Standard Problems and Algorithms on Weighted Graphs

We now discuss two problem commonly associated with weighted graphs in which the edge weights represent costs or distances. These problems are the **minimum spanning tree (MST)** and **distances between vertices**. Each of these two problems has two important algorithms that can be proven to solve it.

The MST (minimum spanning tree) problem is solved by either
- Prim's Minimum Spanning Tree algorithm, or
- Kruskal's Minimum Spanning Tree algorithm.

The distance problems in a graph are solved by either
- Dijkstra's Single-Source Shortest Path Algorithm, or
- Floyd's Algorithm.

We discuss the MST problem first, beginning with a definition of a spanning tree.

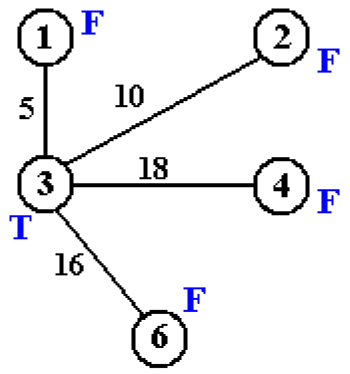
Definition: A ***spanning tree*** for a connected undirected graph, G = (V, E), is a subgraph of G that is an undirected tree and contains all the vertices of G. In a weighted graph, a minimum spanning tree is a spanning tree for which the sum of the edge weights is not greater than the sum of the edge weights of any other spanning tree. Note that any spanning tree of an unweighted undirected graph is a minimum spanning tree, as every spanning tree for an unweighted graph on N nodes must have total edge weight (N – 1), for N – 1 edges, each of weight 1.

Prim's Algorithm for Minimum Spanning Trees

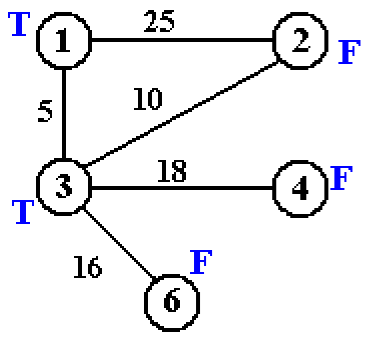
This algorithm is based on labeling all vertices with one of three labels – unseen, tree, or fringe. These labels assist in determining which vertices are to be processed next.

```
Algorithm Prim
    Initialize all vertices with the marking unseen.
    Select an arbitrary vertex s and mark it as tree. This is the beginning of the MST.
    Label all vertices in N(s), the open neighborhood of s, as fringe.
    While there are any vertices still labeled as fringe and the MST is not complete
        Select an edge of minimum weight between a tree node t and a fringe node u.
        Reclassify node u as a tree vertex and add edge (t, u) to the spanning tree.
        Reclassify all unseen vertices in N(u) as fringe.
```

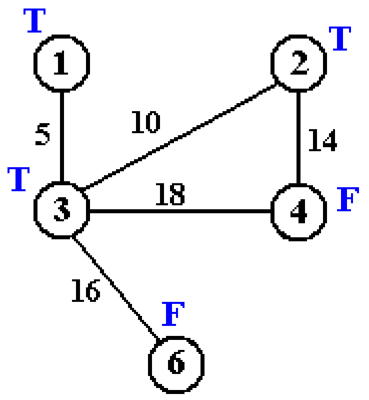
We illustrate Prim's algorithm on the weighted graph shown in Figure 21, above. In this illustration, the only vertices to be shown at each stage will be tree vertices (labeled T) and fringe vertices (labeled F). Unseen vertices will not be shown. We begin at vertex 3.



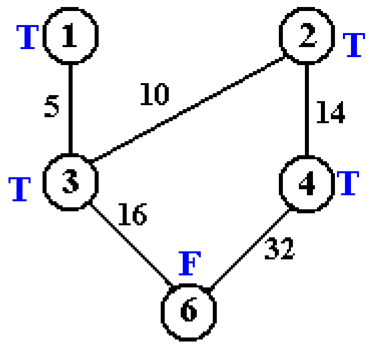
The choice of vertex 3 as the starting vertex *s* was basically arbitrary. This author did not want to begin at vertex 1. Now N(3) = {1, 2, 4, 6}. Vertex 1 is closest to vertex 3, so it is labeled as a tree node and goes into the spanning tree. Let T be the growing spanning tree of G. We say that |T| = 5 (so far). Adding vertex 1 to the tree brings edge (1, 2) into view. Note that we do not as yet see edge (2, 4) because we are tracking only edges with at least one end vertex in the spanning tree.



Vertex 2 is closest to the tree, being at a distance of 10 from vertex 3, so we label it as tree and add it. We now drop the edge (1, 2) as being between two tree vertices and not eligible for use. |T| = 15.

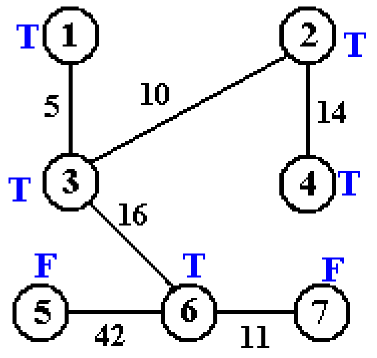


Note that the addition of vertex 2 to the tree does not add any vertices to the fringe set, still {4, 6}. Vertex 4 is closer to the tree than vertex 6, being at a distance of 14 from vertex 2, as opposed to the distance of 16 for vertex 6. We add vertex 4 to the tree and now drop edge (3, 4) as being between two tree vertices. |T| = 29.



Again, note that adding this vertex has not added any new vertices to the fringe, although it has added the new edge
(4, 6). Since vertex 6 is the only vertex in the fringe, it is obviously the closest; the only task is to pick the shorter of the two edges and use it to build the tree. We pick edge (3, 6), with a weight of 16, adding it to the tree to get |T| = 45.

Once this is done, the remainder of the process should be obvious. We have added two vertices to the fringe. Vertex 7 is closer to the tree than vertex 5, so it is added first. Then vertex 5 is finally added. The total weight of the spanning tree is 98.
This is the minimum spanning tree for this graph. If the algorithm has been applied correctly, it is possible to prove that no spanning tree of lesser total weight exists.



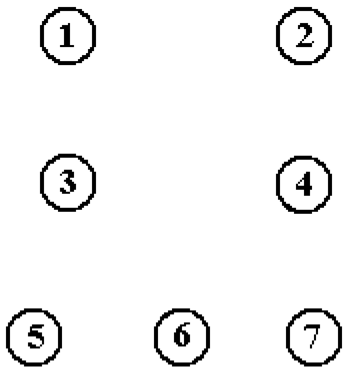
Kruskal's Algorithm for a Minimum Spanning Tree

We now discuss another algorithm for determining a minimum spanning tree of a connected weighted graph G. This is named after Joseph Kruskal who discovered the algorithm in 1956 while he was a second-year graduate student. One has to admit that it is not a small feat for a graduate student to discover an algorithm important enough to be named after him and still be taught almost 50 years after its discovery. For those students with dreams of algorithmic glory in their eyes, we mention that there are two processes involved in originating an algorithm – first discovering the algorithm and then proving that it works as advertised.

