9.0 Network Optimization

9.1 Network Analysis

We now turn to the analysis of problems which can be represented in a *network.* This type of problem is usually considered under the heading of *Combinatorial Optimization.* For these types of problems we no longer use calculus; the solutions are in the form of algorithms. The branch of mathematics behind network analysis is *graph theory* and thus, we often encounter dual terminology (e.g. node is equivalent to vertex). Electrical systems analysis is an obvious application, but other areas like project management and industrial engineering quickly come to mind.

9.2 Representation of Networks

A *network* (or sometimes *graph*) is a collection of *nodes* (or *vertices*) and *arcs* (or *edges*) connecting the nodes. In general, the arcs may be directed. Formally, we can represent a network G as an ordered triple:

$$G = (N(G), A(G), \psi_G)$$

where: $N(G)$ is a nonempty set of nodes, $A(G)$ is a set of arcs disjoint or independent of N, and ψ_G is an *incidence function* which associates a (not necessarily distinct) unordered pair of nodes of G to each arc of G. If the nodes are ordered, then it is a directed graph or network.

We will also use the notation $G = (V(G), E(G), \psi_G)$, and sometimes use the words vertices and edges, especially when the topics we are covering come from graph theory. Here, $V(G)$ is the vertex set, and $E(G)$ is the edge set.

Example: consider the undirected graph defined by

$$N(G) = \{n_1, n_2, n_3, n_4\}$$

$$A(G) = \{a_1, a_2, a_3, a_4, a_5, a_6\}$$

$$\psi_G(a_1) = n_1n2 \qquad \psi_G(a_2) = n_2n_3 \qquad \psi_G(a_3) = n_3n_3$$

$$\psi_G(a_4) = n_3n_4 \qquad \psi_G(a_5) = n_4n_1 \qquad \psi_G(a_6) = n_1n_2$$

A pictorial description of this graph is shown in Figure 9.1.

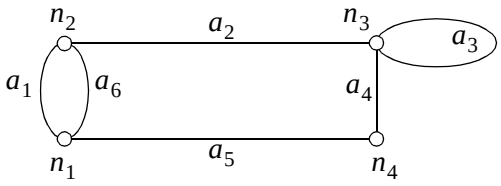


Figure 9.1 Example of an undirected graph.

If a graph contain no single arc loops or self-loop (*e.g.* a_3 above), and no two arcs join the same pair of nodes, then it is called a *simple graph*.

Example: consider the directed graph defined by

$$N(G) = \{n_1, n_2, n_3, n_4\}$$

$$A(G) = \{a_1, a_2, a_3, a_4, a_5, a_6\}$$

$$\psi_G(a_1) = n_1n_2 \qquad \psi_G(a_2) = n_2n_3 \qquad \psi_G(a_3) = n_3n_3$$

$$\psi_G(a_4) = n_3n_4 \qquad \psi_G(a_5) = n_4n_1 \qquad \psi_G(a_6) = n_2n_1$$

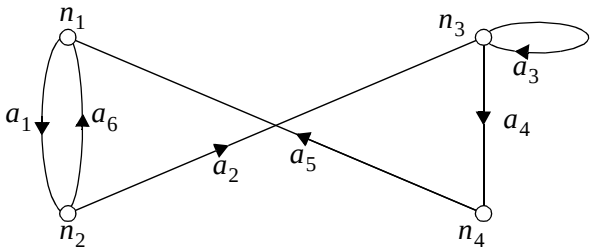


Figure 9.2 Example of a directed graph.

When the graph is directed the notation n_1n_2 represents an arc going from node n_1 to n_2 and is depicted as

Also, except for the fact that the last example is directed, the topology of the graph is the same (but drawn differently) as the previous example. Sometimes the notation (n_1n_2) is used to describe an arc.

A *bipartite graph* is a graph in which the vertex set, say $V(G)$, can be partitioned into two subsets, say *X* and *Y*, such that each edge of the graph has one end in *X* and the other end in *Y*. The partition (*X*, *Y*) is called the bipartition of the graph. The graph shown below in Figure 9.3 is obviously bipartite where the vertices corresponding to the two sets of the bipartition are depicted differently (*i.e.* solid and hollow).

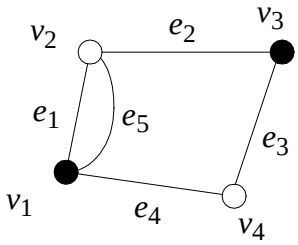


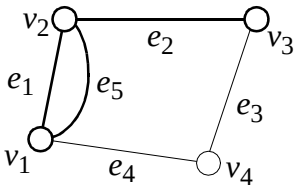
Figure 9.3 Example of a bipartite graph

Given two graphs *G* and *H*, *H* is said to be a *subgraph* of *G*, denoted $H \subseteq G$, if $V(\mathrm{H}) \subseteq V(G)$, $E(H) \subseteq E(G)$, and ψ_H is the restriction or imposition of ψ_G to $E(H)$. If $H \subseteq G$ but $H \neq G$ then *H* is said to be a proper subgraph of *G*, denoted $H \subset G$. Also, *G* is said to be a *supergraph* of *H*. A *spanning subgraph* of *G* is a subgraph *H* with $V(H) = V(G)$. If H_1 and H_2 are subgraphs of *G* then H_1 and H_2 are said to be *disjoint* if they have no vertex in common and *edge-disjoint* if they have no edge in common.

The *union* of two graphs H_1 and H_2, denoted $H_1 \cup H_2$, is a subgraph with vertex set $V(H_1) \cup V(H_2)$ and edge set $E(H_1) \cup E(H_2)$. If H_1 and H_2 are disjoint then their union is denoted $H_1 + H_2$. The *intersection* of H_1 and H_2, denoted $H_1 \cap H_2$, is a subgraph with vertex set $V(H_1) \cap V(H_2) \neq \varnothing$ and edge set $E(H_1) \cap E(H_2)$.

The *vertex degree* $d_G(v)$ of a vertex v in G is the number of edges of G incident with vertex v, where a loop (*i.e.* an edge with incidence function being an ordered pair of the same vertex) counts as two edges. The *minimum degree* of vertices of G is denoted $\delta(G)$ while the *maximum degree* is denoted $\Delta(G)$. It can be shown that in any graph the number of vertices of odd degree is even.

A *walk* in a graph G is a finite non-null sequence $W = v_0e_1v_1e_2v_2 \ldots e_kv_k$ whose terms are alternately vertices and edges such that for $1 \le i \le k$ the ends of e_i are the vertices v_{i-1} and v_i. The walk W may be denoted as $W = (v_0, v_k)$ and W is said to be a walk from vertex v_0 to vertex v_k. Also, v_0 is said to be the *origin* of the walk and v_k the *terminus*. The vertices v_1 to v_{k-1} are called *internal vertices* while k is the *length* ($\varepsilon(W)$) of the walk. A *section* of walk from v_i to v_j is a subsequence of W denoted as a (v_i, v_j) section of W. Walks can be *concatenated* if the terminus of one walk, say W, is the origin of another, say W', and the resulting walk is denoted by WW'. Walks can also be *inverted*, denoted W^{-1}, where the sequence is taken backwards from terminus to origin. Figure 9.4 shows an example of a walk.



$W = v_2e_5v_1e_1v_2e_2v_3$

v_2 – origin

v_3 – terminus

Figure 9.4 Example of a walk in a graph

If the edges of a walk W are distinct then W is called a *trail*; if the vertices are *also* distinct then W is called a *path*. In Figure 9.4, W is also a trail but not a path since vertex v_2 occurs twice in list.

Two vertices of a graph G, u and v say, are said to be *connected* if there is a (u, v) path in G. A graph G can be partitioned into c*omponents* $G[V_1]$, $G[V_2]$, ... , $G[V_\omega]$ of G, generated by partitioning V into nonempty subsets V_1, V_2, ... , V_ω such that two vertices of V are connected if and only if they belong to the same subset V_i. The graph G is said to be *connected* if all its vertices are connected, otherwise it is called a *disconnected* graph.

A *closed* walk is one in which the origin and terminus are the same. A *cycle* is a closed trail wherein the origin and the internal vertices are distinct. A *k-cycle* is a cycle of length k while a

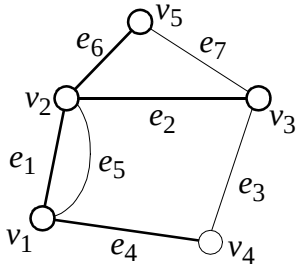
3-cycle is also called a triangle. It can be shown that a graph is bipartite if and only if it contains no odd cycles. An *acyclic* graph is one which contains no cycles. An acyclic graph which is also connected is called a *tree*. In a tree, any two vertices are connected by a unique path.

