augmenting path" is the minimum capacity of its component arcs: $\min\{3, 1, 2\} = 1$. Each component arc is a forward arc, so we increase the flow on $(1, 2)$, $(2, 3)$, and $(3, 4)$ by one unit and update the residual network (Figure 14).

Note that the arc $(2, 3)$ no longer has any forward capacity in the residual network. This was also the arc which determined the capacity of the augmenting path. As you may have guessed, this is no coincidence. We added the full capacity to this arc, so there is no more room for adding flow. This is represented by setting the residual forward capacity to zero. However, another augmenting path exists in the residual network, the path $1 - 3 - 2 - 4$. The capacity of this augmenting path is $\min\{2, 1, 2\} = 1$; arcs $(1, 3)$ and $(2, 4)$ are forward arcs so we increase their flow by one. However, arc $(3, 2)$ is a *reverse* arc, so we *subtract* one from the flow on $(2, 3)$. (Essentially, we are partially "undoing" what we did last time, and rearranging the flow so there is a net increase.)

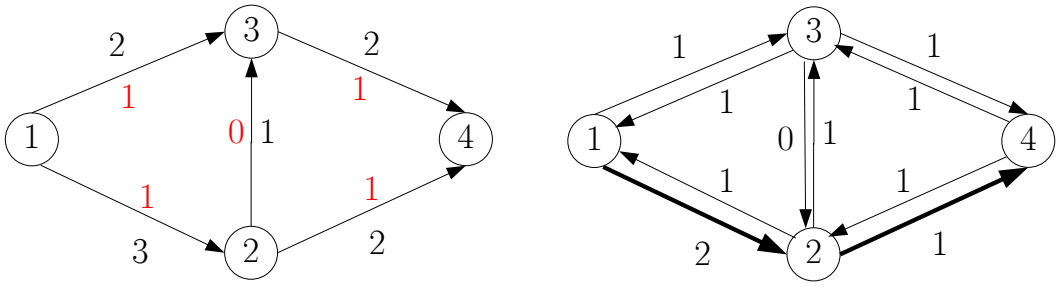
The new flows and residual network are shown in Figure 15; note that in this new configuration we are shipping two units from node 1 to node 4. Continuing, we find the augmenting path $1 - 2 - 4$ with capacity $\min\{2, 1\} = 1$, so we increase the flow on the forward arcs $(1, 2)$ and $(2, 4)$ by one, obtaining Figure 16. Again, we find an augmenting path $1 - 3 - 4$ with capacity $\min\{1, 1\}$ and increase the flow on the forward arcs $(1, 3)$ and $(3, 4)$ by one, increasing b to 4 and obtaining Figure 17. At this point, no more augmenting paths exist in the residual network, so we stop with the optimal solution.

In this example, we knew we were done when we couldn't find a path with positive capacity



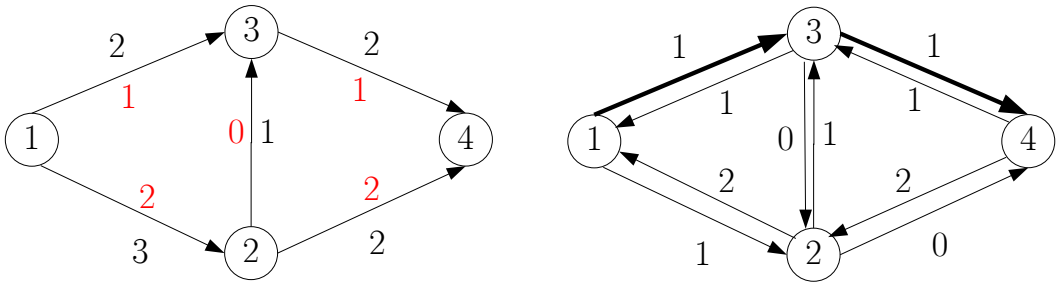
$b = 1$

Figure 14: Augmenting path algorithm, iteration one.