Unlike Prim's algorithm, which continuously grows a tree until it is a spanning tree, Kruskal's just adds candidate edges to an edge set until the edge set becomes the spanning tree. Here is the algorithm.

```
Algorithm Kruskal
    Input: A weighted connected graph $G = (V, E)$
    Sort the edge set E in non-decreasing order of edge weights>
    Label the edges in this order as $e_1, e_2, \ldots, e_m$, where $m = |E|$
    We now have $w(e_1) \le w(e_2) \le \ldots \le w(e_m)$.
    Set $E_T = \Phi$, the empty set.
    Set $k = 0$
    While $|E_T| < |V| - 1$
        $k = k + 1$
        If $E_T \cup \{e_k\}$ is acyclic, then $E_T = E_T \cup \{e_k\}$
    return $E_T$
```

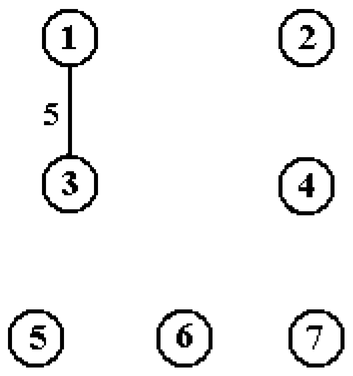
Once again, we use the graph of figure 21 to illustrate an MST algorithm. We begin with the graph having no edges and a list of the edges, sorted by non-decreasing weight.

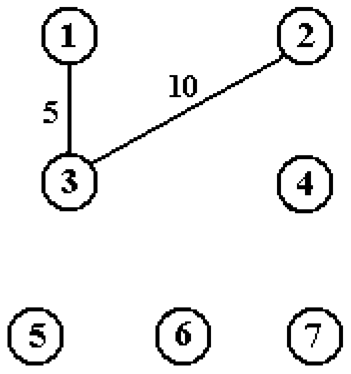


Here is the list of vertices, sorted by weight

Weight 5:	(1, 3)
Weight 10:	(2, 3)
Weight 11:	(6, 7)
Weight 14:	(2, 4)
Weight 16:	(3, 6)
Weight 18:	(3, 4)
Weight 25:	(1, 2)
Weight 32:	(4, 6)
Weight 42:	(5, 6)

We start by adding the least cost edge to the collection. Since it is the first edge into the edge set, we cannot form a cycle. The state of the edge set after this step is shown at right.



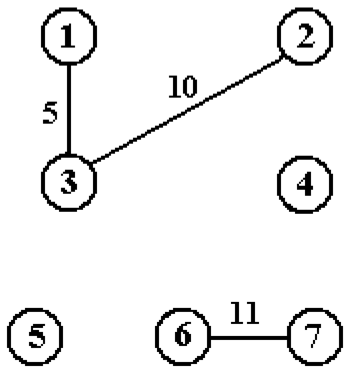


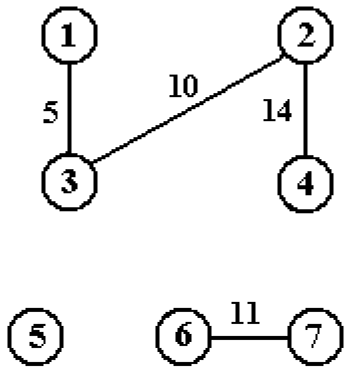
The edge next in the sort order is (2, 3). This does not cause a cycle, so it is added to the collection of edges.

We now have E_T = {(1, 3), (2, 3)}.

The next edge is sort order is (6, 7). Again, this does not cause a cycle, so it is added to the collection of edges.

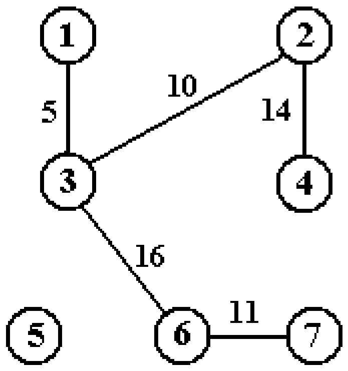
We now have E_T = {(1, 3), (2, 3), (6, 7)}.





The next edge in the sort order is (2, 4) which does not cause a cycle. so it is also included in the collection of edges.

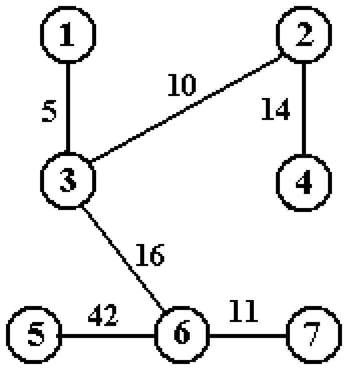
We now have E_T = {(1, 3), (2, 3), (2, 4), (6, 7)}.



The next edge in sort order is (3, 6) which still does not form a cycle, so we can add it to the set.

We now have E_T = {(1, 3), (2, 3), (2, 4), (3, 6), (6, 7)}.

The next edge to be considered is (3, 4). It causes a cycle.
We now consider edge (1, 2). It also causes a cycle.
Edge (4, 6) is next in line. It too causes a cycle.



The last edge that we consider is (5, 6), which fortunately does not cause a cycle. We add it to the edge set, so we get the edge set as
E_T = {(1, 3), (2, 3), (2, 4), (3, 6), (5, 6), (6, 7)}

The edge set contains 6 edges, one less than the number of vertices. We are done, and with great relief, report that the MST generated is identical to that generated by Prim's algorithm.

Spanning Trees for Unweighted Graphs

It should be obvious that either Prim's Algorithm or Kruskal's Algorithm can be adopted to create spanning trees in unweighted graphs. For each, the modification is simple.
In Prim's, one replaces the statement

"Select an edge of minimum weight between a tree node *t* and a fringe node *u*"

with the statement

"Select any edge between a tree node *t* and a fringe node *u*".

In Kruskal's algorithm, the only requirement is to omit the sorting of edges by weight, as they all have the same weight in an unweighted graph, and begin with any listing of edges.

Testing the Growing Tree for Cycles

Up to now, we have been ignoring a very important part of these two algorithms. At each stage of both algorithms, we consider adding an edge and ask whether or not the addition of this edge will cause a cycle. This has begged the question of detecting cycles. We now attempt to address this problem, beginning with an obvious lemma of little applicability to our problem.

Lemma 19: If G is an (n, m)-graph with $n \geq m$, then G contains a cycle.
Proof: Assume otherwise. Let G be an (n, m)-graph, with $m = (n - 1) + k$, $k \geq 1$. If G is really acyclic, then removal of any k of the edges will not add a cycle to the graph. Let e be the last of the k edges removed (if $k = 1$, then e is the only edge removed). What we have at the end of this process is an acyclic (n, $n - 1$) graph; in other words, a tree. By Theorem 8, the addition of any edge to this graph will cause a cycle. But we assume that before the removal of e, G was acyclic, so that it cannot now have a cycle by adding the edge back. Thus we have a contradiction, and our original graph G could not have been acyclic. Hence the lemma is proven.

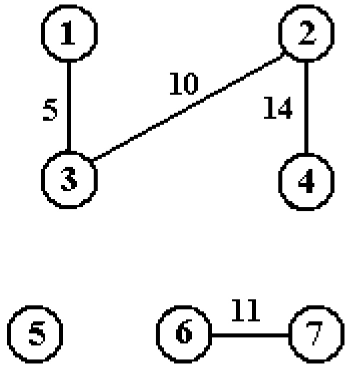
While this is a perfectly fine lemma, it was proven just to show the author of these notes could do it. It fact, application of either Prim's algorithm and Kruskal's algorithm to an (n, m)-graph involves construction of edge sets of no more than ($n - 1$) edges; that is to say that the intermediate edge sets (viewing the tree in Prim's as just a set of edges) have fewer than ($n - 1$) edges and the final edge sets have exactly ($n - 1$) edges. It each case, the number of edges is too small for Lemma 19 to apply, but it was fun to prove.