A tree of a graph G is a subgraph G_1 of G such that any two of the following are true:

1) the subgraph G_1 is connected,
2) G_1 has no cycles,
3) the number of edges in G_1 is $k-1$,

where G has n vertices and G_1 has k vertices with $k \le n$.

A *spanning tree* is a tree of G having the same number of vertices as G, and therefore, $n-1$ edges. Examples are shown in Figure 9.5. It is usual to represent a tree by the set of edges in the tree; the vertices will be implied.



$T = \{e_1, e_2, e_6\}$

$ST = \{e_1, e_2, e_4, e_6\}$

Figure 9.5 Examples of a tree and spanning tree in a graph.

A graph G is called a *weighted graph* if for every edge $e \in E(G)$ there exists a weight $w(e) \in \mathbb{R}$ (real number). If H is a subgraph of G then the weight of H is given as the sum of the individual edge weights in H

$$w(H) = \sum_{e \in E(H)} w(e) \tag{9.1}$$

and the weight of a path in G is called the *length* of the path. The minimum weight of a (u,v) path is called the *distance* $d(u, v)$. If two vertices, u and v, are not connected by an edge, that is if $uv \notin E$, then the weight of uv is assumed infinite (*i.e.* $w(uv) = \infty$).

9.2.1 Matrix representation of a graph

The adjacency matrix is another way of storing the knowledge of a graph. Consider a graph which is represented by $G = (N, A)$, where the nodes are numbered sequentially, say $N = \{1, 2, \ldots n\}$ and the edges are listed as pairs of numbers representing the nodes which terminate the edge, say $\{(1, 2), (5, 10), \ldots\}$. The adjacency matrix $X = [x_{ij}] \in \mathbb{R}^{n \times n}$ is defined such that

$$x_{ij} = \begin{cases} 1 & (i, j) \in A \\ 0 & \text{otherwize} \end{cases}$$

where $i, j \in N$. If G is undirected, then $x_{ij} = x_{ji}$ and the adjacency matrix X is symmetric.

Similarly, we can define the cost matrix, $C = [c_{ij}] \in \mathbb{R}^{n \times n}$, where c_{ij} is the cost or weight of edge (i, j). In an undirected graph, the cost matrix will be symmetric, $c_{ij} = c_{ji}$. If an edge between two nodes i and j does not exist then the cost will be defined as infinite, $c_{ij} = \infty$.

Example: The graph shown in Figure 9.6 has an adjacency matrix given by

$$X = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}.$$

The rows represent the source of an edge while the columns represent the sink of an edge. A 1 on the diagonal represents a loop.

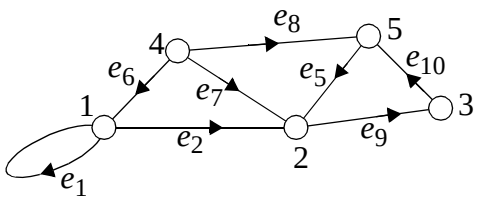


Figure 9.6 Directed graph for which adjacency matrix is defined.

Another way to represent the graph using a matrix is via the so-called node-arc incidence matrix $Z = [z_{ik}] \in \mathbb{R}^{n \times m}$ where n is the number of nodes and m is the number of arcs. In this case, we must assume that the nodes as well as the arcs are numbered consecutively, as in say

$$N = \{1, 2, \ldots n\}, A = \{a_1, a_2 \ldots a_m\}$$

and that each arc can be represented as a pair of nodes, say $a_k = (i, j)$ where i is the source node and j is the sink node of the arc. Then the node-arc incidence matrix is given by

$$Z = [z_{ik}], z_{ik} = \begin{cases} 1 & a_k = (i, j) \\ -1 & a_k = (j, i) \\ 0 & \text{otherwize} \end{cases}$$

where $i, j \in N$ and $a_k \in A$. For an undirected network G, we use only positive 1's in the matrix:

$$Z = [z_{ik}], z_{ik} = \begin{cases} 1 & a_k = (i, j) \text{ or } a_k = (j, i) \\ 0 & \text{otherwize} \end{cases}$$

Example: The graph shown in Figure 9.7 has an adjacency matrix given by

$$Z = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

The rows represent the noders while the columns represent the edges. Each column has exactly 2 non-zero entires.

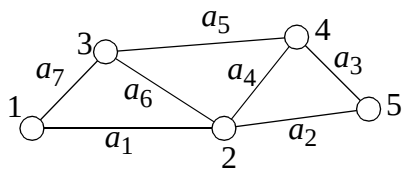


Figure 9.7 Graph for which node-arc incidence matrix is defined.

9.3 Finding a Spanning Tree

There are two fundamental search techniques which can be used for find a spanning tree. The first is called Breadth-First Search and the second is called Depth-First Search. These are fundamental general purpose searching strategies which are used in many applications.

9.3.1 Breadth-First Search for a Spanning Tree

Given a graph and any starting node, the strategy is to find all new nodes which can be reached on an edge from the current node. Then we visit each of the nodes in the same order in which they found applying the algorithm. The method is exemplified by an example.

Example: Breadth-first search of a graph for a spanning tree

Consider the graph shown in Figure 9.8.

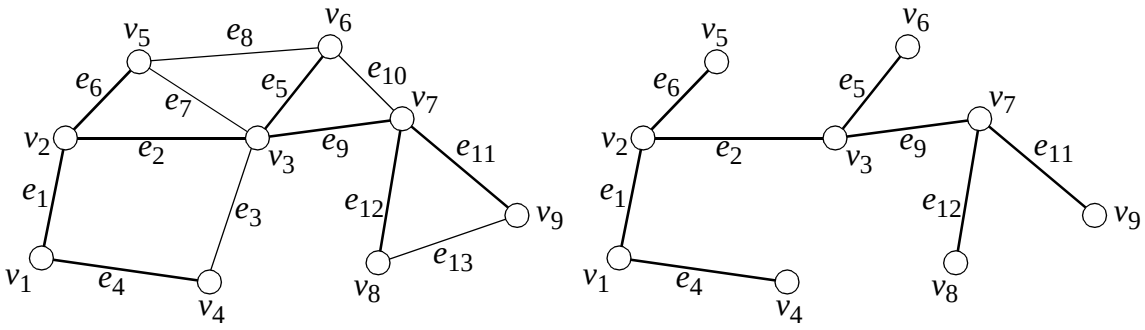


Figure 9.8 Breadth-first search for a spanning tree.

If we start with node v_1 then we will take the following steps:

1) add edges $\{e_1, e_4\}$, nodes in the tree are $\{v_1, v_2, v_4\}$
2) move to node v_2 and add edges $\{e_2, e_6\}$, nodes in the tree are $\{v_1, v_2, v_4, v_3, v_5\}$
3) move to node v_4, no edges to add
4) move to node v_3 and add edges $\{e_5, e_9\}$, nodes in the tree are $\{v_1, v_2, v_4, v_3, v_5, v_6, v_7\}$
5) move to node v_5, no edges to add
6) move to node v_6, no edges to add
7) move to node v_7, and add edges $\{e_{11}, e_{12}\}$, nodes in the tree are now $\{v_1, v_2, v_4, v_3, v_5, v_6, v_7, v_8, v_9\}$ and since all nodes are covered, we stop.

9.3.2 Depth-First Search for a Spanning Tree

The depth-first search technique takes a different philosophy. In this algorithm we keep moving to a node as we add it into the tree and try to add a new edge from that new node. If we can't add a new node, we backtrack to the first node in which we can continue to proceed forward.

The method is best explained by an example. Again we consider the same example which was used for the breadth-first search.

Example: Depth-first search of a graph for a spanning tree. Consider the graph in Figure 9.9.

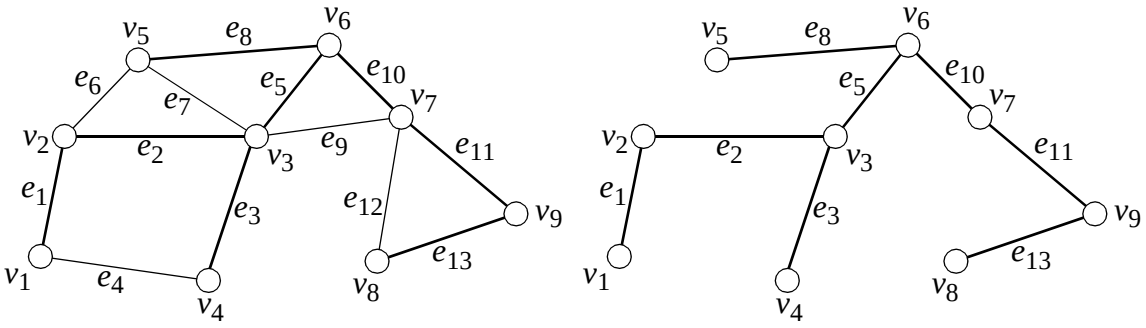


Figure 9.9 Depth-first search for a spanning tree.

