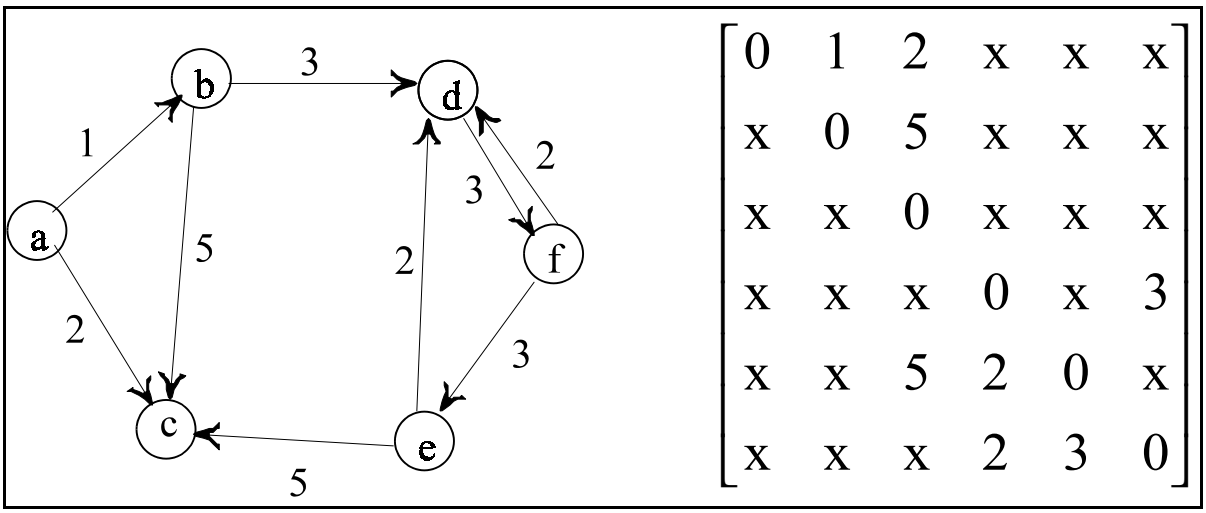
9

Shortest Paths

A fundamental problem in graphs is finding the shortest path from vertex A to vertex B. Fortunately there are several simple (and efficient algorithms for doing this). We will look at the three best of these: Dijkstra's algorithm, Floyd's algorithm, and the Bellman-Ford algorithm. First we need to discuss different types of shortest path problems and various conventions that we will use in solving them.

Representation of Graphs

The representation of the graph in the computer can be done in several ways. Usually the most convenient approach is the one used here. We will represent each graph as an array.



$$\begin{bmatrix} 0 & 1 & 2 & x & x & x \\ x & 0 & 5 & x & x & x \\ x & x & 0 & x & x & x \\ x & x & x & 0 & x & 3 \\ x & x & 5 & 2 & 0 & x \\ x & x & x & 2 & 3 & 0 \end{bmatrix}$$

Figure 1 A Graph with Matrix Representation

This is illustrated in **Figure 1**. If the matrix is M and the vertices are numbered in alphabetical order, then we have M(5,3) = 5 since there is an arc from the fifth node (e) to the third node (c) of distance 5. M(3,5) is marked x, since there is no direct arc from c to e. Instead of x, one

might use ∞. The advantage of ∞ is that if it turns out to be impossible to go from c to e, then the distance is effectively ∞. However, there is a question of how to represent this in a computer. One approach is to actually use some symbol like "x." If the graph has no negative distances than -1 works. Another approach is to use a large number, say 1,000,000 when that number is larger than the length of any path that will be generated. That way, when the shortest distances have been computed, if the shortest distance from a to b turns out to be 1,000,000 or greater, this means that in fact it is impossible to go from a to b. If this seems a little puzzling, come back to it after you have studied the algorithms in this chapter.

Negative Arcs

The "distance" given on arc corresponds to something that we want to minimize such as distance itself, costs, weight or any penalty in general. A negative arc from node a to node b might mean that if you go from node a to node b that you are paid. There are two problems with this. First, some algorithms will not work with negative arcs. In this chapter that is true of Dijkstra's algorithm. Secondly, if there is a negative circuit, then there is no minimal distance between some pairs of points. Consider the circuit in **Figure 2**. In leaving node b and returning to node b, the distance is -1, hence this graph has a negative circuit. One consequence is that there is no minimum