Prim's algorithm avoids the creation of cycles by labeling the vertices inserted into the growing spanning tree as tree vertices or fringe vertices. The only edges added to the tree are those that connect a fringe vertex to a tree vertex. The only way for a vertex to become a fringe vertex is for it to have been an unseen vertex and be adjacent to a vertex that has just become a tree vertex. In particular there is no way to convert a tree vertex to a fringe vertex and therefore no way to connect two tree vertices. Hence the algorithm will not form a cycle.

The correctness of Kruskal's algorithm depends on the one line in the algorithm.

```
        If $E_T \cup \{e_k\}$ is acyclic, then $E_T = E_T \cup \{e_k\}$
```

We now address the problem of showing that adding an edge to the edge set E_T does not produce a cycle. We begin by giving a different description of the edge set E_T. Recalling the definition of a tree, we define a **forest** as a collection of one or more trees not connected to each other.



Let's return to one stage in the development of the spanning tree in the above example using Kruskal's algorithm. Here we see the situation after adding four edges. There are two connected components, one containing vertices 1, 2, 3, and 4 and the other containing vertices 6 and 7. These can be seen as two trees in a forest that will eventually grow to a spanning tree.

In presenting Kruskal's algorithm, we presented the discussion in terms of a set of edges; E_T = { (1, 3), (2, 3), (2, 4), (6, 7). In order to complete the algorithm, we must view it as a forest, with two trees T_1 and T_2, with $V(T_1)$ = {1, 2, 3, 4) and $V(T_2)$ = {6, 7}, viewed as connected components.

We now state an important lemma.
Lemma 20: Let F be a forest, that is, any undirected acyclic graph. Let $e = (v, w)$ be an edge that is not in F. There is a cycle consisting of *e* and edges in F if and only if *v* and *w* are in the came connected component of F.

The proof of this lemma is rather simple and consists of the observation that one creates a cycle in a graph only by providing two paths between two vertices in the graph. If the addition of $e = (v, w)$ to the forest F creates a cycle, there must have already been at least one path between the vertices *v* and *w*, hence they would be in the same connected component.

There are many algorithms useful for determining connected components in a graph. Among these are BFS (Breadth First Search) and DFS (Depth First Search). Since we are building our graph F one edge at a time, we state a simpler algorithm, based on the use of linked lists to represent connected components of F.

```
Algorithm ConnectedComp ($e = (u, v)$)        -- Add edge $e = (u, v)$ to F
    If $e = (u, v)$ is the first edge added, create component $C_1 = \{u, v\}$.
    Otherwise scan all of the existing component linked lists for vertices $u$ and $v$.
        If one component contains both $u$ and $v$, reject the edge as forming a cycle.
        If component $C_J$ contains one vertex (call it $u$) and vertex $v$ is not
            contained in any component, add vertex $v$ to component $C_J$.

        If vertex $u$ belongs to component $C_J$ and vertex $v$ belongs to component $C_K$,
            then merge the two components into one component.
        If neither $u$ nor $v$ are in a component, create a new component $C_N$ and
            set $C_N = (u, v)$.
```

With this in mind, let's reconsider the application of Kruskal's algorithm to the graph described in figure 21. In order to apply a strictly algorithmic approach, we give a description of the graph as it would be used in a computer program, not as it would be drawn on paper. We present the description at a point that the edges have been sorted by non-decreasing weight.

V = { 1, 2, 3, 4, 5, 6, 7 }
E = { (1, 3), (2, 3), (6, 7), (2, 4), (3, 6), (3, 4), (1, 2), (4, 6), (5, 6) }
W = { 5, 10, 11, 14, 16, 18, 25, 32, 42 }

We now apply the algorithm. As we create the edge set E_T, we create the connected components as needed. The first connected component will be C_1.

1 Consider the first edge (1, 3). Set C_1 = {1, 3} and E_T = { (1, 3) }.

2 Consider edge (2, 3). Vertex 3 is in C_1 and vertex 2 is not.
C_1 = {1, 2, 3} and E_T = { (1, 3), (2, 3) }.

3 Consider edge (6, 7). Neither vertex 6 nor vertex 7 is in C_1, so create a new component. C_1 = {1, 2, 3}, C_2 = {6, 7}
E_T = { (1, 3), (2, 3), (6, 7) }.

4 Consider edge (2, 4). Vertex 2 is in C_1 and vertex 4 is not in any component.
C_1 = {1, 2, 3, 4}, C_2 = {6, 7}
E_T = { (1, 3), (2, 3), (2, 4), (6, 7) }.

5 Consider edge (3, 6). Vertex 3 is in C_1, and vertex 6 is in C_2. Merge the components and call the new component C_1.
C_1 = {1, 2, 3, 4, 6, 7}.
E_T = { (1, 3), (2, 3), (2, 4), (3, 6), (6, 7) }.

6 Consider edge (3, 4). Both vertices 3 and 4 are in C_1. Reject the edge.

7 Consider edge (1, 2). Both vertices 1 and 2 are in C_1. Reject the edge.

8 Consider edge (4, 6). Both vertices 4 and 6 are in C_1. Reject the edge.

9 Consider edge (5, 6). Vertex 6 is in C_1 and vertex 5 is not in any component.
C_1 = {1, 2, 3, 4, 5, 6, 7}.
E_T = { (1, 3), (2, 3), (2, 4), (3, 6), (5, 6), (6, 7) }.

Applicability to Disconnected Graphs

We now consider the operation of Prim's algorithm and Kruskal's algorithms on a graph G that is not connected, but rather comprises $k > 1$ connected components. It should be obvious that Prim's algorithm will not process the entire graph. Let *s* be the starting vertex for Prim's algorithm and number the components of the graph so that *s* is a part of connected component 1. Prim's algorithm will find a minimum spanning tree for component 1 and then terminate, since none of the vertices in the other components are adjacent to vertices in component 1.

It should be obvious that Kruskal's algorithm as applied to this graph, will produce a collection of *k* spanning trees, one tree spanning each of the *k* connected components. This might be called a **spanning forest** if anybody but this author used such terminology.

Euclidean Graphs and Those that are not.

Let G be a weighted graph, with edge weights denoted as w(*u*, *v*) for adjacent vertices *u* and *v*.

We can specify that a graph is to be called Euclidean, if for any three mutually adjacent vertices $x, y, z \in V(G)$. we have the inequality $w(x, y) \leq w(x, z) + w(y, z)$. Obviously graphs that represent positions of points on a plane satisfy this equality. This inequality is often called the triangle inequality.

One should note that some weighted graphs lack the nice properties that come to mind when we imagine maps as the model of directed graphs. The following figure is a real example, based on the costs of a one-way direct flight between two cities on a given day in the year 2003. Note that the triangle inequality is frequently violated.

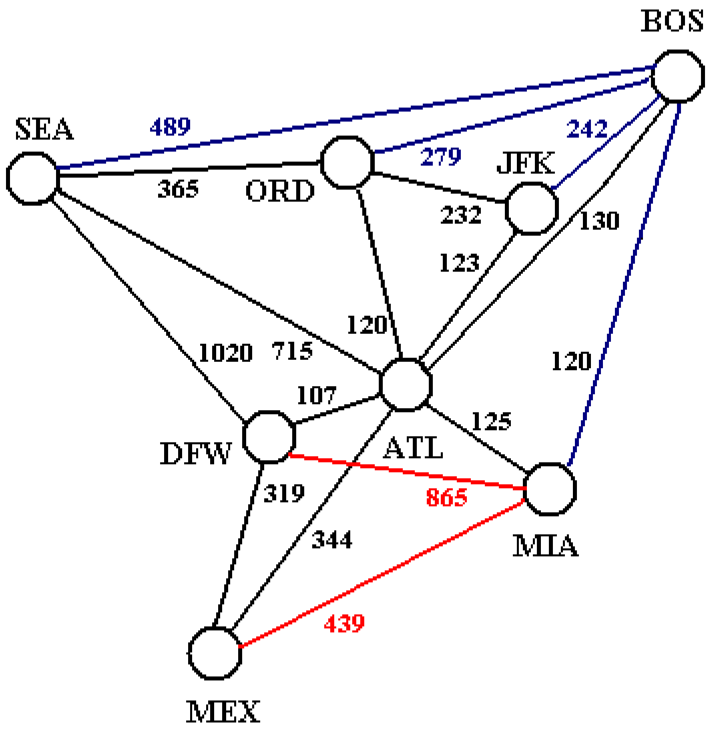


Figure 25: Some One-Way Direct Air Fares

Note in particular, the options for flights between DFW (Dallas-Fort Worth) and MIA (Miami). A direct flight costs $865, while an indirect flight through ATL (Atlanta) costs only $232 and an indirect flight through MEX (Mexico City) costs only$768.