If we start with node v_1 then we will take the following steps:

1) add edges $\{e_1\}$, nodes in the tree are $\{v_1, v_2\}$
2) move to node v_2 and add edge $\{e_2\}$, nodes in the tree are $\{v_1, v_2, v_3\}$
3) move to node v_3 and add edge $\{e_3\}$, nodes in the tree are $\{v_1, v_2, v_3, v_4\}$
4) move to node v_4, can't add an edge
5) backtrack to node v_3 and add edge $\{e_5\}$, nodes in the tree are $\{v_1, v_2, v_3, v_4, v_6\}$
6) move to node v_6 and add edge $\{e_8\}$, nodes in the tree are $\{v_1, v_2, v_3, v_4, v_6, v_5\}$
7) move to node v_5, can't add an edge
8) backtrack to node v_6 and add edge $\{e_{10}\}$, nodes in the tree are $\{v_1, v_2, v_3, v_4, v_6, v_5, v_7\}$
9) move to node v_7 and add edge $\{e_{11}\}$, nodes in the tree are $\{v_1, v_2, v_3, v_4, v_6, v_5, v_7, v_9\}$
10) move to node v_9 and add edge $\{e_{13}\}$, nodes in the tree are $\{v_1, v_2, v_3, v_4, v_6, v_5, v_7, v_9, v_8\}$.
11) All nodes are in the tree so stop.

Note that the spanning trees which are obtained using these techniques are not the same and that, even when using the same algorithm, the tree which is obtained will depend on: a) the starting node; and b) the order in which the edges are stored in the list of edges.

9.4 Minimum Spanning Tree

The two algorithms for finding a spanning tree are quite simple to understand. Now if we have a weighted graph and we want to find the spanning tree of minimum weight, how should we proceed? We will discuss two algorithms for finding such a minimum spanning tree: Prim's algorithm and Kruskal's minimum forest algorithm. These can be explained by considering the following example.

Example: Say we have the weighted network shown in Figure 9.10 where the numbers along the arcs represent the weight of the arc. Now assume that we already have the two bold arcs in the minimum spanning tree; that is the arcs with weights 1 and 2. Should we next add the arc labelled 7? This is the lowest weight arc incident on the existing tree. Alternatively, should we add the arc with weight 3? This is the lowest weight arc in the entire graph which could be legally added to the tree. The first choice is representative of Prim's algorithm while the second choice is representative of Kruskal's minimum forest algorithm.

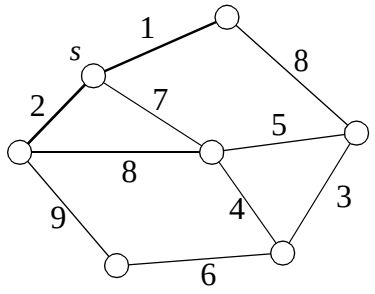


Figure 9.10 Example weighted graph for which to find a minimum spanning tree.

9.4.1 Prim's Algorithm

Prim's algorithm is stated as follows. Start from any node s and build a tree by repeating the following rule: add the shortest edge that is incident to the existing tree. For the example given in Figure 9.10, if we started with the node labelled s, we would add the following edges (using the weight as the identifier in this case): $\{1, 2, 7, 4, 3, 6\}$ for a total weight of 22.

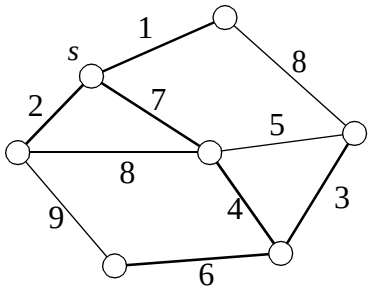


Figure 9.11 Minimum spanning tree using Prim's or Kruskal's algorithm.

9.4.2 Kruskal's minimum forest algorithm

In this algorithm we add edges in increasing order of length, rejecting any that complete a loop. For the example of Figure 9.10 we would add edges in the following order: $\{1, 2, 3, 4, 6, 7\}$. Therefore we would arrive at the same minimum spanning tree as that shown in Figure 9.11 but in a different order of adding edges. In general the minimum spanning tree on a weighted graph may not be unique but it will definitely have a unique weight.

9.5 Shortest Path Algorithms

A graph algorithm for finding the shortest path from a root vertex u_0 to all other vertices in the graph is *Dijkstra's algorithm* named after its founder (see Dijkstra [2], and [1], pp 19 - 20). Actually, in Dijkstra's 1959 article he gives two algorithms. The first finds the tree of minimum weight which spans a connected graph *G*, called the *minimum spanning tree* of the graph. The second finds the path of minimum distance between a root vertex u_0 and the rest of the vertices of *G*, called the *single source shortest distance spanning tree*.

The algorithm uses the fact that if *S* is a proper subset of *V* (the set of n+1 vetices), where the root vertex is chosen so that it is an element of *S*, then

$$\mathrm{d}(u_0, \bar{S}) = \min\{\mathrm{d}(u_o, u) + w(uv)\}\Big|_{u \in S,\, v \in \bar{S}} \tag{9.2}$$

where $\bar{S}$ is V - S. An increasing sequence of subsets $S_0, S_1, \ldots, S_n$ of V is constructed (with $S_0 = \{u_0\}$) so that once the i^{th} subset is constructed the shortest paths from the root vertex u_0 to all the vertices in S_i will be known. These shortest paths will be denoted $P(u_0v)$ where $v \in S_i$. The algorithm is shown in Figure 9.12.

Dijkstra's Algorithm

$S = \{u_0\}$; $\bar{S} = V - S$; $L(u_0) = 0$

***Loop* 1:** $\forall v \in \bar{S}$

$L(v) = \infty$; $P(u_0\, v) = \{u_0\}$

***end Loop* 1**

$i = 0$;

Loop* 2: *while $\bar{S} \neq \varnothing$

$min\_label = \infty$

***Loop* 3:** $\forall v \in \bar{S}$

if $L(u_i) + w(u_i\, v) < L(v)$ ***then***

$L(v) = L(u_i) + w(u_i\, v)$

$P(u_0\, v) = P(u_0\, u_i) \cup v$

endif

if $L(v) < min\_label$ ***then***

$min\_v = v$; $min\_label = L(v)$

endif

end Loop 3

$u_{i+1} = min\_v$; $S = S \cup u_{i+1}$; $\bar{S} = \bar{S} - u_{i+1}$; $i = i + 1$

Figure 9.12 Dijkstra's algorithm with shortest path determination

When the algorithm terminates, the labels at each vertex $L(v)$, $v \in V$, will contain the distance from the root vertex u_0 and the path variable $P(u_0 v)$ will contain the minimum path specification as a list of vertices from the root vertex to v. Note that a list of vertices is sufficient to represent a path in a simple graph. An example of the use of this algorithm is given below in Figure 9.3 where the labels and paths after each step are shown next to the graph. The edge weights and the root vertex are identified in the first drawing.

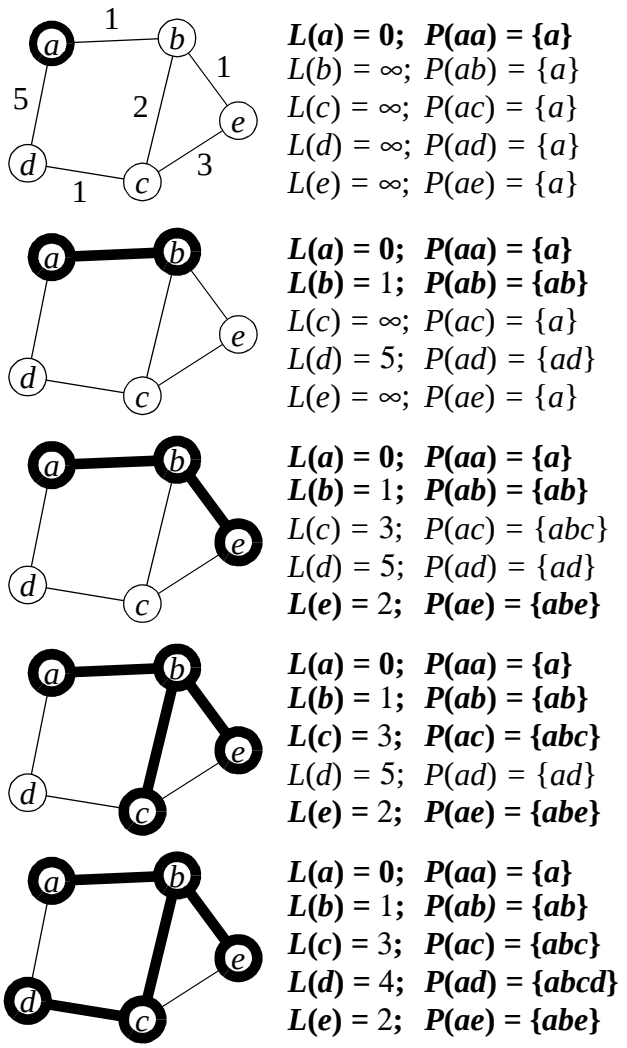


Figure 9.13 Example application of shortest path algorithm

In Figure 9.3 the bold vertices represent the elements of each subset *S* after each step in the algorithm. The final paths are shown in the bottom diagram of the figure with the distances given by the labels. Of course, if the distance to only one vertex from the root vertex is required, the algorithm is terminated when that vertex is reached. This algorithm, is a *good* algorithm in that it finds the solution in polynomial time (*i.e.* it is of order ν^2 where ν is the number of vertices).