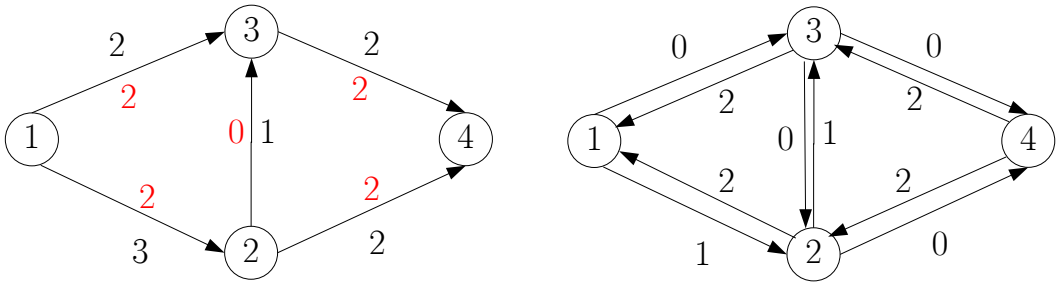
$b = 2$

Figure 15: Augmenting path algorithm, iteration two.



$b = 3$

Figure 16: Augmenting path algorithm, iteration three.



$b = 4$

Figure 17: Augmenting path algorithm, iteration four.

connecting the origin and the destination in the residual network. Note that there are several paths, but all of them were "blocked" by an arc with zero residual capacity. These "blocking" arcs can be important from a practical perspective, if they reflect bottlenecks that need to be improved if the maximum flow is to increase. Clearly, improving a non-blocking arc would be pointless; what may not be so clear is that improving the capacity on a blocking arc does not always improve the maximum flow. A more precise definition of "blocking" is needed, which is the next topic we turn to.

The concept of a *cut* makes this idea rigorous. Define a *cut* to be a partition[3] of the set of nodes N into two sets R and S, where R contains the origin (and possibly some other nodes), and S contains the destination (and possibly some other nodes). A *cut arc* is an arc whose tail node is in R, and whose head node is in S; that is, the cut arcs represent travel from R to S.

For example, in Figure 13, one cut might be $R = \{1, 3\}$, $S = \{2, 4\}$. (This is a partition because every node is assigned to either R or S, but not both.) Then the cut arcs are $(1, 2)$ and $(3, 4)$ beacuse these arcs start in R and end in S. $(1, 3)$ is not a cut arc because both of its end nodes are in R, $(2, 4)$ is not a cut arc because both of its end nodes are in S, and $(2, 3)$ is not a cut arc because it goes from S to R, not the other way around. On the other hand, if the cut was $R = \{1, 2\}$, $S = \{3, 4\}$, then the cut arcs are $(1, 3)$, $(2, 3)$, and $(2, 4)$. If the cut was $R = \{1, 2, 3\}$, $S = \{4\}$, then the cut arcs are $(2, 4)$ and $(3, 4)$.

The *capacity of a cut* is the sum of the capacities in the cut arcs. So for the cut $R = \{1, 3\}$, $S = \{2, 4\}$, its capacity is $2+3 = 5$. For the cut $R = \{1, 2\}$, $S = \{3, 4\}$, the capacity is $2+1+2 = 5$. For the cut $R = \{1, 2, 3\}$, $S = \{4\}$, the capacity is $2 + 2 = 4$. If we write the capacity of a cut as $u(R, S)$, then we have the following result:

Theorem 1. (Weak max-flow/min-cut theorem). *If $(b, \mathbf{x})$ is a feasible solution to (MAXFLOW) (that is, it ships b units of flow from r to s), and if $\{R, S\}$ is a partition of N, we must have $b \leq u(R, S)$.*

This theorem says that the amount of flow shipped from r to s in *any* feasible solution (not necessarily optimal) can never exceed the capacity of *any* cut. The proof of this theorem is fairly obvious: any flow shipped from r to s starts in the set R and ends in the set S, so it must cross a cut arc. Since the flow is feasible, the capacity of the cut arcs cannot be exceeded; and thus the amount of flow moving from r to s can never be greater than the sum of the capacity of the cut arcs. This is not surprising; what may be more surprising is that this holds *for any cut whatsoever*.