As an aside, one should note that it is not easy to understand air fares.

Warshall's Algorithm

Warshall's algorithm is the first of two graph algorithms based on dynamic programming that we shall study in this class. Warshall's algorithm computes the **transitive closure** (defined below) of a directed graph and, by extension, an undirected graph since every undirected graph can be represented as a directed graph.

Background

We first need to recall the two basic Boolean functions: AND and OR. Each of these is a Boolean function of two Boolean variables, which can take the value 0 (False) or 1 (True). Each of these two functions can be defined by a truth table or by description.

For Boolean variables X and Y

X AND Y = 1 if and only if X = 1 and Y = 1. Otherwise it is 0.
X OR Y = 0 if and only if X = 0 and Y = 0. Otherwise it is 1.

The truth table definitions are given to be complete.

X	Y	AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

The **adjacency matrix** $A = \{A_{IJ}\}$ of a directed graph is the Boolean matrix that has $A_{IJ} = 1$ if and only if there is a directed edge from vertex I to vertex J. If $A_{IJ} \neq 1$, then $A_{IJ} = 0$. For a directed graph having N vertices, the adjacency matrix is an N-by-N matrix.

The **transitive closure** $T = \{T_{IJ}\}$ of a directed graph with N vertices is an N-by-N Boolean matrix in which $T_{IJ} = 1$ if and only if there exists a non-trivial directed path (a directed path of positive length) from vertex I to vertex J. If $T_{IJ} \neq 1$, then $T_{IJ} = 0$.

Obviously, if $A_{IJ} \neq 0$, then $T_{IJ} \neq 0$. If $A_{IJ} = 0$, then T_{IJ} may be either 0 or 1.

There are a number of ways to compute the transitive closure of a directed graph. We focus on one of the more efficient methods, called Warshall's algorithm. Warshall's algorithm computes the transitive closure matrix T through a sequence of Boolean matrices

$$R^{(0)}, R^{(1)}, \ldots R^{(K-1)}, R^{(K)}, \ldots R^{(N)}.$$

The definition of these matrices is that $R^{(K)}_{IJ} = 1$ if and only if there is a directed path from vertex I to vertex J with each intermediate vertex, if any, not numbered higher than K. By this definition, the matrix $R^{(0)}$ has elements $R^{(0)}_{IJ} = 1$ if and only the intermediate path contains no vertices numbered above 0; thus no vertices – $R^{(0)} = A$, the adjacency matrix.

The matrix $R^{(N)}$ shows all directed paths that can use any vertex from the vertex set V(G). Thus we conclude that $R^{(N)} = T$, the transitive closure.

To be specific, we give a few obvious definitions.

$R^{(1)}_{IJ} = 1$ if and only if there is either a direct path from vertex I to J or if there is a path of length 2 that goes through only vertex 1.

$R^{(2)}_{IJ} = 1$ if and only if there is either
1) a direct path from vertex I to J, or
2) a path of length 2 that goes through only vertex 1, or
3) a path of length 2 that goes through only vertex 2, or
4) a path of length 3 that goes through only vertices 1 and 2, in either order.

So far all we have is a bunch of interesting definitions and observations. There is one fact that turns the above into a useable algorithm – the fact that matrix $R^{(K)}$ can be computed from only its predecessor matrix $R^{(K-1)}$.

Consider the computation of element $R^{(K)}_{IJ}$, which is 1 if and only if there is a directed path from vertex I to vertex J, containing no vertex numbered higher than K. Now consider the matrix $R^{(K-1)}$, with its element $R^{(K-1)}_{IJ}$ defined similarly.

Suppose that $R^{(K-1)}_{IJ} = 1$. Then there is a path from vertex I to vertex J, containing no intermediate vertex numbered higher than (K – 1), certainly fulfilling the criteria that there be no intermediate vertex numbered higher than K. Thus we arrive at the first conclusion.

$$\text{If } R^{(K-1)}_{IJ} == 1 \text{ then } R^{(K)}_{IJ} = 1.$$

If $R^{(K-1)}_{IJ} = 0$, there is another way to create a directed path from vertex I to vertex J, containing no vertex numbered higher than K.
1) there is a path from vertex I to vertex K involving no vertex numbered higher than (K – 1), and
2) there is a path from vertex K to vertex J involving no vertex numbered higher than (K – 1).

Thus we have the second conclusion

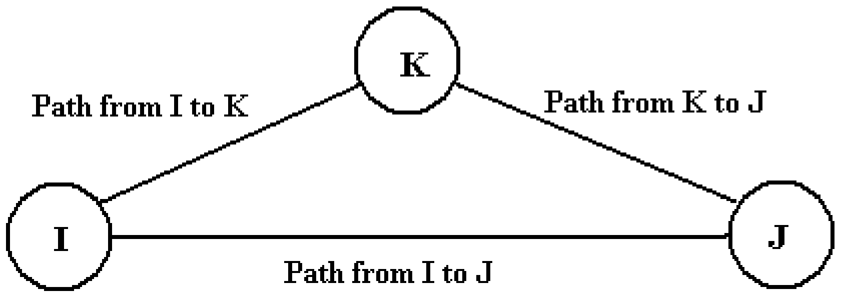
$$\text{If } R^{(K-1)}_{IK} == 1 \text{ and } R^{(K-1)}_{KJ} = 1 \text{ then } R^{(K)}_{IJ} = 1.$$

Based on these two conclusions, we have the fundamental Boolean equation of the method.

$$R^{(K)}_{IJ} = R^{(K-1)}_{IJ} \text{ OR } (R^{(K-1)}_{IK} \text{ AND } R^{(K-1)}_{KJ})$$

Note that by definition of a path in a graph, neither the start vertex nor the end vertex is a part of the path. Thus consider the paths from vertex I to vertex K containing no intermediate vertex numbered greater than K. Since vertex K is the end point of the path, it cannot appear as an intermediate vertex and the path, if present, contains no intermediate vertex numbered greater than (K – 1).