9.5.1 References

[1] J. A. Bondy, and U. S. R. Murty, Graph Theory with Applications, American Elsevier Pub. Co., Inc., 1976.

[2] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs", Numerische Mathematik, vol. 1, pp. 269 - 271, 1959.

9.6 Shortest Distance Between All Pairs of Nodes (Floyd-Warshall Algorithm)

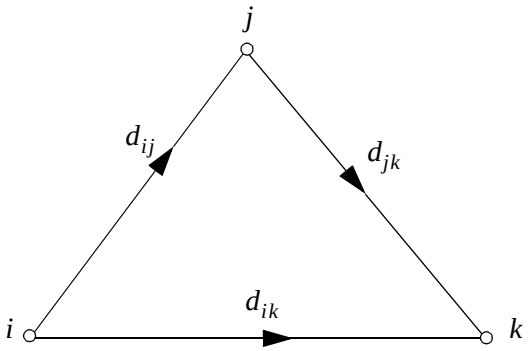
We now want to solve for the shortest path between all nodes in a graph $G = (N, A)$. To do this, we could apply Dijkstra's algorithm, which gives us the shortest distance from a root node to all other nodes, n times. However, there is a more efficient algorithm due to Floyd and Warshall (1962).

Let $N = \{1, 2, \ldots, n\}$ be the set of nodes and let the matrix $C = [c_{ij}]$ be the $n \times n$ matrix of weights, so that c_{ij} is the weight of arc (i, j). We now create a $n \times n$ matrix of distances from *i* to *j* : $D = [d_{ij}]$ such that d_{ij} will be the minimal distance from node *i* to *j*. Initially, we set $D = C$ and then we iterate *n* times. After *n* iterations, *D* will contain the minimal distances between each pair of nodes. (In this algorithm, we assume that $d_{ii} = \infty$) We first define the so-called *triangle* operation.

Definition: *triangle* or *triple* operation

Given an $n \times n$ distance matrix d_{ij}, the triangle operation for a fixed node *j* is

$$d_{ik} = \min\{d_{ik}, d_{ij} + d_{jk}\} \qquad \forall\, i, k = 1(1)n \quad i, k \neq j$$



Theorem: If we perform the triangle operation for successive values $j = 1(1)n$, each entry d_{ik} becomes equal to the length of the shortest path from *i* to *k*, assuming weights $c_{ij} \geq 0$.

We now give the Floyd-Warshall algorithm which uses the triangle operation.

Algorithm: Floyd-Warshall Algorithm

Input: $n \times n$ matrix $[C_{ij}]$ (non-negative entries)

Output: $n \times n$ matirx $[d_{ij}]$, where d_{ij} is shortest distance from *i* to *j* under $[C_{ij}]$.

for all $i \neq j$ do $d_{ij} = C_{ij}$,

for $i = 1(1)n$ do $d_{ii} = \infty$

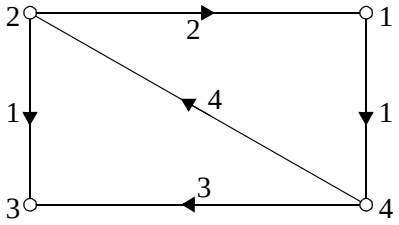
for $j = 1(1)n$ do

for $i = 1(1)n$ $\quad i \neq j$ do begin

for $k = 1(1)n$ $\quad k \neq j$ do begin

$$d_{ik} = \min\{d_{ik}, d_{ij} + d_{jk}\}$$

Example: consider the following graph:



$$C = \begin{bmatrix} 0 & \infty & \infty & 1 \\ 2 & 0 & 1 & \infty \\ \infty & \infty & 0 & \infty \\ \infty & 4 & 3 & 0 \end{bmatrix}$$

now we initialize the D matrix

$$[d_{ij}] = D = \begin{bmatrix} \infty & \infty & \infty & 1 \\ 2 & \infty & 1 & \infty \\ \infty & \infty & \infty & \infty \\ \infty & 4 & 3 & \infty \end{bmatrix}$$

← pivot row

elements to be updated in next generation, e.g.: $d_{24} = \min\{\infty, 2 + 1\} = 3$

pivot column

$$j = 1 \qquad D = \begin{bmatrix} \infty & \infty & \infty & 1 \\ 2 & \infty & 1 & 3 \\ \infty & \infty & \infty & \infty \\ \infty & 4 & 3 & \infty \end{bmatrix}$$

$$j = 2 \qquad D = \begin{bmatrix} \infty & \infty & \infty & 1 \\ 2 & \infty & 1 & 3 \\ \infty & \infty & \infty & \infty \\ 6 & 4 & 3 & 7 \end{bmatrix}$$

$$j = 3 \qquad D = \begin{bmatrix} \infty & \infty & \infty & 1 \\ 2 & \infty & 1 & 3 \\ \infty & \infty & \infty & \infty \\ 6 & 4 & 3 & 7 \end{bmatrix}$$

$$j = 4 \qquad D = \begin{bmatrix} 7 & 5 & 4 & 1 \\ 2 & 7 & 1 & 3 \\ \infty & \infty & \infty & \infty \\ 6 & 4 & 3 & 7 \end{bmatrix}$$

Which gives the final answer:

$$\begin{array}{llll} d_{11} = 7 & d_{12} = 5 & d_{13} = 4 & d_{14} = 1 \\ d_{21} = 2 & d_{22} = 7 & d_{23} = 1 & d_{24} = 3 \\ d_{31} = \infty & d_{32} = \infty & d_{33} = \infty & d_{34} = \infty \\ d_{41} = 6 & d_{42} = 4 & d_{43} = 3 & d_{44} = 7 \end{array}$$

Note that starting the diagonals with $d_{ii} = \infty$ finds the paths which are chains (or cycles). That is, we ended up with $d_{11} = 7, d_{22} = 7, d_{33} = \infty, d_{44} = 7$ which are cycles.

Now how do we keep track of the shortest path? We keep track of a new matrix which is initially set to zero, called the *route matrix*:

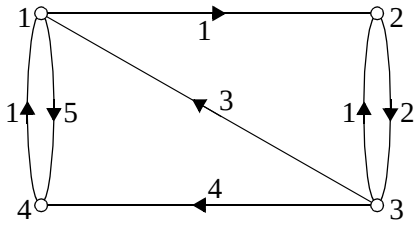
$$E = [e_{ik}] \qquad e_{ik} = 0 \qquad i, k = 1(1)n$$

Then every time we apply the triangle operation in the algorithm, we also execute:

$$e_{ik} = \begin{cases} j & \text{if } d_{ik} > d_{ij} + d_{jk} \\ e_{ik} & \text{otherwize} \end{cases}$$

i.e. if an intermediate node, *j*, can be used to create a shorter path, then we should store it.