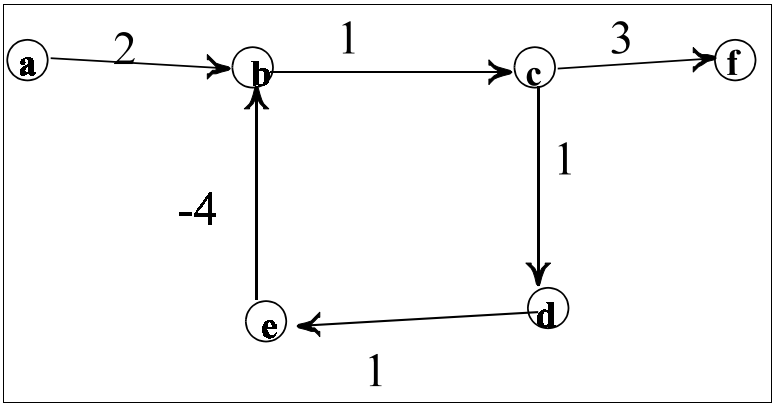


Figure 2 A Graph with a Negative Circuit

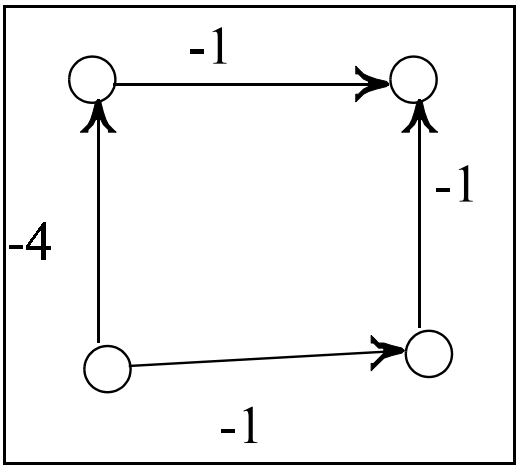


Figure 3 A Graph that is not a Negative Circuit

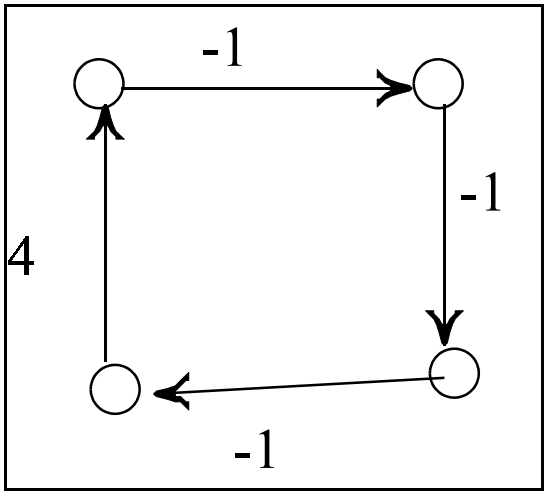
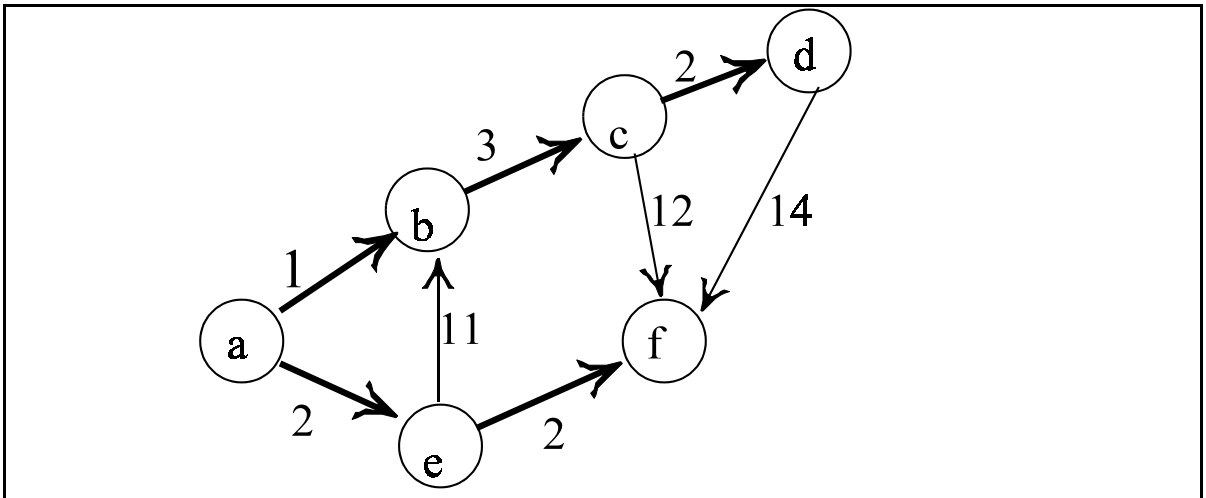


Figure 4 Another Graph that is not a Negative Circuit

distance from a to f. If on leaving a, we cycle at b, we can cycle as often as we wish, each time adding -1 to the total distance. Hence the distance from a to b can be a small as we wish. None of the algorithms presented here will work in the case of a negative circuit. However, a negative circuit does not correspond to anything we would expect to model. In general when we receive something free we still pay more than we receive. Frequent flier miles are a good example. To receive frequent flier miles you must first pay to fly some minimal number of miles. The airline is not going to provide a deal where its total cost is negative (not if it is to survive). The graph in **Figure 3** is okay. Although it gives negative arcs and negative paths, there is no negative circuit. Similarly the graph in **Figure 4** is not a negative circuit. Although three of its four edges are negative, the total distance is positive (+1).