The situation discussed above may be illustrated by the figure below



Each path involves only vertices in the set 1 .. (K - 1)

The pseudocode for the simple version of Warshall's algorithm is found in the book and repeated here. Note that it constructs a sequence of (N + 1) matrices – $R^{(0)}$ through $R^{(N)}$.

```
Algorithm Warshall (A[1..N, 1..N])
// Implements Warshall's algorithm for computing the
transitive
// closure of a graph on N nodes with adjacency matrix A.
//
// This uses a sequence of (N + 1) matrices, each N-by-N, and
// labeled $R^{(0)}$, $R^{(1)}$, …, $R^{(N - 1)}$, $R^{(N)}$.
//
// $R^{(0)}$ = A     // Set matrix $R^{(0)}$ equal to matrix A.
//
   For K = 1 to N Do
      For I = 1 to N Do
            For J = 1 to N Do
                 $R^{(K)}$[I,J] = $R^{(K-1)}$[I,J] Or
                             ($R^{(K-1)}$[I,K] And $R^{(K-1)}$[K,J])
            End Do // J
      End Do // I
   End Do // K
//
   Return $R^{(N)}$
```

The above algorithm has been written with the sequence of matrices in order to emphasize the dynamic-programming nature of the algorithm. In face, it is quite possible to avoid generating more than one matrix in this algorithm, as we shall see shortly. In order to facilitate this discussion, I shall rewrite the assignment statement of the algorithm above in an equivalent, but apparently less efficient, form.

```
                 $R^{(K)}$[I,J] = $R^{(K-1)}$[I,J]
//               If $R^{(K)}$[I,J] == 0, then do the computation.
                 If !$R^{(K)}$[I,J]
                      Then $R^{(K)}$[I,J] = $R^{(K-1)}$[I,K] And $R^{(K-1)}$[K,J]
```

The key to simplifying the space complexity of this algorithm is based on the following.

1) $R^{(K)}[I, K] = R^{(K-1)}[I, K]$ as no path from vertex I to vertex K involves vertex K as as intermediate node, hence all paths involving only vertices 1 though K must be limited to vertices 1 though (K – 1).

2) $R^{(K)}[K, J] = R^{(K-1)}[K, J]$ for the same reason.

Based on this observation, we can create a simpler version of Warshall’s algorithm.

```
Algorithm Warshall (A[1..N, 1..N], R[1..N, 1..N])
// Implements Warshall’s algorithm for computing the transitive
// closure of a graph on N nodes with adjacency matrix A.
//
// This implements the version of Warshall’s algorithm that uses
// only two matrices – one for the input and one for the output.
//
// Input:  A – the N-by-N adjacency matrix of the graph.
// Output: R – the N-by-N transitive closure of the graph.
//
   For I = 1 to N Do           // Set R = A
      For J = 1 to N Do
           R[I, J] = A[I, J]
      End Do // J
   End Do // I
//
   For K = 1 to N Do
      For I = 1 to N Do
           For J = 1 to N Do
                If !R[I,J]
                     Then R[I,J] = R[I,K] And R[K,J]
           End Do // J
      End Do // I
   End Do // K
//
   Return R
```

As an example, we compute the transitive closure of the graph with adjacency matrix

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

In order to make the example perfectly clear, we follow the first version of Warshall’s algorithm with the numbered matrices beginning with $R^{(0)} = A$.

The start matrix is $R^{(0)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$.

K = 1
The loop for K = 1 generates $R^{(1)}$. The operative code is the following line.

```
R(1)[I,J] = R(0)[I,J]
If (0 == R(1)[I,J])
     Then R(1)[I,J] = R(0)[I,K] And R(0)[K,J]
```

But note that $R^{(0)}[1, J] = 0$ for all values of J. The end result is that none of the zero values in the matrix $R^{(0)}$ are changed to 1, hence we have $R^{(1)} = R^{(0)}$.

We now have $R^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

K = 2
The loop for K = 2 generates $R^{(2)}$. The operative code is the following line.

```
R(2)[I,J] = R(1)[I,J]
If (0 == R(2)[I,J])
     Then R(2)[I,J] = R(1)[I,2] And R(1)[2,J]
```

I = 1, J = 1 $R^{(1)}[1, 1] = 0$, so $R^{(2)}[1, 1] = R^{(1)}[1, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 1] = 1 \bullet 0 = 0$
I = 1, J = 2 $R^{(1)}[1, 2] = 1$, so $R^{(2)}[1, 2] = 1$
I = 1, J = 3 $R^{(1)}[1, 3] = 0$, so $R^{(2)}[1, 3] = R^{(1)}[1, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 3] = 1 \bullet 1 = 1$
I = 1, J = 4 $R^{(1)}[1, 4] = 0$, so $R^{(2)}[1, 3] = R^{(1)}[1, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 4] = 1 \bullet 0 = 0$
I = 1, J = 5 $R^{(1)}[1, 5] = 1$, so $R^{(2)}[1, 5] = 1$

I = 2, J = 1 $R^{(1)}[2, 1] = 0$, so $R^{(2)}[2, 1] = R^{(1)}[2, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 1] = 0 \bullet 0 = 0$
I = 2, J = 2 $R^{(1)}[2, 2] = 0$, so $R^{(2)}[2, 2] = R^{(1)}[2, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 2] = 0 \bullet 0 = 0$
I = 2, J = 3 $R^{(1)}[2, 3] = 1$, so $R^{(2)}[2, 3] = 1$
I = 2, J = 4 $R^{(1)}[2, 4] = 0$, so $R^{(2)}[2, 4] = R^{(1)}[2, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 4] = 0 \bullet 0 = 0$
I = 2, J = 5 $R^{(1)}[2, 5] = 1$, so $R^{(2)}[2, 5] = 1$

I = 3, J = 1 $R^{(1)}[3, 1] = 0$, so $R^{(2)}[3, 1] = R^{(1)}[3, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 1] = 1 \bullet 0 = 0$

I = 3, J = 2 $R^{(1)}[3, 2] = 1$, so $R^{(2)}[3, 2] = 1$

I = 3, J = 3 $R^{(1)}[3, 3] = 0$, so $R^{(2)}[3, 3] = R^{(1)}[3, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 3] = 1 \bullet 1 = 1$

I = 3, J = 4 $R^{(1)}[3, 4] = 1$, so $R^{(2)}[3, 4] = 1$

I = 3, J = 5 $R^{(1)}[3, 5] = 0$, so $R^{(2)}[3, 5] = R^{(1)}[3, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 5] = 1 \bullet 1 = 1$

I = 4, J = 1 $R^{(1)}[4, 1] = 0$, so $R^{(2)}[4, 1] = R^{(1)}[4, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 1] = 0 \bullet 0 = 0$

I = 4, J = 2 $R^{(1)}[4, 2] = 0$, so $R^{(2)}[4, 2] = R^{(1)}[4, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 2] = 0 \bullet 0 = 0$

I = 4, J = 3 $R^{(1)}[4, 3] = 0$, so $R^{(2)}[4, 3] = R^{(1)}[4, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 3] = 0 \bullet 1 = 0$

I = 4, J = 4 $R^{(1)}[4, 4] = 0$, so $R^{(2)}[4, 4] = R^{(1)}[4, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 4] = 0 \bullet 0 = 0$

I = 4, J = 5 $R^{(1)}[4, 5] = 1$, so $R^{(2)}[4, 5] = 1$

I = 5, J = 1 $R^{(1)}[5, 1] = 0$, so $R^{(2)}[5, 1] = R^{(1)}[5, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 1] = 0 \bullet 0 = 0$

I = 5, J = 2 $R^{(1)}[5, 2] = 0$, so $R^{(2)}[5, 2] = R^{(1)}[5, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 2] = 0 \bullet 0 = 0$

I = 5, J = 3 $R^{(1)}[5, 3] = 0$, so $R^{(2)}[5, 3] = R^{(1)}[5, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 3] = 0 \bullet 1 = 0$

I = 5, J = 4 $R^{(1)}[5, 4] = 1$, so $R^{(2)}[5, 4] = 1$

I = 5, J = 5 $R^{(1)}[5, 5] = 0$, so $R^{(2)}[5, 5] = R^{(1)}[5, \mathbf{2}]$ And $R^{(1)}[\mathbf{2}, 5] = 0 \bullet 1 = 0$

We now have $R^{(2)} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

K = 3
The loop for K = 3 generates $R^{(3)}$. The operative code is the following line.

```
R(3)[I,J] = R(2)[I,J]
If (0 == R(3)[I,J])
     Then R(3)[I,J] = R(2)[I,3] And R(2)[3,J]
```