Example:



$$C = \begin{bmatrix} 0 & 1 & \infty & 5 \\ \infty & 0 & 2 & \infty \\ 3 & 1 & 0 & 4 \\ 1 & \infty & \infty & 0 \end{bmatrix}$$

$$D^0 = \begin{bmatrix} \infty & 1 & \infty & 5 \\ \infty & \infty & 2 & \infty \\ 3 & 1 & \infty & 4 \\ 1 & \infty & \infty & \infty \end{bmatrix} \qquad E^0 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$D^1 = \begin{bmatrix} \infty & 1 & \infty & 5 \\ \infty & \infty & 2 & \infty \\ 3 & 1 & \infty & 4 \\ 1 & 2 & \infty & 6 \end{bmatrix} \qquad E^1 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$ the only changes

$$D^2 = \begin{bmatrix} \infty & 1 & 3 & 5 \\ \infty & \infty & 2 & \infty \\ 3 & 1 & 3 & 4 \\ 1 & 2 & 4 & 6 \end{bmatrix} \qquad E^2 = \begin{bmatrix} 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 2 & 1 \end{bmatrix}$$

$$D^3 = \begin{bmatrix} 6 & 1 & 3 & 5 \\ 5 & 3 & 2 & 6 \\ 3 & 1 & 3 & 4 \\ 1 & 2 & 4 & 6 \end{bmatrix} \qquad E^3 = \begin{bmatrix} 3 & 0 & 2 & 0 \\ 3 & 3 & 0 & 3 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 2 & 1 \end{bmatrix}$$

$$D^4 = \begin{bmatrix} 6 & 1 & 3 & 5 \\ 5 & 3 & 2 & 6 \\ 3 & 1 & 3 & 4 \\ 1 & 2 & 4 & 6 \end{bmatrix} \qquad E^4 = \begin{bmatrix} 3 & 0 & 2 & 0 \\ 3 & 3 & 0 & 3 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 2 & 1 \end{bmatrix}$$

Note that there was no change during the last step. Now to find the paths we proceed as follows:

To find the path from node 2 to node 1: $P_{21}: d_{21} = 5, e_{21} = 3, e_{23} = 0 \Rightarrow P_{21} = \{2, 3, 1\}$

To find the path from node 1 to node 3: $P_{13}: d_{13} = 3, e_{13} = 2, e_{12} = 0 \Rightarrow P_{13} = \{1, 2, 3\}$

To find the path from node 1 to node 1: $P_{11}: d_{11} = 3, e_{11} = 3, e_{13} = 2, e_{12} = 0$

$$\Rightarrow P_{11} = \{1, 2, 3, 1\}$$

The procedure is as follows:

$P_{ij} = \{j\}$ add terminal node

$k = j$

while $e_{ik} \neq 0$

$P_{ij} = e_{ik} \cup P_{ij}$

$k = e_{ik}$

continue

$P_{ij} = i \cup P_{ij}$ add source node

9.7 Maximum Flow - Network Problems

We now consider networks, $G = (N, A)$, where the arcs are weighted by some sort of *capacity, e.g.*: 2 messages per hour, 10 gallons per minute, 100 cars per minute, and a source node, $s\ \{ = 0\}$, and terminal (or destination) node, $t\ \{= m\}$ (*i.e.* $|N| = m + 1$). To be clear, the capacity here is different than weight; when an arc is traversed, the total weight of the arc is used, but not necessarily the total capacity.

Example:

Consider the example network shown in Figure 9.14. The capacity of arc (1,2) is denoted f_{12} and for a general arc $(i, j) \rightarrow f_{ij}$. If (i, j) doesn't exist then we set $f_{ij} = 0$. In this example, we have for instance: $f_{01} = 2, f_{12} = 4, f_{13} = 1$. We assume an unlimited supply of commodity at the source *s*.

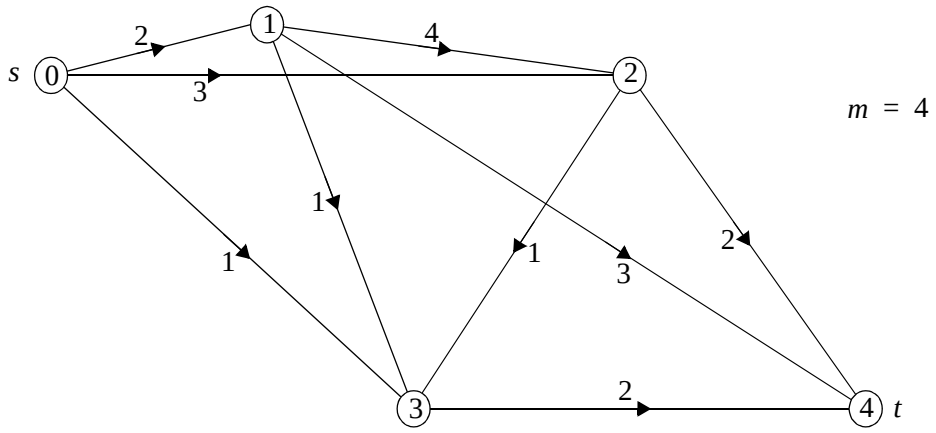


Figure 9.14 Example network with capacities on the arcs.

Let: x_{ij} represent the amount of flow from node i to j

f represent total flow from s to t

f_{ij} maximum flow or *capacity* from node i to j

We assume that the flow and capacities are non-negative and thus:

$$x_{ij} \geq 0 \qquad f_{ij} \geq 0 \qquad f \geq 0$$

We also assume that no commodity is produced at intermediate nodes $\Rightarrow$ *conservation of flow.* Therefore we have that the flow into node i equals the flow out of node i, and this can be expressed as

$$\sum_{i=0}^{m} x_{ji} = \sum_{i=0}^{m} x_{ij} \qquad i = 1(1)m-1.$$

Note that the conservation of flow is satisfied at all interior nodes (*i.e.* excluding the source and destination nodes). The remaining constraints are that the flow must be less than the capacity for each arc:

$$x_{ij} \leq f_{ij} \qquad i = 0(1)n \qquad j = 0(1)n. \blacksquare$$

9.8 Maximum Flow Problem:

How do we get as much of the commodity as possible through the network from source to terminal?

Mathematically, we can state the objective as:

$$\underset{x}{\text{maximize}} \qquad f = \sum_{i=1}^{m} x_{oi}$$

which states that we wish to maximize the flow leaving the source node (0).

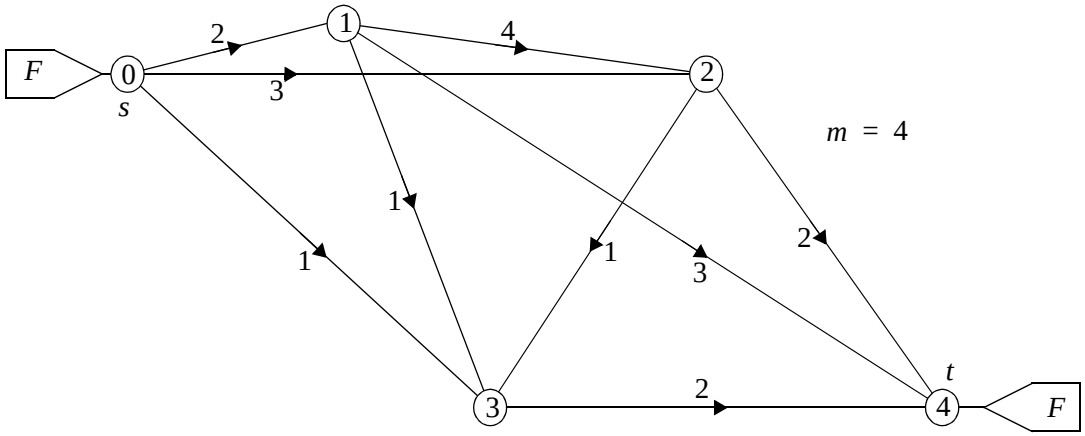
At the terminal node, the flow will accumulate and thus, do to the conservation of flow at the interior nodes, we have:

$$f = \sum_{i=0}^{m-1} x_{im}$$

which is the flow into sink node. We could have just as easily maximized this function. Including it as one of the constraints *may* simplify the solution.

Example:

As we saw before, we can consider the maximum flow problem as a linear program.



$$\underset{x}{\text{maximize}} \qquad f = x_{01} + x_{02} + x_{03}$$

subject to, conservation of flow:

$$x_{01} = x_{12} + x_{13} + x_{14}$$

$$x_{02} + x_{12} = x_{23} + x_{24}$$

$$x_{03} + x_{13} + x_{23} = x_{34}$$

maximum flow in each arc:

$$x_{01} \le 2 \qquad x_{02} \le 3 \qquad x_{03} \le 1$$

$$x_{12} \le 4 \qquad x_{13} \le 1 \qquad x_{14} \le 3$$

$$x_{23} \le 1 \qquad x_{24} \le 2$$

$$x_{34} \le 2$$

non-negative flow:

$$x_{ij} \ge 0 \qquad f \ge 0$$

and that the flow into the sink node must be the same as the flow out of the source node:

$$\{x_{14} + x_{24} + x_{34} = f\} \text{ (not necessary). } \blacksquare$$

Solving the maximum flow problem using linear programming is not efficient, and faster algorithms have been developed. Before we study these algorithms, let us define some terms and state a theorem.