Shortest Paths

It makes sense that if we are going to find the shortest distance from node a to node b, that we should find the shortest path. The key to finding paths is knowing how to describe them.



Vertex	a	b	c	d	e	f
Distance	0	1	4	6	2	4
Prior	/	a	b	c	a	e

Figure 5 A Graph with Embedded Shortest Path Tree and Corresponding Table

The shortest paths from node a to all other nodes is given by showing the last node visited before each node. An illustrative example is given in **Figure 5**. In the graph itself, the shortest paths from node a are indicated by the darker edges. These form a *shortest path tree*. An exercise will be to show that the shortest paths from a node to all other nodes will always be representable as a tree. The second row of the table shows the shortest distances from node a to the vertex named in the same column and first row. For example, the 6 under f, indicates that the shortest path from a to f is of length 6. Whereas we can represent this visually as a tree which shows clearly that the shortest path from a to f goes through b and c in that order, the computer algorithms must store this information in a table. That works as follows: To find the shortest path from a to d (by using the table) we look at the third row of the table, the row marked "prior." Under the d column we find (in the third row) the node c. This means that the prior node visited before d (in the shortest path from a) is c. We look in the c column and we find that the prior node visited before c is b. Similarly we find that the prior node to b is a itself. The shortest route from a to b is the direct arc from a to b and the shortest path from a to d is a-b-c-d.

☐ **Exercise 1** Show that in a given graph that the shortest paths from some a specified node a can be represented by a tree. (We assume that the graph has no circuit of negative length.)

Important Comments About the Shortest Path Algorithms

Efficiency of algorithms is a topic outside the domain of this book. However, all three of the algorithms here are relatively efficient. If we want to find the shortest path from one node to all other nodes, then Dijkstra's algorithm is usually more efficient that the Bellman-Ford algorithm. However, the Bellman-Ford algorithm is even more simple than Dijkstra's algorithm. Also, the Bellman-Ford algorithm can handle negative arcs whereas Dijkstra's algorithm can't. If on the other hand we want the shortest paths from all nodes (to all nodes) we can run Dijkstra's algorithm on each node (as a source) or we can do the same thing with the Bellman-Ford algorithm or we can use Floyd's algorithm. In this case the efficiency of Floyd's algorithm is similar to running Dijkstra's algorithm for each node. Lastly, though Floyd's algorithm finds shortest paths from each node, there is nothing keeping you from using it to find the shortest paths from any particular node other than the fact that either of the other algorithms is usually faster for that purpose.

Dijkstra's Algorithm

Almost certainly Dijkstra's algorithm is the most used algorithm for finding shortest paths. It is the algorithm generally used for finding routes through the Internet.

Conventions for Dijkstra's algorithm

Dijkstra's algorithm finds all shortest paths from a source vertex, S, to all other vertices. There are no negative arcs allowed. The graph is represented by the array M, where M(i, j) is the direct distance from node i to node j. The set of vertices of the graph is denoted by V. Each vertex, v, will have a *state* value denoted *State*(v). There are two states, T, for *temporary* and P, for *permanent.* Also associated with each vertex, v, are

two other functions: *Dist*(v) is the distance to v from S and *Prior*(v) is the vertex visited just prior to v in the shortest path from S. The idea of the algorithm is reasonably simple. The algorithm proceeds recursively through stages. At each stage a new vertex is marked as *permanent* in its distance from S. Then the distances to the vertices marked *temporary* are updated through the latest permanent vertex. That is, if it is possible to shorten the current distance to a vertex through the most recently permanent vertex, then its distance (and prior) are updated. One this is done, a new stage begins and the temporary vertex with least distance from S is marked permanent. Once all vertices are marked as permanent then the algorithm is terminated.

Dijkstra's Algorithm (itself)

Input: A set of vertices V with a particular vertex S. The distances between pairs ov vertices is given by a matrix M.