I = 1, J = 1 $R^{(2)}[1, 1] = 0$, so $R^{(3)}[1, 1] = R^{(2)}[1, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 1] = 1 \bullet 0 = 0$

I = 1, J = 2 $R^{(2)}[1, 2] = 1$, so $R^{(3)}[1, 2] = 1$

I = 1, J = 3 $R^{(2)}[1, 3] = 1$, so $R^{(3)}[1, 3] = 1$

I = 1, J = 4 $R^{(2)}[1, 4] = 0$, so $R^{(3)}[1, 4] = R^{(2)}[1, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 4] = 1 \bullet 1 = 1$

I = 1, J = 5 $R^{(2)}[1, 5] = 1$, so $R^{(3)}[1, 5] = 1$

I = 2, J = 1	$R^{(2)}[2, 1] = 0$, so $R^{(3)}[2, 1] = R^{(2)}[2, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 1] = 1 \bullet 0 = 0$
I = 2, J = 2	$R^{(2)}[2, 2] = 0$, so $R^{(3)}[2, 2] = R^{(2)}[2, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 2] = 1 \bullet 1 = 1$
I = 2, J = 3	$R^{(2)}[2, 3] = 1$, so $R^{(3)}[2, 3] = 1$
I = 2, J = 4	$R^{(2)}[2, 4] = 0$, so $R^{(3)}[2, 4] = R^{(2)}[2, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 4] = 1 \bullet 1 = 1$
I = 2, J = 5	$R^{(2)}[2, 5] = 1$, so $R^{(3)}[2, 5] = 1$
I = 3, J = 1	$R^{(2)}[3, 1] = 0$, so $R^{(3)}[3, 1] = R^{(2)}[3, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 1] = 1 \bullet 0 = 0$
I = 3, J = 2	$R^{(2)}[3, 2] = 1$, so $R^{(3)}[3, 2] = 1$
I = 3, J = 3	$R^{(2)}[3, 3] = 1$, so $R^{(3)}[3, 3] = 1$
I = 3, J = 4	$R^{(2)}[3, 4] = 1$, so $R^{(3)}[3, 4] = 1$
I = 3, J = 5	$R^{(2)}[3, 5] = 1$, so $R^{(3)}[3, 5] = 1$
I = 4, J = 1	$R^{(2)}[4, 1] = 0$, so $R^{(3)}[4, 1] = R^{(2)}[4, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 1] = 0 \bullet 0 = 0$
I = 4, J = 2	$R^{(2)}[4, 2] = 0$, so $R^{(3)}[4, 2] = R^{(2)}[4, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 2] = 0 \bullet 1 = 0$
I = 4, J = 3	$R^{(2)}[4, 3] = 0$, so $R^{(3)}[4, 3] = R^{(2)}[4, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 3] = 0 \bullet 1 = 0$
I = 4, J = 4	$R^{(2)}[4, 4] = 0$, so $R^{(3)}[4, 4] = R^{(2)}[4, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 4] = 0 \bullet 1 = 0$
I = 4, J = 5	$R^{(2)}[4, 5] = 1$, so $R^{(3)}[4, 5] = 1$
I = 5, J = 1	$R^{(2)}[5, 1] = 0$, so $R^{(3)}[5, 1] = R^{(2)}[5, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 1] = 0 \bullet 0 = 0$
I = 5, J = 2	$R^{(2)}[5, 2] = 0$, so $R^{(3)}[5, 2] = R^{(2)}[5, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 2] = 0 \bullet 1 = 0$
I = 5, J = 3	$R^{(2)}[5, 3] = 0$, so $R^{(3)}[5, 3] = R^{(2)}[5, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 3] = 0 \bullet 1 = 0$
I = 5, J = 4	$R^{(2)}[5, 4] = 1$, so $R^{(3)}[5, 4] = 1$
I = 5, J = 5	$R^{(2)}[5, 5] = 0$, so $R^{(3)}[5, 5] = R^{(2)}[5, \mathbf{3}]$ And $R^{(2)}[\mathbf{3}, 5] = 0 \bullet 1 = 0$

We now have $R^{(3)} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

K = 4

The loop for K = 4 generates $R^{(4)}$. The operative code is the following line.

```
$R^{(4)}$[I,J] = $R^{(3)}$[I,J]
If (0 == $R^{(4)}$[I,J])
     Then $R^{(4)}$[I,J] = $R^{(3)}$[I,4] And $R^{(3)}$[4,J]
```

I = 1, J = 1	$R^{(3)}[1, 1] = 0$, so $R^{(4)}[1, 1] = R^{(3)}[1, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 1] = 1 \bullet 0 = 0$
I = 1, J = 2	$R^{(3)}[1, 2] = 1$, so $R^{(4)}[1, 2] = 1$
I = 1, J = 3	$R^{(3)}[1, 3] = 1$, so $R^{(4)}[1, 3] = 1$
I = 1, J = 3	$R^{(3)}[1, 4] = 1$, so $R^{(4)}[1, 4] = 1$
I = 1, J = 5	$R^{(3)}[1, 5] = 1$, so $R^{(4)}[1, 5] = 1$

I = 2, J = 1	$R^{(3)}[2, 1] = 0$, so $R^{(4)}[2, 1] = R^{(3)}[2, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 1] = 1 \bullet 0 = 0$
I = 2, J = 2	$R^{(3)}[2, 2] = 1$, so $R^{(4)}[2, 2] = 1$
I = 2, J = 3	$R^{(3)}[2, 3] = 1$, so $R^{(4)}[2, 3] = 1$
I = 2, J = 3	$R^{(3)}[2, 4] = 1$, so $R^{(4)}[2, 4] = 1$
I = 2, J = 5	$R^{(3)}[2, 5] = 1$, so $R^{(4)}[2, 5] = 1$
I = 3, J = 1	$R^{(3)}[3, 1] = 0$, so $R^{(4)}[3, 1] = R^{(3)}[3, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 1] = 1 \bullet 0 = 0$
I = 3, J = 2	$R^{(3)}[3, 2] = 1$, so $R^{(4)}[3, 2] = 1$
I = 3, J = 3	$R^{(3)}[3, 3] = 1$, so $R^{(4)}[3, 3] = 1$
I = 3, J = 3	$R^{(3)}[3, 4] = 1$, so $R^{(4)}[3, 4] = 1$
I = 3, J = 5	$R^{(3)}[3, 5] = 1$, so $R^{(4)}[3, 5] = 1$
I = 4, J = 1	$R^{(3)}[4, 1] = 0$, so $R^{(4)}[4, 1] = R^{(3)}[4, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 1] = 0 \bullet 0 = 0$
I = 4, J = 2	$R^{(3)}[4, 2] = 0$, so $R^{(4)}[4, 2] = R^{(3)}[4, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 2] = 0 \bullet 0 = 0$
I = 4, J = 3	$R^{(3)}[4, 3] = 0$, so $R^{(4)}[4, 3] = R^{(3)}[4, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 3] = 0 \bullet 0 = 0$
I = 4, J = 4	$R^{(3)}[4, 4] = 0$, so $R^{(4)}[4, 4] = R^{(3)}[4, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 4] = 0 \bullet 0 = 0$
I = 4, J = 5	$R^{(3)}[4, 5] = 1$, so $R^{(4)}[4, 5] = 1$
I = 5, J = 1	$R^{(3)}[5, 1] = 0$, so $R^{(4)}[5, 1] = R^{(3)}[5, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 1] = 1 \bullet 0 = 0$
I = 5, J = 2	$R^{(3)}[5, 2] = 0$, so $R^{(4)}[5, 2] = R^{(3)}[5, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 2] = 1 \bullet 0 = 0$
I = 5, J = 3	$R^{(3)}[5, 3] = 0$, so $R^{(4)}[5, 3] = R^{(3)}[5, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 3] = 1 \bullet 0 = 0$
I = 5, J = 4	$R^{(3)}[5, 4] = 1$, so $R^{(4)}[5, 4] = 1$
I = 5, J = 5	$R^{(3)}[5, 3] = 0$, so $R^{(4)}[5, 3] = R^{(3)}[5, \mathbf{4}]$ And $R^{(3)}[\mathbf{4}, 5] = 1 \bullet 1 = 1$