Definition: *Cut in the network*

A cut in a network produces a separation of nodes into 2 groups, S and T, such that $s \in S$ and $t \in T$. ❒

Definition: *Capacity of cut*

The total capacity on edges crossing from the set S to the set T is called the capacity of the cut. The flow cannot exceed this capacity. ❒

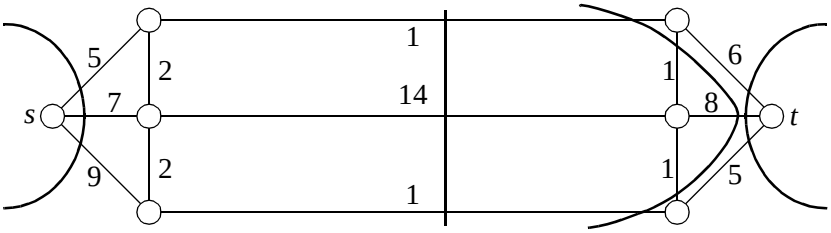
Definition: *Minimal cut*

The minimal cut is the cut of the smallest possible capacity. ❒

Theorem: *Max-flow/Min-cut*

The value of the maximum flow equals the capacity of the minimal cut. ❒

Example:



Cuts: $5 + 7 + 9 = 21$

$6 + 8 + 5 = 19$

$1 + 14 + 1 = 16$

$1 + 1 + 8 + 1 + 1 = 12$ (minimal cut = maximum flow)

Note that the minimal cut is not unique, but it's value is. ■

9.9 Ford and Fulkerson Labeling Algorithm for Maximum Flow Problems

Definition: Augmentation (or augmenting) path P

Given a flow network $N = (s, t, V, E, C)$ where:

s is the *source* node

t is the *terminus* node

V is the *set of nodes*

E is the *set of arcs*

C is the *capacities*

and a feasible $s-t$ flow f_{st}, an augmentation path P is a path from s to t in the undirected graph G, resulting from N by ignoring the arc directions, with the following properties:

1. for every arc $(i, j) \in E$ that is traversed by P in the forward direction (called a *forward arc*), we have $f_{ij} < c_{ij}$. That is forward arcs of P are *unsaturated*.
2. for every arc $(i, j) \in E$ that is traversed by P in the reverse direction (called a *backward arc*) we have $f_{ji} > 0$. ❐

Give an augmentation path, we can *increase* the *flow* from s to t while *maintaining flow conservation* at every node by increasing the flow on every forward arc of P and decreasing it along every backward arc.

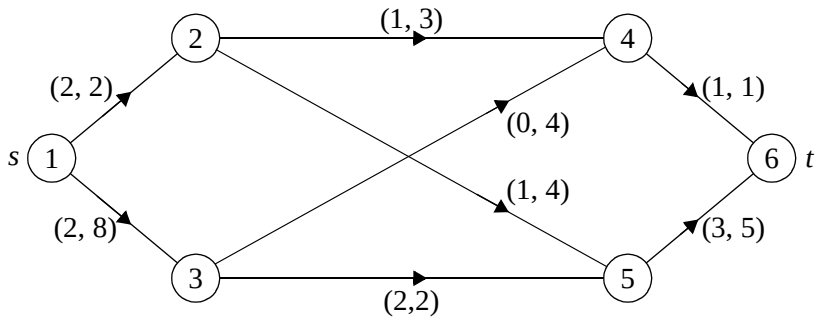
The *maximum amount* of flow augmentation possible along P is

$$\delta = \underset{e_{ij} \in P}{\text{minimum}} \left\{ \begin{array}{ll} c_{ij} - f_{ij} & \text{along a forward arc} \\ f_{ji} & \text{along a backward arc} \end{array} \right\}$$

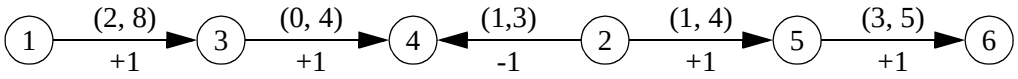
where $e_{ij} = (i, j)$ is an arc in the path.

Example:

Consider the following network where the label $(2, 4) \Rightarrow f_{ij} = 4, c_{ij} = 4$, that is the flow equals 2 and the capacity equals 4. This instance of this network has feasible flow $f = 4$ but the question is whether or not it is maximum possible flow obtainable in the network.



Now consider the following augmentation path:



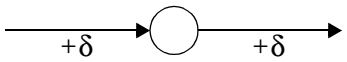
$$\delta = 1 = \underset{e_{ij} \in P}{\text{minimum}} \left\{ \begin{array}{ll} c_{ij} - f_{ij} & \text{along a forward arc} \\ f_{ji} & \text{along a backward arc} \end{array} \right\}$$

Thus, the flow can be imcreased by $\delta = 1$ on this augmentation path. ■

Theorem: a feasible flow f is maximum if there is no augmentation path with respect to f. ❒

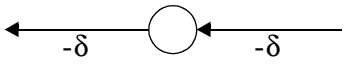
The following comments can be made about this procedure:

1. if a node has both forward arcs, as in



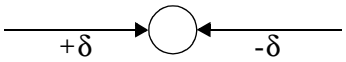
then the flow in and the flow out goes up by δ which implies that the flow is conserved.

2. If a node has both arcs as backward arcs, as in



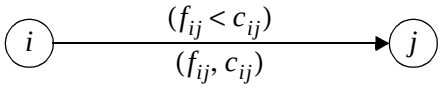
then again the flow is conserved.

3. If a node has one of each, the flow is still conserved:



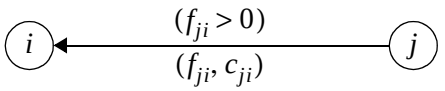
Thus, if we cannot find an augmenting path for a labeled network, then that flow must be maximum. Now, how do we use this theorem to determine a algorithm which will find the maximum flow? The algorithm we will describe is attributed to Ford and Fulkerson (1956).

In the algorithm, all nodes $j \neq s$ get a label of the form $(p(j), \Delta)$ where $p(j)$ is the node from which j receives flow, and Δ is the amount of flow sent from $p(j)$ to j. The source node, s, has label (s, ∞). Thus, for a forward arc $(p(i), \Delta)$, the label on node j will be



$(i, \min(\Delta, c_{ij} - f_{ij}))$

while for a reverse arc $(P(i), \Delta)$, the label on j will be



$(i, \min(\Delta, f_{ji}))$

The algorithm can now be described as follows.

Algorithm: Ford-Fulkerson Maximum Flow

Step 1 - initialization - $f = 0$ (or any feasible flow).

The source is labeled (s, ∞). All nodes are unscanned, and all nodes except s are unlabeled. Let $i = s$.

Step 2 - scan node i

$\forall j$ such that $(i, j) \in E$, $f_{ij} < c_{ij}$ and j is unlabeled, label j with $(i, \min(\Delta, c_{ij} - f_{ij}))$

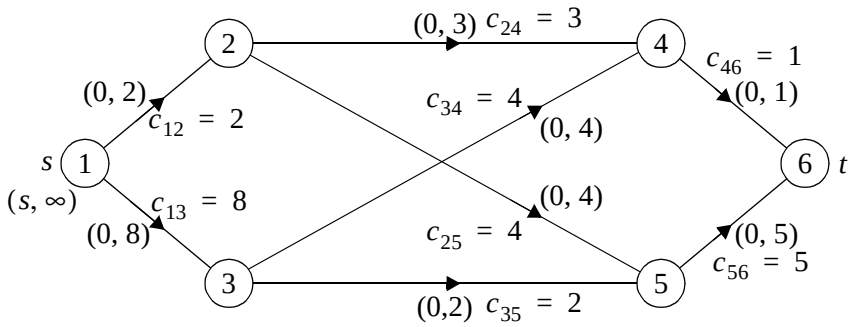
$\forall j$ such that $(j, i) \in E$, j is unlabeled, and $f_{ji} > 0$, label j with $(-i, \min(\Delta, f_{ji}))$. (Notice the negative sign to signify a backward path.)

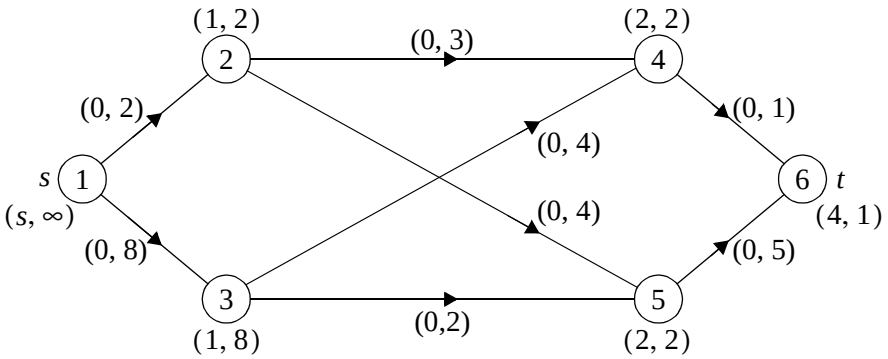
Node i is scanned.