1. **For** each $v \in V$, set State(v)$\leftarrow$T, Dist(v)$\leftarrow\infty$, Prior(v)$\leftarrow\varnothing$. [Initialization: each vertex is initialized as temporary, infinitely far from S and with no prior node.]
2. State(S)$\leftarrow$P, Dist(S)$\leftarrow$0. [The source vertex is initialized as zero distance from itself and as labeled permanent (since the distance cannot be improved). Note that the prior is still $\varnothing$.]
3. **Repeat**

    Set Temp_Dist$\leftarrow\infty$. [We are going to find the temporary vertex with the minimum distance.]
    For each $v \in V$
        If State(v)=T then Temp_Dist$\leftarrow$Min(Temp_Dist, Dist(v))
    Choose a vertex, u, such that State(u)=T and Dist(u)=Temp_Dist. [This is the vertex mentioned previously, and if there is a tie, pick any one of the minimal vertices.]
    Set State(U)$\leftarrow$P. [Next we will update the information on our temporary vertices.]
    For each $v \in V$
        If State(v)=T
            If Dist(v) > Dist(u) + M(u, v)
                Set Dist(v) $\leftarrow$ Dist(u) + M(u, v) and Prior(v)$\leftarrow$u.

The algorithm terminates when all vertices are labeled *P*. It is important to recognize that some of these vertices may still have Dist = ∞and Prior = ∅. These are the vertices that cannot be reached from S. A key part of the algorithm is the statement:

If Dist(v) > Dist(u) + M(u, v) then Set Dist(v) ← Dist(u) + M(u, v) and Prior(v)←u.

This statement occurs in each of the algorithms (Dijkstra's, Bellman-Ford, and Floyd's). It is most easily understood in the Bellman-Ford algorithm.

The following series of graphs illustrate Dijkstra's algorithm:

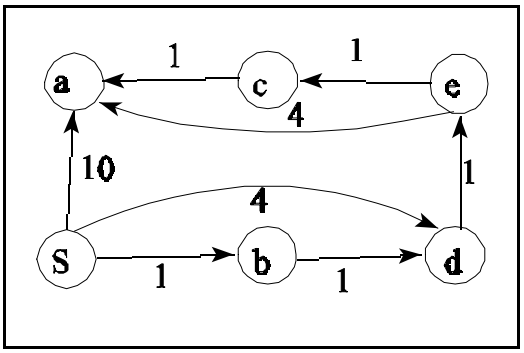


Figure 6 A Graph with Distances

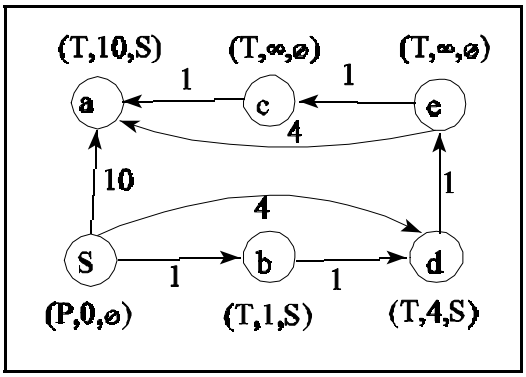


Figure 7 First Step of Dijkstra's Algorithm

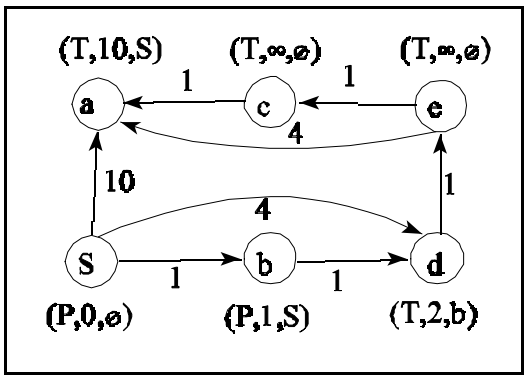


Figure 8 First Iteration of the Repeat Loop

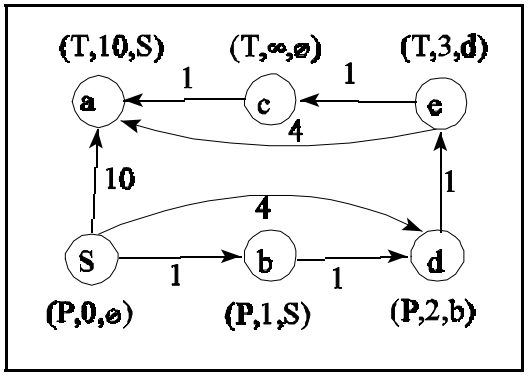


Figure 9 Second Iteration of the Repeat Loop

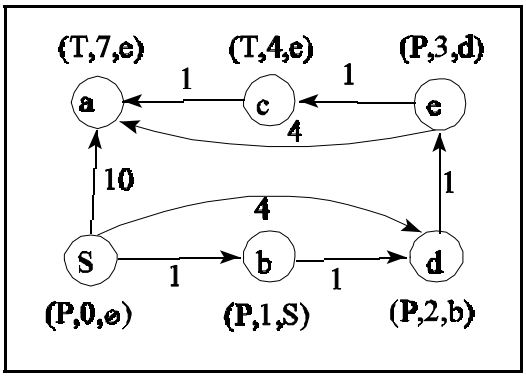


Figure 10 Third Iteration of the Repeat Loop

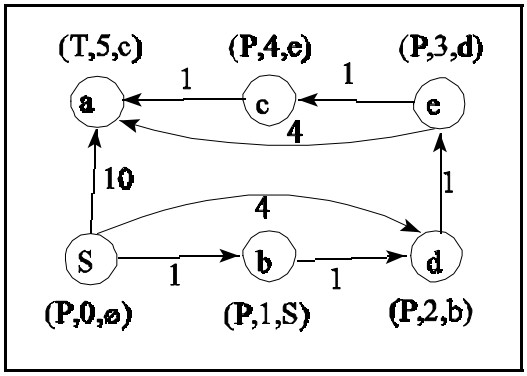


Figure 11 Fourth Iteration of the Repeat Loop

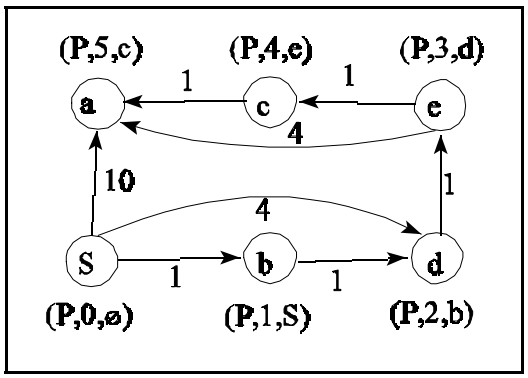


Figure 12 Final Iteration of the Repeat Loop

☐ **Exercise 2** Why does Dijkstra's algorithm work?

☐ **Exercise 3** Solve the following graph for shortest distances from S.

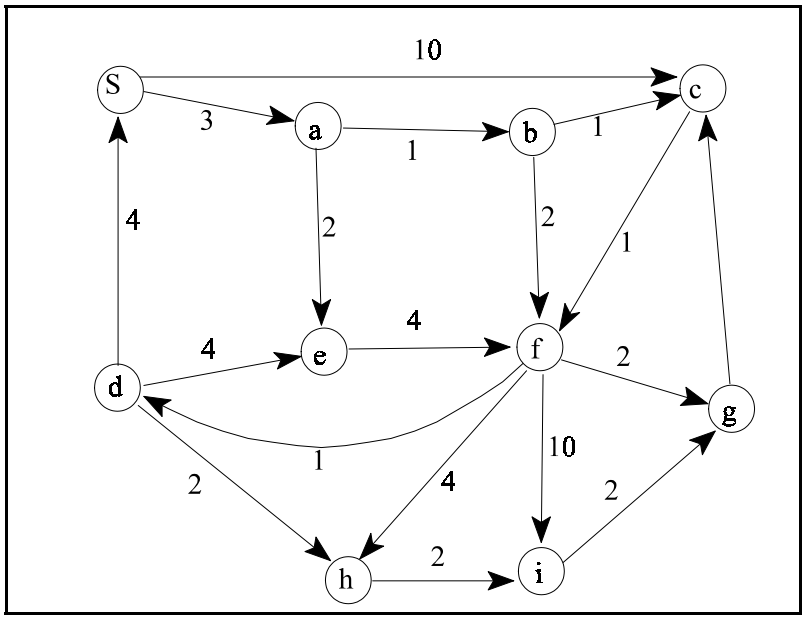


Figure 13 Find Shortest Distances from S

The Bellman-Ford Algorithm

The Bellman-Ford algorithm for shortest paths is almost completely intuitive. The basis for it is as follows: Given a graph with n vertices (and as always with no negative circuits) the most arcs there can be in a shortest path is n-1. This is easily proven by induction. Should a shortest path have more than n-1 then it must have a negative circuit. Similarly, a graph with a negative circuit will have a shortest path of n or more arcs. All of these observations lead to the Bellman-Ford algorithm

Conventions for the Bellman-Ford Algorithm

The conventions here are nearly the same as for Dijkstra's algorithm. One difference is that the Bellman-Ford algorithm can handle arcs of negative length. I am writing the algorithm here such that the input is the same as for Dijkstra's algorithm. However, the algorithm is more efficient if it is written using a list of the arcs.

The Bellman-Ford Algorithm (itself)

Input: A set of vertices V with a particular vertex S. The distances between pairs ov vertices is given by a matrix M.