We now have $R^{(4)} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$

K = 5

The loop for K = 5 generates $R^{(5)}$. The operative code is the following line.

```
    $R^{(5)}[I,J] = R^{(4)}[I,J]$
    If (0 == $R^{(5)}[I,J]$)
         Then $R^{(5)}[I,J] = R^{(4)}[I,5]$ And $R^{(4)}[5,J]$
```

I = 1, J = 1	$R^{(4)}[1, 1] = 0$, so $R^{(5)}[1, 1] = R^{(4)}[1, \mathbf{5}]$ And $R^{(4)}[\mathbf{5}, 1] = 1 \bullet 0 = 0$
I = 1, J = 2	$R^{(4)}[1, 2] = 1$, so $R^{(5)}[1, 2] = 1$
I = 1, J = 3	$R^{(4)}[1, 3] = 1$, so $R^{(5)}[1, 3] = 1$
I = 1, J = 3	$R^{(4)}[1, 4] = 1$, so $R^{(5)}[1, 4] = 1$
I = 1, J = 5	$R^{(4)}[1, 5] = 1$, so $R^{(5)}[1, 5] = 1$
I = 2, J = 1	$R^{(4)}[2, 1] = 0$, so $R^{(5)}[2, 1] = R^{(4)}[2, \mathbf{5}]$ And $R^{(4)}[\mathbf{5}, 1] = 1 \bullet 0 = 0$
I = 2, J = 2	$R^{(4)}[2, 2] = 1$, so $R^{(5)}[2, 2] = 1$
I = 2, J = 3	$R^{(4)}[2, 3] = 1$, so $R^{(5)}[2, 3] = 1$
I = 2, J = 3	$R^{(4)}[2, 4] = 1$, so $R^{(5)}[2, 4] = 1$
I = 2, J = 5	$R^{(4)}[2, 5] = 1$, so $R^{(5)}[2, 5] = 1$
I = 3, J = 1	$R^{(4)}[3, 1] = 0$, so $R^{(5)}[3, 1] = R^{(4)}[3, \mathbf{5}]$ And $R^{(4)}[\mathbf{5}, 1] = 1 \bullet 0 = 0$
I = 3, J = 2	$R^{(4)}[3, 2] = 1$, so $R^{(5)}[3, 2] = 1$
I = 3, J = 3	$R^{(4)}[3, 3] = 1$, so $R^{(5)}[3, 3] = 1$
I = 3, J = 3	$R^{(4)}[3, 4] = 1$, so $R^{(5)}[3, 4] = 1$
I = 3, J = 5	$R^{(4)}[3, 5] = 1$, so $R^{(5)}[3, 5] = 1$
I = 4, J = 1	$R^{(4)}[4, 1] = 0$, so $R^{(5)}[4, 1] = R^{(4)}[4, \mathbf{5}]$ And $R^{(4)}[\mathbf{5}, 1] = 1 \bullet 0 = 0$
I = 4, J = 2	$R^{(4)}[4, 2] = 0$, so $R^{(5)}[4, 2] = R^{(4)}[4, \mathbf{5}]$ And $R^{(4)}[\mathbf{5}, 2] = 1 \bullet 0 = 0$
I = 4, J = 3	$R^{(4)}[4, 3] = 0$, so $R^{(5)}[4, 3] = R^{(4)}[4, \mathbf{5}]$ And $R^{(4)}[\mathbf{5}, 3] = 1 \bullet 0 = 0$
I = 4, J = 4	$R^{(4)}[4, 4] = 0$, so $R^{(5)}[4, 4] = R^{(4)}[4, \mathbf{5}]$ And $R^{(4)}[\mathbf{5}, 4] = 1 \bullet 1 = 1$
I = 4, J = 5	$R^{(4)}[4, 5] = 1$, so $R^{(5)}[4, 5] = 1$
I = 5, J = 1	$R^{(4)}[5, 1] = 0$, so $R^{(5)}[5, 1] = R^{(4)}[5, \mathbf{5}]$ And $R^{(4)}[\mathbf{5}, 1] = 1 \bullet 0 = 0$
I = 5, J = 2	$R^{(4)}[5, 2] = 0$, so $R^{(5)}[5, 2] = R^{(4)}[5, \mathbf{5}]$ And $R^{(4)}[\mathbf{5}, 2] = 1 \bullet 0 = 0$
I = 5, J = 3	$R^{(4)}[5, 3] = 0$, so $R^{(5)}[5, 3] = R^{(4)}[5, \mathbf{5}]$ And $R^{(4)}[\mathbf{5}, 3] = 1 \bullet 0 = 0$
I = 5, J = 4	$R^{(4)}[5, 4] = 1$, so $R^{(5)}[5, 4] = 1$
I = 5, J = 5	$R^{(4)}[5, 5] = 1$, so $R^{(5)}[5, 5] = 1$

We now have $R^{(5)} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$

The above matrix, $R^{(5)}$, represents the transitive closure of the directed graph. What can we say about the graph?

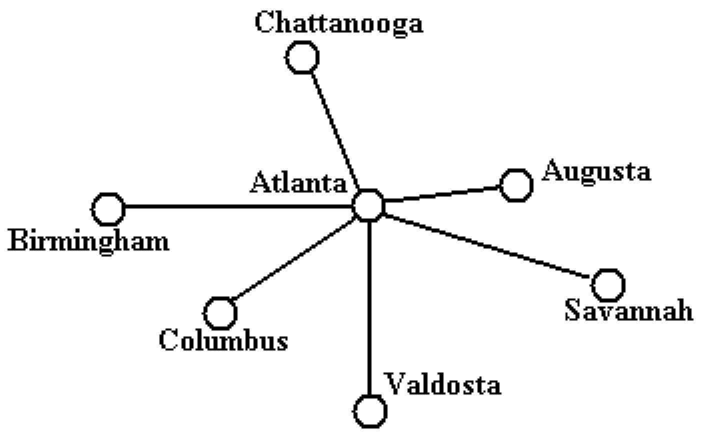
Vertex 1	there is no path to vertex 1 from any vertex.
Vertices 2 and 3	there is a path to these vertices from vertices 1, 2, and 3
Vertices 4 and 5	there is a path to these vertices from every vertex.

Floyd's Algorithm for the All-Pairs Shortest-Path Problem

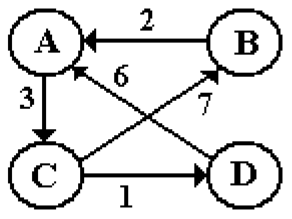
Given a weighted connected graph (directed or undirected), the **all-pairs shortest path** problem asks us to find the distance of the shortest path from each vertex to every other vertex in the connected component of the graph. **Weighted graphs**, discussed in section 1.4 of the textbook, are a special case of simple graphs in which there is a weight, cost, or distance associated with each of the edges of the graph. We are all familiar with weighted graphs and use them all the time for daily activities such as travel.