Step 3 - if the terminus is labeled, go to step 4, otherwise choose a labeled and unscanned node i and go to step 2. If none exists, the current flow is maximum. (Nodes are scanned in the order in which they were labeled).

Step 4 - suppose t is labeled $(p(t), \Delta)$. An augmenting path has been found. Use the first element of each label to trace the path back to s. Increase the flow by Δ on all forward arcs of the path and decrease the flow by Δ on all reverse arcs. Erase all labels and return to step 1. ■

Example: Consider the following network where we start with an initial flow of $f = 0$.





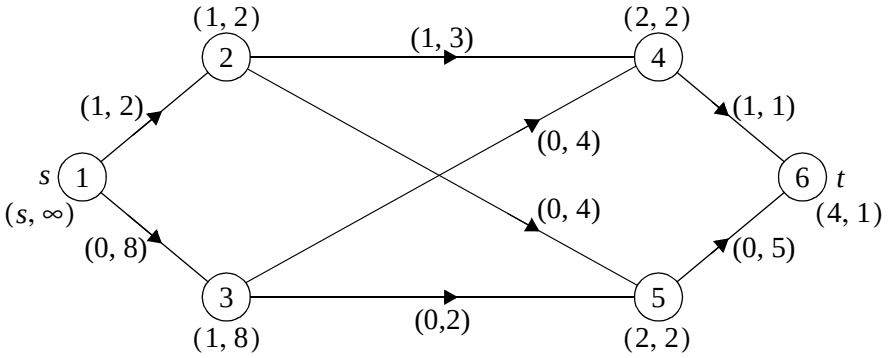
Scan 1 → labels 2 and 3

Scan 2 → labels 4 and 5

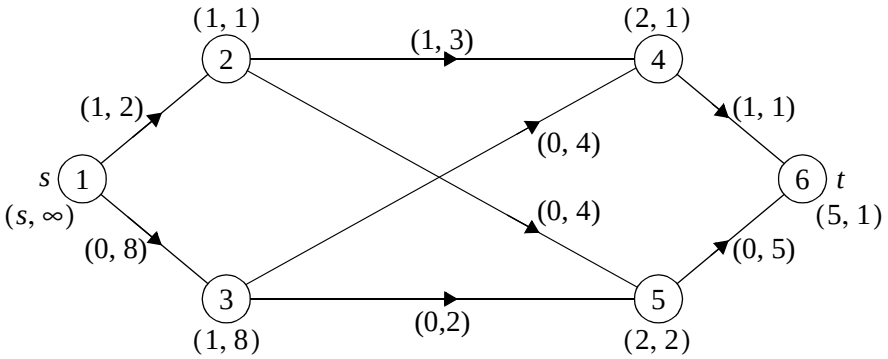
Scan 3 → no new labels

Scan 4 → label *t*

Step 4 - augmenting path is $\{6, 4, 2, 1\}$, Δ on $t = 1 \Rightarrow$ increase all forward arcs by 1.



Return back to *step* 1:



Scan 1 → labels 2 and 3

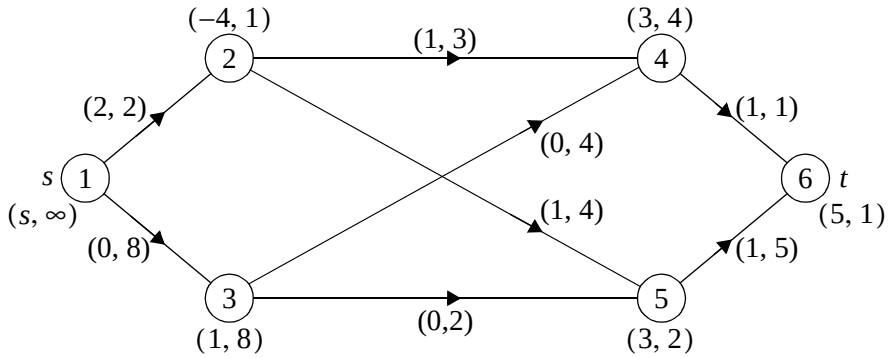
Scan 2 → labels 4 and 5

Scan 3 → no new labels

Scan 4 → arc (4, *t*) is saturated so no label!

Scan 5 → label *t*

Step 4 - augmenting path $\{5, 2, 1\}$, Δ on t is $1 \Rightarrow$ increase all forward arcs by 1.



Step 1:

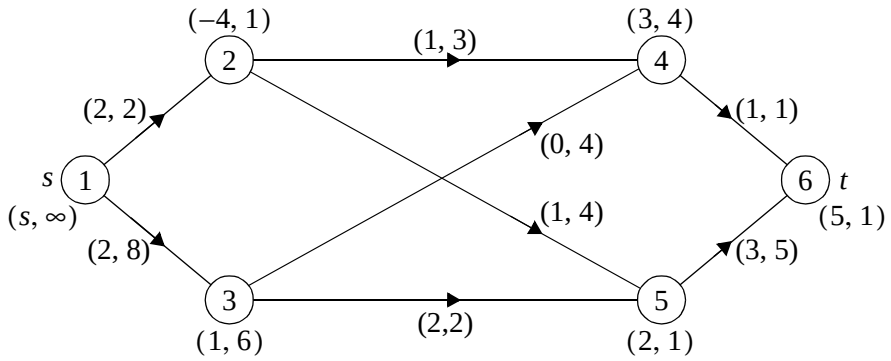
Scan 1 → label 3 since $(1, 2)$ is saturated.

Scan 3 → label 4, 5

Scan 4 → label 2 (reverse label)

Scan 5 → label *t*

Step 4 - augmentation path $\{5, 3, 1\}$, $\Delta = 2 \Rightarrow$ increase all forward arcs on path by 2.



Step 1:

Scan 1 → label 3 (since $(1, 2)$ is saturated)

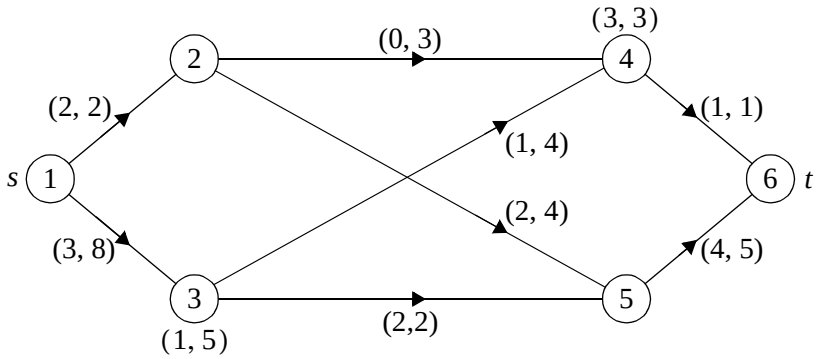
Scan 3 → label 4 (since $(3, 5)$ is saturated)

Scan 4 → label 2 (since $(4, 6)$ is saturated)

Scan 2 $\rightarrow$ label 5

Scan 5 $\rightarrow$ label t

Step 4 - augmentation path is $\{5, 2, -4, 3, 1\}$, $\Delta = 1 \Rightarrow$ increase all forward arcs on path by 1, and, since we have a reverse arc, decrease all reverse arcs on path by 1.



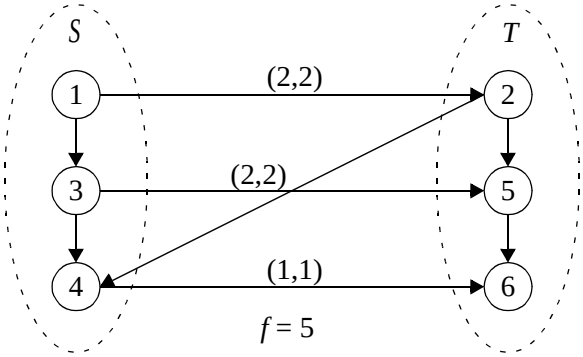
Step 1:

Scan 1 $\rightarrow$ label 3

Scan 3 $\rightarrow$ label 4

Scan 4 $\rightarrow$ can not label anymore, since (4, 6) is saturated $f_{24} = 0$.

The terminus is not labeled, so current flow is maximum. The maximum flow is found to be 5 (this is done by cutting around *s* or *t*). The cut generated by the last labeled nodes $\{1, 3, 4\}$ (forward arcs only is $\{(1, 2), (3, 5), (4, 6)\}$ which has capacity equal to maximum flow.