1. **For** each $v \in V$, set State(v)←T, Dist(v)←∞, Prior(v)←∅. The number of vertices is denoted n. [Initialization: each vertex is initialized as temporary, infinitely far from S and with no prior node.]
2. State(S)←P, Dist(S)←0. [The source vertex is initialized as zero distance from itself and as labeled permanent (since the distance cannot be improved). Note that the prior is still ∅.]
3. Condition←True; Counter←0
4. **While** Condition
   Counter←counter + 1
   If counter = n + 1 Print "There is a negative circuit" and Stop.
   Condition←False
   For $u \in V$
   For $v \in V$ $(v \neq u)$
   If Dist(v) > Dist(u) + M(u, v)
   Set Dist(v) ← Dist(u) + M(u, v) and Prior(v)←u
   Set Condition←True

The main body of the algorithm is block 4. It looks at every arc in the graph and tries to use it to improve some path. If there is no improvement then the variable *Condition* stays false and the algorithm terminates. If on the other hand there are improvements through n iterations of block 4, then there must be a negative circuit, and the variable counter reaches the value n + 1. In that case the algorithm simply prints "There is a negative circuit" and terminates. Hence there are two different termination criteria. Notice that only the second termination criteria is explicit.

A Key Observation on the Bellman-Ford Algorithm

The algorithm as stated here wastes time in each iteration of the while-loop by looking at each pair of vertices, u and v, for an arc. There is an arc from u to v if M(u, v) is finite. If instead of writing

For $u \in V$

For $v \in V$ $(v \neq u)$

we write

For each $a \in A$

where A is the set of all arcs of the graph, then there is no time wasted looking at pairs of vertices that have no arcs between them.

Floyd's Algorithm

Floyd's algorithm is efficient, easy to program and handles negative arcs. It finds shortest paths from each vertex. It is covered in this section because of these fine qualities. It is much less intuitive than Dijkstra's algorithm or the Bellman-Ford algorithm. It is harder to prove (that it works) than the other two algorithms. No argument will be given here. Note, that most people when they first learn the algorithm are surprised that it does work.

Conventions for Floyd's Algorithm

Floyd's algorithm finds all shortest paths from all vertices to all other vertices. Negative arcs are allowed. The graph is represented by the array M, where M(i, j) is the direct distance from node i to node j. The set of vertices of the graph is denoted by V. Associated with each pair of vertices, u and v, are two functions: *Dist*(u, v) is the distance u to v and *Prior*(u, v) is the vertex visited just prior to v in the shortest path from S.

Floyd's Algorithm (itself)

1. **For** each $u \in V$
   For $v \in V$
   Set Dist(u, v)←M(u, v), Prior(u, v)←u if M(u, v) < ∞ otherwise Prior(u, v)←∅. [This initializes the distances to the direct distances of the graph, and it initializes the prior accordingly.]
2. **For** $w \in V$
   For $u \in V$
   For $v \in V$
   If Dist(u, v) < Dist(u, w) + Dist(w, v)
   Set Dist(u, v) ← Dist(u, w) + Dist(w, v)
   Set Prior(u, v)←w
3. **For** $u \in V$ **If** Dist(u, u) < 0 then Print "There is a negative circuit!"

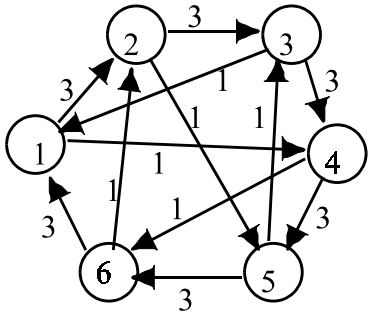
Again, I am not going to justify Floyd's algorithm. However, it is appropriate to describe how it works, if not why. The heart of the algorithm is block 2. If the graph has n vertices, then block 2 consists of precisely n iterations of n iterations of n iterations for a total of n^3 iterations. The first or "outer" iterations consists of taking each vertex in turn and checking if using it as an intermediate vertex whether it can shorten the distance between each pair of vertices in the graph. There are then n iterations of the matrix Dist. Dist starts out as the matrix M and is

updated n times. Often Floyd's algorithm is implemented using two version of Dist and then the principal statement becomes:

If Dist(u, v) < Dist(u, w) + Dist(w, v) Set NewDist(u, v) ← Dist(u, w) + Dist(w, v)

In this case each time w changes value (in the outer loop) Dist is replaced by NewDist. However, it is not necessary to do that.