Even more surprising, we can find some flow $\mathbf{x}$ and some cut $\{R, S\}$ such that the amount being shipped from r to s is *exactly* equal to the capacity of that cut. This can be expressed formally as:

Theorem 2. (Strong max-flow/min-cut theorem). *If $(b^*, \mathbf{x}^*)$ is an optimal solution to (MAXFLOW), and if $u^* = \min\{u(R, S)\}$ among all possible cuts $\{R, S\}$, then $b^* = u^*$.*

That is, the maximum flow that can be shipped from r to s is exactly equal to the smallest possible cut separating R from S. To see why this is true, consider the augmenting path algorithm. If we

[3] A partition divides a set into smaller sets; each element of the original set is assigned to one and only one of the smaller sets.

have found the solution to the maximum flow problem, then there are no more paths connecting r to s which have positive capacity remaining. Let R be the set of nodes for which paths *do* exist from r with remaining capacity; and let S be all other nodes. Because the algorithm has stopped, $s \in S$. Clearly every node is either in R or S, so this forms a valid cut. Also, by the definition of R and S, every arc connecting R to S has no remaining capacity. Thus, the capacity of the cut R, S is exactly equal to the sum of the flow on the cut arcs, so $b = u(R, S)$. From the weak max-flow/min-cut theorem, this immediately tells us that b is maximal, and $u(R, S)$ is minimal.

Note that this serves as a proof of correctness for the augmenting path algorithm as well! This result says that we can find the value of the maximum flow by finding the minimum cut in the network. For example, from Figure 13, we saw that the cut $R = \{1, 2, 3\}$, $S = \{4\}$ had capacicty four, which is exactly equal to the flow b found by the augmenting path algorithm. Thus $b = 4$ is maximal, and the capacity of this cut is minimal.

5 Minimum Cost Flow Problem

The shortest path problem considered arc costs, but ignored capacities; the maximum flow problem considered capacities, but ignored arc costs. The minimum cost flow problem (or "min cost" for short) can be seen as a generalization of these: now *both* costs and capacities come into play. Furthermore, there can be more than one origin and destination. In fact, it turns out that the min cost flow problem is a formal generalization of shortest path and maximum flow as well, that is, if we have an algorithm for solving min cost flow, we can also use it to solve shortest path and maximum flow.

In the min cost flow problem, each arc (i, j) has a cost c_{ij} and a capacity u_{ij}, and every node i has a supply b_i. If $b_i > 0$, then the node can be considered a *source* (that is, it produces some good which needs to be transported). If $b_i < 0$, the node is considered a *sink* (that is, it consumes some good which must arrive from elsewhere). If $b_i = 0$, the node neither produces nor consumes anything, and is called a *transhipment node*.[4]

Figure 18 shows an example of this. In this case, there is just one source (node 1), which produces four units, and one sink (node 4), which consumes four units. The task is the figure out how to move these four units from node 1 to node 4 (that is, to decide the flow x_{ij} on each arc) in such a way that the total cost $\sum_{(i,j)\in A} c_{ij}x_{ij}$ is minimized.

Many algorithms exist to solve this problem, but the *successive shortest path* algorithm is perhaps the most intuitive. The algorithm finds the shortest path connecting an origin to a destination, and sends the most flow possible on that path (until either an arc reaches capacity, or all of the supply and demand is accounted for.) At this point, the next shortest path is found, the most flow sent on that path, and so on until every unit that needs to be transported is assigned. As with

[4]What if a node both produces and consumes a good? No problem! In this case b_i represents the *net* production (that is, the difference between the amount produced and consumed). If this difference is positive, the remaining amount must be transported away, and this is b_i. If negative, the absolute value is the amount which must be transported in, and again this is $|b_i|$.

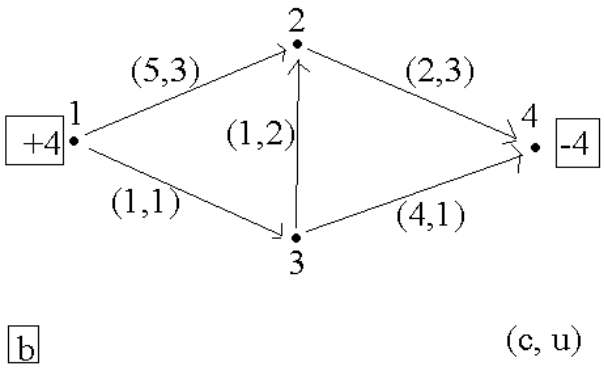


Figure 18: Example network for minimum cost flow.

the augmenting path algorithm, it is critical to use the residual graph to allow you to "correct" the flow as needed. (Otherwise, at each step you would have to think ahead to all of the future ramifications of sending flow on a particular path, which is very hard to do. Much easier to use the residual graph, which lets you fix the flow as needed.) The capacity on the residual graph is the same as for the maximum flow problem ($u_{ij} - xij$ for a forward arc (i, j), x_{ij} for a reverse arc (j, i)). The costs of arcs in the residual graph are the same as in the original graph for forward arcs (c_{ij} for (i, j)) and are changed in sign for reverse arcs ($-c_{ij}$ for (j, i)).