9.10 Shortest Path with Fixed Charges

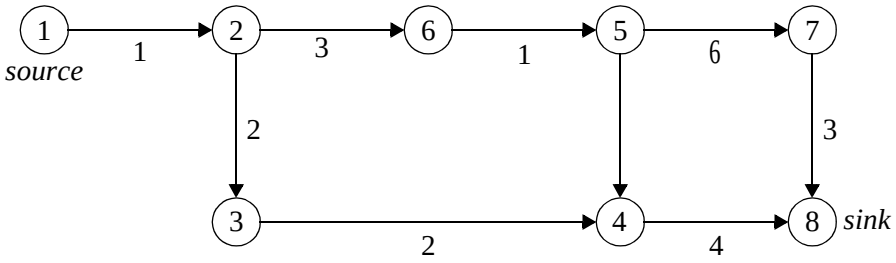
When applying Dijkstra's algorithm, it was assumed that:

"If the shortest path betwen node s to node t passes through node k, then that segment of the route from s to k is also the shortest path to k. Also, the route from k to t is the shortest between these two nodes".

Now what if the network has *turn penalties*? These are fixed charges based on the direction of entering and the direction of exiting a node. Then, in general, the above will not be true.

Example:

Consider the following network:



What is the shortest path from 1 to 8, given that there is an additional penalty of 3 for taking any turn? If we use brute force, then we can proceed by finding all the paths.

Path	Arc Sequence	Travel Cost	Turn Cost	Total Cost
P_1	(1, 2)	1	0	1
	(2, 6)	3	0	3
	(6, 5)	1	0	1
	(5, 7)	6	0	6
	(7, 8)	3	3	6 = **17**

Path	Arc Sequence	Travel Cost	Turn Cost	Total Cost
P_2	(1, 2)	1	0	1
	(2, 6)	3	0	3
	(6, 5)	1	0	1
	(5, 4)	2	3	5
	(4, 8)	4	3	7 = **17**
P_3	(1, 2)	1	0	1
	(2,3)	2	3	5
	(3, 4)	2	3	5
	(4, 8)	4	0	4 = **15**

Thus, P_3 is the shortest path from 1 - 8.

but, now notice that the shortest path from 1 to 4 is the subpath of P_2 not P_3. That is

$$(1, 2)(2, 6)(6, 5)(5, 4) \Rightarrow \quad \text{total cost} = 10.$$

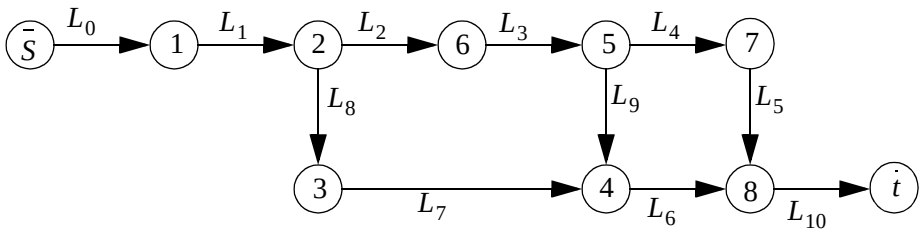
whereas:

$$(1, 2)(2, 3)(3, 4) \Rightarrow \quad \text{total cost} = 11$$

So how do we solve this problem efficiently?

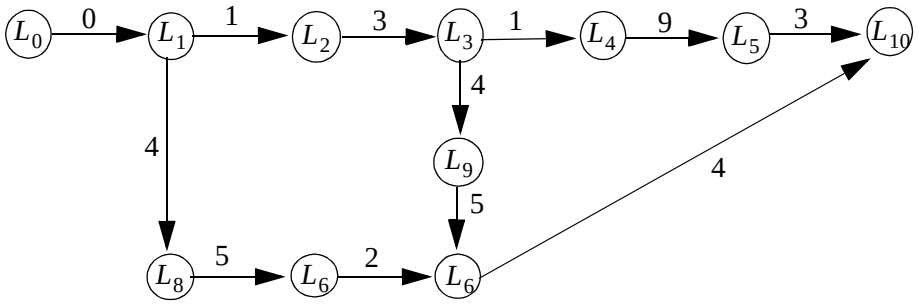
We create a new pseudo network and then apply any shortest path algorithm on the new network.

1) add a pseudo source and sink "*terminal*" node to the network:



2) and label each arc of the network

3) create a new network with one node for every labelled arc:

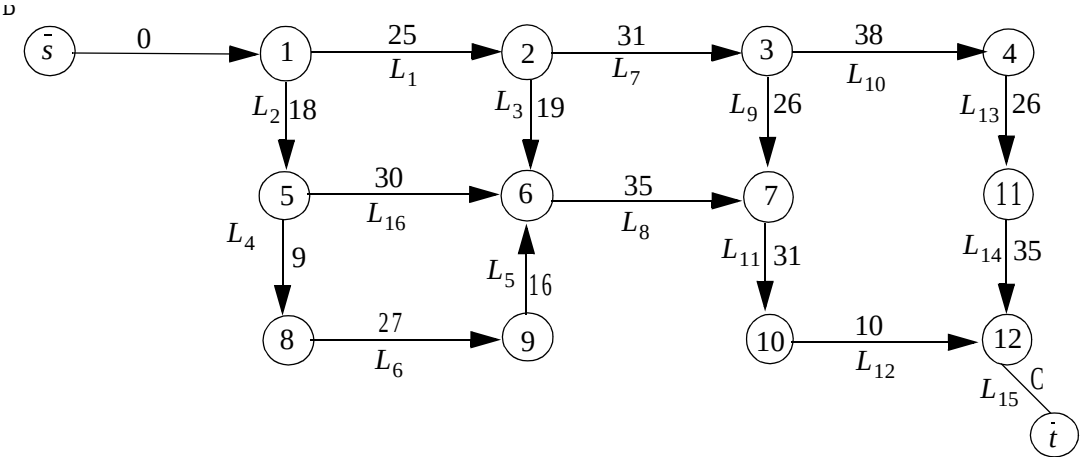


The weights cost of $(L_i, L_j) = C(L_i) + p(L_i, L_j)$ where: $C(L_i)$ - original cost of L_i, $p(L_i, L_j)$ - "*turn penalty*" $C[(\bar{s}, s)] = C[(t, \dot{t})] = 0$ by definition. Nodes $\bar{s}$ and $\dot{t}$ are never turned.

Shortest path in new network:

	Original Node	New Cost	Old Cost
$(L_0, L_1) \Rightarrow$	1	0	-
$(L_1, L_8) \Rightarrow$	2	4	1
$(L_8, L_7) \Rightarrow$	3	5	5
$(L_7, L_6) \Rightarrow$	4	2	5
$(L_6, L_{10}) \Rightarrow$	8	4	4
	Total	15	15

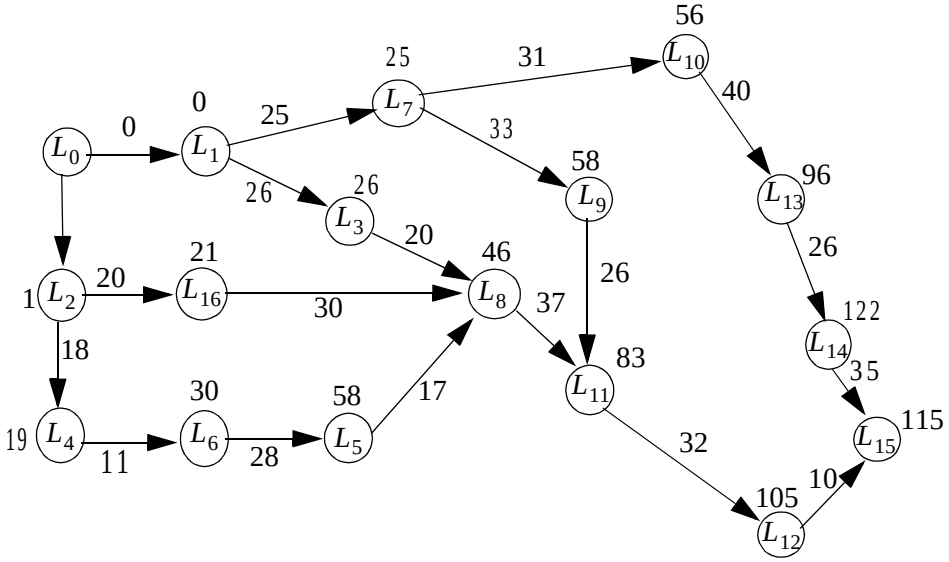
Example: conveyor system design



Points 1, 2, 6, 10, 9 implies 1 extra unit for a turn

Points 3, 5, 8, 7, 4 implies 2 extra units for a turn.

Pseudo Network:



Therefore, the shortest path is $L_1, L_3, L_8, L_{11}, L_{12}$ or nodes: 1, 2, 6, 7, 10, 12.

minimum cost = 115 units