An example of Floyd's algorithm is given below. In this case the matrix $Dist_i$ is the matrix Dist after the vertex i has been considered as an intermediate vertex; that is $Dist_i$ is the matrix Dist after the i iteration of the outer loop. Similarly the matrices $Prior_i$ are the corresponding Prior matrices. The matrix $Dist_0$ is the initial matrix with no intermediate vertices and is the same matrix as M. Likewise, $Prior_0$ is the corresponding Prior matrix.



$$Dist_0 = M = \begin{pmatrix} 0 & 3 & x & 1 & x & x \\ x & 0 & 3 & x & 1 & x \\ 1 & x & 0 & 3 & x & x \\ x & x & x & 0 & 3 & 1 \\ x & x & 1 & x & 0 & 3 \\ 3 & 1 & x & x & x & 0 \end{pmatrix} \quad \Pr ior_0 = \begin{pmatrix} 1 & 1 & x & 1 & x & x \\ x & 2 & 2 & x & 2 & x \\ 3 & x & 3 & 3 & x & x \\ x & x & x & 4 & 4 & 4 \\ x & x & 5 & x & 5 & 5 \\ 6 & 6 & x & x & x & 6 \end{pmatrix}$$

$$Dist_1 = \begin{pmatrix} 0 & 3 & x & 1 & x & x \\ x & 0 & 3 & x & 1 & x \\ 1 & 4 & 0 & 2 & x & x \\ x & x & x & 0 & 3 & 1 \\ x & x & 1 & x & 0 & 3 \\ 3 & 1 & x & 4 & x & 0 \end{pmatrix} \quad \Pr ior_1 = \begin{pmatrix} 1 & 1 & x & 1 & x & x \\ x & 2 & 2 & x & 2 & x \\ 3 & 1 & 3 & 1 & x & x \\ x & x & x & 4 & 4 & 4 \\ x & x & 5 & x & 5 & 5 \\ 6 & 6 & x & 1 & x & 6 \end{pmatrix} \quad Dist_2 = \begin{pmatrix} 0 & 3 & 6 & 1 & 4 & x \\ x & 0 & 3 & x & 1 & x \\ 1 & 4 & 0 & 2 & 5 & x \\ x & x & x & 0 & 3 & 1 \\ x & x & 1 & x & 0 & 3 \\ 3 & 1 & 4 & 4 & 2 & 0 \end{pmatrix} \quad \Pr ior_2 = \begin{pmatrix} 1 & 1 & 2 & 1 & 2 & x \\ x & 2 & 2 & x & 2 & x \\ 3 & 1 & 3 & 1 & 2 & x \\ x & x & x & 4 & 4 & 4 \\ x & x & 5 & x & 5 & 5 \\ 6 & 6 & 2 & 1 & 2 & 6 \end{pmatrix}$$

$$Dist_3 = \begin{pmatrix} 0 & 3 & 6 & 1 & 4 & x \\ 4 & 0 & 3 & 5 & 1 & x \\ 1 & 4 & 0 & 2 & 5 & x \\ x & x & x & 0 & 3 & 1 \\ 2 & 5 & 1 & 3 & 0 & 3 \\ 3 & 1 & 4 & 4 & 2 & 0 \end{pmatrix} \quad \Pr ior_3 = \begin{pmatrix} 1 & 1 & 2 & 1 & 2 & x \\ 3 & 2 & 2 & 3 & 2 & x \\ 3 & 1 & 3 & 1 & 2 & x \\ x & x & x & 4 & 4 & 4 \\ 3 & 3 & 5 & 3 & 5 & 5 \\ 6 & 6 & 2 & 1 & 2 & 6 \end{pmatrix} \quad Dist_4 = \begin{pmatrix} 0 & 3 & 6 & 1 & 4 & 2 \\ 4 & 0 & 3 & 5 & 1 & 6 \\ 1 & 4 & 0 & 2 & 5 & 3 \\ x & x & x & 0 & 3 & 1 \\ 2 & 5 & 1 & 3 & 0 & 3 \\ 3 & 1 & 4 & 4 & 2 & 0 \end{pmatrix} \quad \Pr ior_4 = \begin{pmatrix} 1 & 1 & 2 & 1 & 2 & 4 \\ 3 & 2 & 2 & 3 & 2 & 4 \\ 3 & 1 & 3 & 1 & 2 & 4 \\ x & x & x & 4 & 4 & 4 \\ 3 & 3 & 5 & 3 & 5 & 5 \\ 6 & 6 & 2 & 1 & 2 & 6 \end{pmatrix}$$