More precisely, the successive shortest path algorithm can be stated as follows:

1. Start with the zero flow $\mathbf{x} = 0$
2. Choose a source node r and a sink node s
3. Create the residual graph $R(\mathbf{x})$, and find the shortest path π from r to s with positive capacity.
4. Send the most amount of flow possible on this path: $\Delta = \min\{b(r), |b(s)|, u_{ij} : (i, j) \in \pi\}$.
5. Update the flow $\mathbf{x}$, and the supply and demand and the source and sink: $b(r) \leftarrow b(r) - \Delta$, $b(s) \leftarrow b(s) + \Delta$
6. If $b(i) = 0$ for all nodes, terminate. Otherwise, return to step 2.

(Recall that when sending flow on a path in the residual network, x_{ij} is increased for forward arcs, and decreased for backward arcs.)

Again consider the network in Figure 18. When applied to this network, the successive shortest path algorithm works as follows (you may find it useful to refer to Figure 19. In this figure, arcs with zero capacity in the residual graph are omitted for clarity.)

Iteration 1 . Start with the zero flow. The only source and sink are nodes 1 and 4, and the shortest path connecting these nodes is $(1, 2, 3, 4)$ which has a cost of 4 and a capacity of 1.

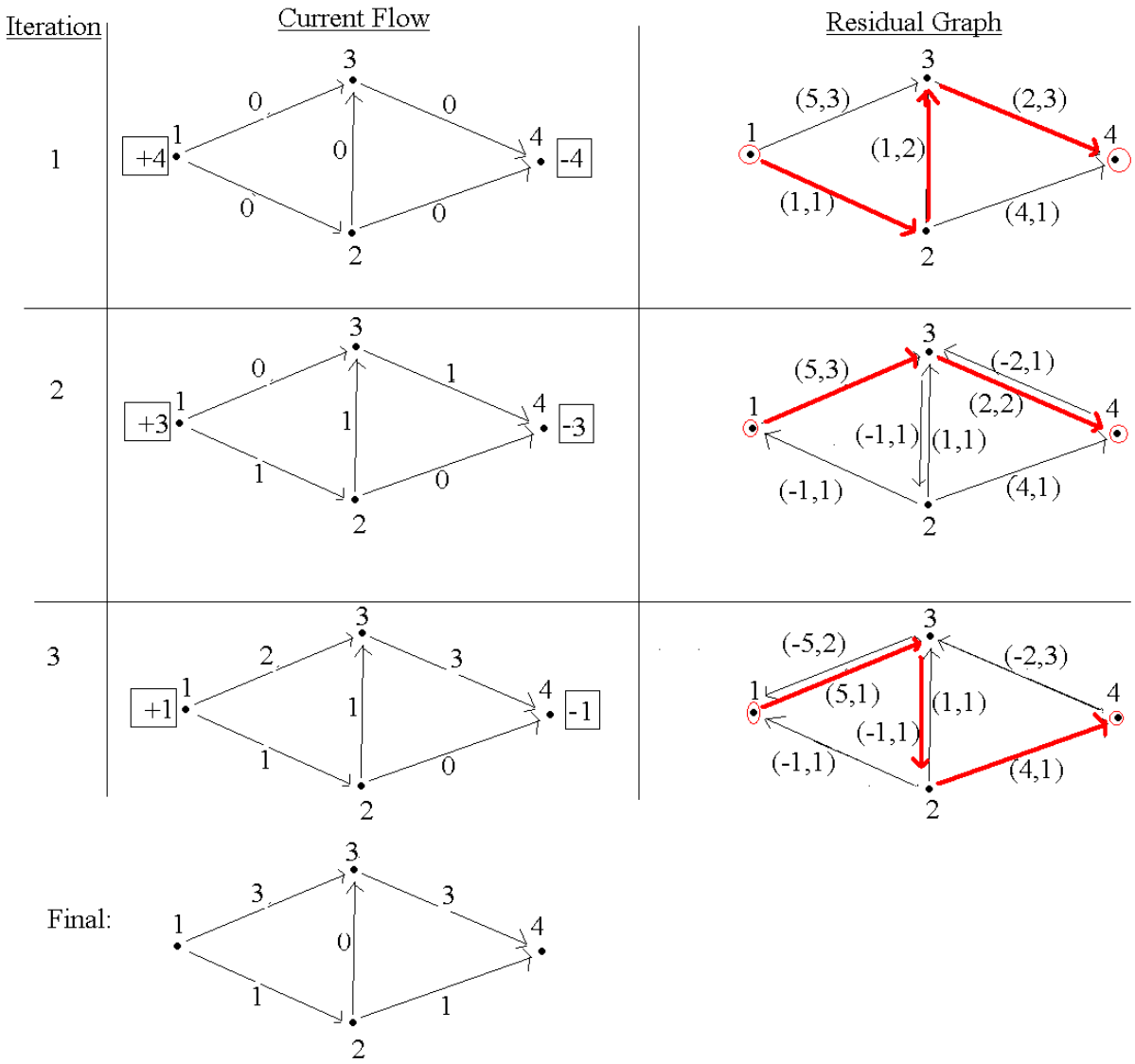


Figure 19: Example network for minimum cost flow.

The amount of flow we send is $\Delta = \min\{4, 4, 1\} = 1$, so $x_{12} = 1$, $x_{23} = 1$, $x_{34} = 1$, $b_1 = +3$, and $b_4 = -3$.

Iteration 2 . In the new residual graph, the shortest path connecting nodes 1 and 4 is now $(1, 3, 4)$ which has a cost of 7 and a capacity of 2. The amount of flow we send is $\Delta = \min\{3, 3, 2\} = 2$, so $x_{13} = 2$, x_{34} is increased by 2 to 3, $b_1 = +1$, and $b_4 = -1$.