$$Dist_5 = \begin{pmatrix} 0 & 3 & 5 & 1 & 4 & 2 \\ 3 & 0 & 2 & 4 & 1 & 4 \\ 1 & 4 & 0 & 2 & 5 & 3 \\ 6 & 8 & 4 & 0 & 3 & 1 \\ 2 & 5 & 1 & 3 & 0 & 3 \\ 3 & 1 & 3 & 4 & 2 & 0 \end{pmatrix} \quad \Pr ior_5 = \begin{pmatrix} 1 & 1 & 5 & 1 & 2 & 4 \\ 5 & 2 & 5 & 5 & 2 & 5 \\ 3 & 1 & 3 & 1 & 2 & 4 \\ 5 & 5 & 5 & 4 & 4 & 4 \\ 3 & 3 & 5 & 3 & 5 & 5 \\ 6 & 6 & 5 & 1 & 2 & 6 \end{pmatrix} \quad Dist_6 = \begin{pmatrix} 0 & 3 & 5 & 1 & 4 & 2 \\ 3 & 0 & 2 & 4 & 1 & 4 \\ 1 & 4 & 0 & 2 & 5 & 3 \\ 4 & 2 & 4 & 0 & 3 & 1 \\ 2 & 4 & 1 & 3 & 0 & 3 \\ 3 & 1 & 3 & 4 & 2 & 0 \end{pmatrix} \quad \Pr ior_6 = \begin{pmatrix} 1 & 1 & 5 & 1 & 2 & 4 \\ 5 & 2 & 5 & 5 & 2 & 5 \\ 3 & 1 & 3 & 1 & 2 & 4 \\ 6 & 6 & 5 & 4 & 4 & 4 \\ 3 & 6 & 5 & 3 & 5 & 5 \\ 6 & 6 & 5 & 1 & 2 & 6 \end{pmatrix}$$

Figure 14 An Implementation of Floyd's Algorithm

The solution to the graph is the last pair of matrices: $Dist_6$ and $Prior_6$. They indicate for example that the shortest path from node 3 to node 5 is of length 5. To find the path, we look at $Prior_6(3,5)$ and see the entry 2. This indicates that on leaving 3, the last node visited before node 5, is node 2. On looking at $Prior_6(3,2)$ we see the entry 1. On looking at $Prior_6(3,1)$ we see the entry 3. This indicates that the shortest path from 3 to 5 is the sequence 3-1-2-5. Note that it is not necessary that there be a path between every pair of vertices as happened in the above example. Also, many of these shortest paths are not unique. That is, the final matrix Dist will be unique, but typically the matrix Prior, which gives the paths themselves, is not unique. When a shortest path shown in Prior is not unique, the path shown is determined simply by the order that the vertices are addressed in the implementation of the algorithm.

☐ **Exercise 4** Use Floyd's algorithm to find the shortest distances between all pairs of vertices in the following "graph:"

$$\begin{pmatrix} 0 & 1 & 2 & x & x & x \\ 2 & 0 & x & x & x & 1 \\ x & x & 0 & 2 & 1 & x \\ 1 & x & x & 0 & x & 2 \\ x & 2 & x & 1 & 0 & x \\ x & x & 1 & x & 2 & 0 \end{pmatrix}$$

1. Several points need to be made before we answer this question. First there may be more than one shortest path from a node to some other node. Secondly, there may not exist a path from a given node to some other node. If we look at the shortest paths from node x to the nodes that it can reach then these can be represented by a tree as in the example in the text. The reason is simple. If the graph of shortest paths is not a tree, then it has a circuit. If the circuit is directed from a node to itself, either it is redundant or it has a negative length. However, we are not considering problems with negative circuits. Otherwise if there is a circuit that is not directed then two arcs enter the same node. This represents two paths from the source node. One arc is redundant and can be removed.

2. As phrased this question does not require a formal proof. The answer is merely the outline of a possible (inductive) proof. At each step of the algorithm a new node is labeled *P*, and therefore the algorithm will always terminate (as is required of all algorithms). Thus to prove the algorithm does what it is supposed to do, we need to show that whenever a node is labeled *P* its distance is correct. In the first step the node S is correctly labeled since the distance to it from S must be 0. Assume for the moment that when we label a node with *P* its distance is minimal. If that is so for the first k nodes labeled *P*, it must be true for the k+1'th node labeled P. Since distance to the new node is minimal with respect to the first k nodes labeled *P* and going through any of the nodes labeled *T* would give a longer distance than the present distance (because there are no negative lengths). Therefore the new node has its distance at the optimal quantity. The assumption above is clearly correct at step 1. Since it remains correct for each subsequent node labeled *P*, we are done.

3. Shortest distances are (S,0) (a,3) (b,4) (c,5) (d,7) (e,11) (f,6) (g,8) (h,9) (i,11).

4. $$\begin{pmatrix} 0 & 1 & 2 & 4 & 3 & 2 \\ 2 & 0 & 2 & 4 & 3 & 1 \\ 3 & 3 & 0 & 2 & 1 & 4 \\ 1 & 2 & 3 & 0 & 4 & 2 \\ 2 & 2 & 4 & 1 & 0 & 3 \\ 4 & 4 & 1 & 3 & 2 & 0 \end{pmatrix}$$