Iteration 3 . In the new residual graph, the shortest path connecting nodes 1 and 4 is $(1, 3, 2, 4)$ which as a cost of 8 and a capacity of 1. The amount of flow we send is $\Delta = \min\{1, 1, 1\} = 1$, so x_{13} is increased to 3, x_{23} is *decreased* to 0, x_{24} is increased to 1, $b_1 = 0$, and $b_4 = 0$. At this point there are no more sources and sinks ($b(i) = 0$ for all nodes), so we are done.

Notice that Iteration 3 essentially "undid" part of Iteration 1 by removing flow from arc $(2, 3)$. However, you will notice that flow conservation is maintained, and on the whole more flow was still sent from node 1 to node 4 after that iteration — this is the magic of the residual network! At Iteration 1, there was no easy way to know that sending flow on $(2, 3)$ would turn out to be suboptimal... but this was not a problem.

If there are multiple sources and sinks, it does not matter how you pick r and s. For example, consider the slightly modified network in Figure 20, where now both nodes 1 and 2 are sources, and nodes 3 and 4 are sinks.

Iteration 1 . Start with the zero flow. Pick the source and sink arbitrarily; say $r = 1$ and $s = 3$. The shortest path connecting nodes 1 and 3 is $(1, 2, 3)$ with a cost of 2 and a capacity of 1. The amount of flow we send is $\Delta = \min\{2, 2, 1\} = 1$, so $x_{12} = 1$, $x_{23} = 1$, $b(1) = +1$, and $b(3) = -1$.

Iteration 2 . Pick the source and sink arbitrarily; say $r = 2$ and $s = 3$. The shortest path connecting nodes 2 and 3 in the residual graph is $(2, 3)$ with a cost of 1 and a capacity of 1. The amount of flow we send is $\Delta = \min\{2, 1, 1\} = 1$, so x_{23} is increased to 2, $b(2) = +1$, and $b(3) = 0$.

Iteration 3 . Pick the source arbitrarily (say $r = 2$); the only sink remaining is $s = 4$. The shortest path connecting nodes 2 and 4 in the residual graph is $(2, 4)$ with a cost of 4 and a capacity of 1. The amount of flow we send is $\Delta = \min\{1, 2, 1\} = 1$, so $x_{24} = 1$, $b(2) = 0$, and $b(4) = -1$.

Iteration 4 . The only source and sink remaining are $r = 1$ and $s = 4$. The shortest path connecting nodes 1 and 4 in the residual graph is $(1, 3, 4)$ with a cost of 7 and a capacity of 3. The amount of flow we send is $\Delta = \min\{1, 1, 3\} = 1$, so $x_{13} = 1$, $x_{34} = 1$, $b(1) = 0$, and $b(4) = 0$. No sources or sinks remain, so we are done.

In this case, the optimal cost is 14; no matter what order we chose sources and sinks in this algorithm, we would have ended up with a solution whose cost was 14 units as well.

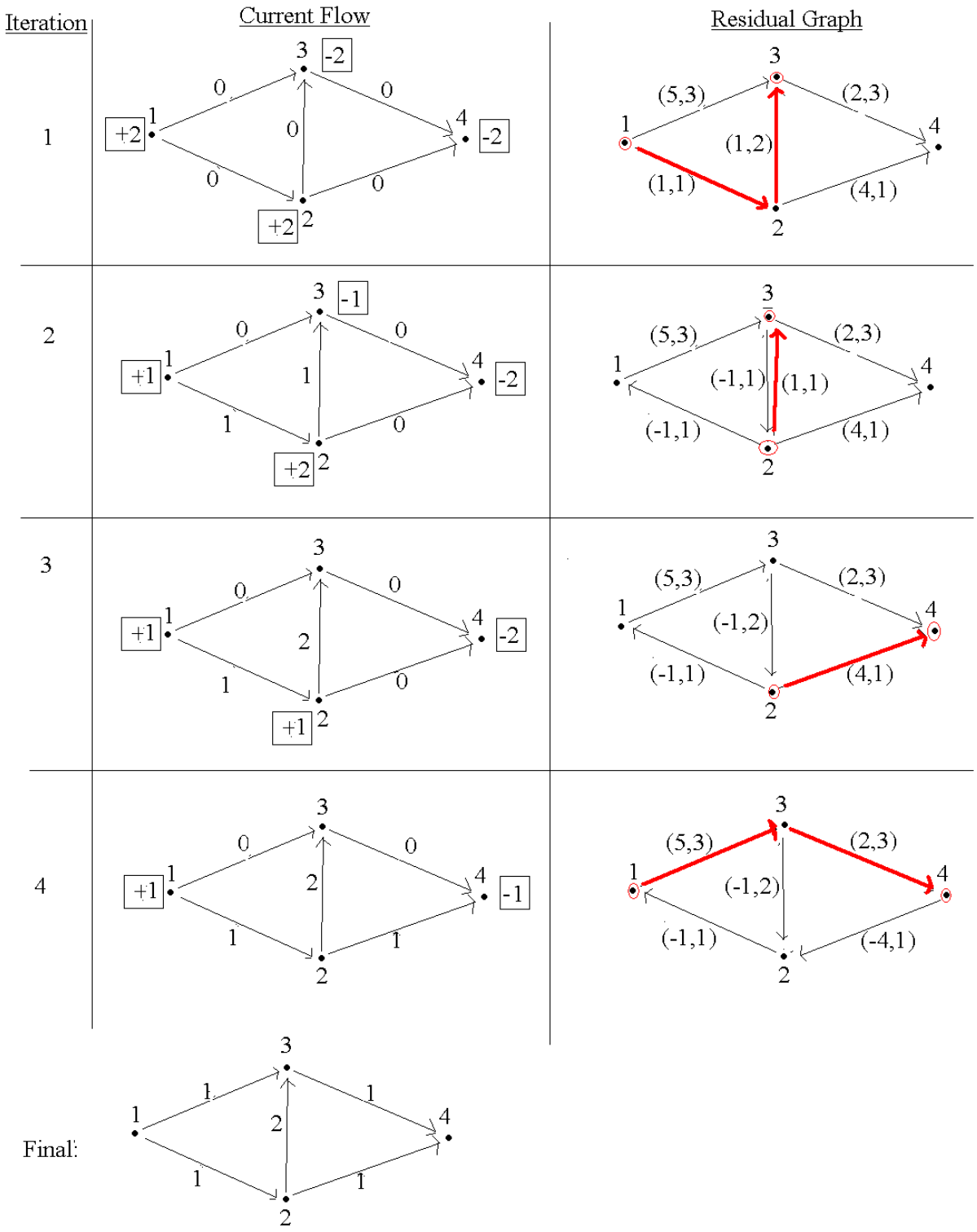


Figure 20: Example network for minimum cost flow.