Entries in the adjacency matrix of a simple graph are 1 if the edge exists and 0 if the edge does not. In a weighted graph, the entries in the adjacency matrix represent the weight of the edge if it exists. Handling non-existent edges is somewhat of a problem.

Consider the following simple graph with unweighted edges, based on the interstate highway system in the state of Georgia. This is somewhat familiar to us, except that we expect to see distances on each of the edges. The association of a distance with each edge is what changes the graph from a simple graph to a simple weighted graph.



We shall now take a very close look at the example in the book, spending some time to consider issues that the book overlooks. Consider the following figure.



The graph is shown at left, its adjacency matrix just below.

$$W = \begin{bmatrix} 0 & & 3 & \\ 2 & 0 & & \\ & 7 & 0 & 1 \\ 6 & & & 0 \end{bmatrix}$$. Note the missing entries.

Here we face the problem of handling non-existent edges. For problems worked by hand, the answer is rather simple – just represent the non-existent edges by edges of an infinite length. This may represent a problem when writing a computer program to execute the algorithm.

If we take the infinite edge approach and we represent the matrix of edge weights (the generalized adjacency matrix) as.

$$W = \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix}$$

The problem here is how to represent infinity in the computer program. If we go with the IEEE standard floating point, we can represent infinity as follows.

Single Precision:	0x7F80 0000
Double Precision:	0x7FF0 0000 0000 0000

One should exercise some caution in using this approach because there is no guarantee that the CPU hardware will support this part of the standard. Also, can one say $X = \infty$?

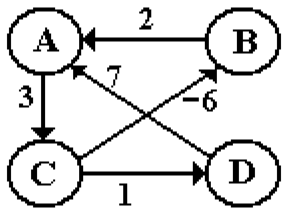
Another approach is to select a real number at the upper limit of the range of the data type being used to represent the weights. For the two more common data types, we might say

For 16-bit integers	Infinity = 32767
For single-precision floating point	Infinity = $3.5 \bullet 10^{38}$.

Again, this approach also might present some difficulties. Algorithms that make use of infinity as a working part can lead to unexpected difficulties when actually implemented.

These notes will follow the standard presentation of the algorithm and then suggest a more reasonable way to handle the infinity problem.

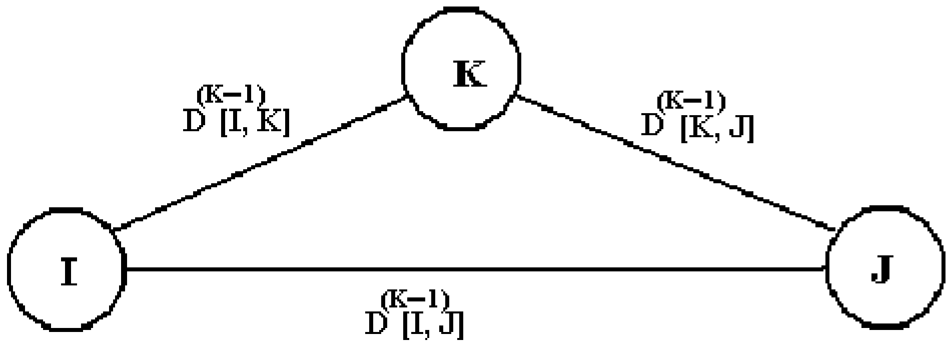
Before we investigate the algorithm, we should take note of a situation that the algorithm was not designed to handle. This is a graph, such as one at right, that has a cycle of negative total weight. Consider the cycle $A \to C \to B \to A$, with total weight of $2 + 3 - 6 = -1$. Of course, one should be suspicious of any graph having edges with negative weight.



The approach used in Floyd's algorithm is so similar to that used in Warshall's algorithm that some textbooks teach the two algorithms under the name Floyd–Warshall. The algorithm computes the distance matrix of a weighted graph with N vertices by using a sequence of (N + 1) matrices $D^{(0)}$, $D^{(1)}$, …. $D^{(N)}$, with $D^{(K)}[I, J]$ representing the shortest path from vertex I to vertex J with no vertex numbered higher than K being included in the path.

As in Warshall's algorithm, we consider two possible paths between vertices I and J

1) a path from I to J including no vertex numbered higher than (K – 1), and
2) a path from I to K including no vertex numbered higher than (K – 1), followed by vertex K, followed by a similar path from vertex K to vertex J.



Each path involves only vertices in the set 1 .. (K - 1)

Inspection of the above figure leads immediately to the algorithm for computing the K^{th} partial distance.

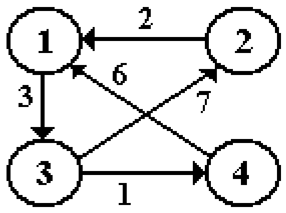
$$D^{(K)}[I, J] = \min(D^{(K-1)}[I, J], D^{(K-1)}[I, K] + D^{(K-1)}[K, J]), \text{ for } 1 \le K \le N$$

with $D^{(0)}[I, J] = W[I, J]$. Note also that vertex K cannot be a part of the path ending on itself, so that $D^{(K)}[I, K] = D^{(K-1)}[I, K]$ and $D^{(K)}[K, J] = D^{(K-1)}[K, J]$.

As in Warshall's algorithm, we use the last fact to avoid generating a sequence of matrices, and to write the next matrix in the sequence over its predecessor. The algorithm is below.

```
Algorithm Floyd ( W[1..N, 1..N] )
// Implements Floyd's algorithm for shortest paths
// Input:  The weight matrix of the graph
// Output: The distance matrix of the graph.
//
   D = W
//
   For K = 1 to N Do
      For I = 1 to N Do
           For J = 1 to N Do
                X = D[I, K] + D[K, J]
                If X < D[I, J] Then D[I, J] = X
           End Do // J
      End Do // I
   End Do // K
```

As an example, we shall investigate a graph that is isomorphic to the example above.



Two graphs are considered to be **isomorphic** if they have the same basic structure, with only superficial differences. This graph is the same as the original example, with the sole exception that the vertices are labeled with numbers, and not letters. This facilitates application of the algorithm.

We now consider the infinity problem as applied to this specific example. We make the observation that no path between two vertices may use more edges than are actually in the graph; thus it may not have a total distance greater than the sum of the weights of all the edges. Here the sum of the edge weights is 1 + 2 + 3 + 6 + 7 = 19. I pick any number larger than twice the total edge weight to represent infinity, here I choose 40.

In this example, I plan to show the sequence $D^{(0)}$, $D^{(1)}$, $D^{(2)}$, $D^{(3)}$, and $D^{(4)}$ explicitly.

$$D^{(0)} = W = \begin{bmatrix} 0 & 40 & 3 & 40 \\ 2 & 0 & 40 & 40 \\ 40 & 7 & 0 & 1 \\ 6 & 40 & 40 & 0 \end{bmatrix}$$

In the example below, I make use of an obvious inequality not found in the algorithm. All distances in the above example are non-negative, so that for any vertices we have

$D[I, J] \geq 0$ and $D[I, K] + D[K, J] \geq 0$,
so if $D[I, J] = 0$, then it is not possible to have $D[I, K] + D[K, J] < D[I, J] = 0$.

K = 1
This loop generates the matrix $D^{(1)}$. The operative code is the following.

```
   X = D(0)[I, 1] + D(0)[1, J]
   If X < D(0)[I, J] Then D(1)[I, J] = X,
                     Else D(1)[I, J] = D(0)[I, J]
```

I = 1, J = 1	$D^{(0)}[1, 1] = 0$, so no change.
I = 1, J = 2	$D^{(0)}[1, 2] = 40$.
	$X = D^{(0)}[1, \mathbf{1}] + D^{(0)}[\mathbf{1}, 2] = 0 + 40 = 40$, so no change.
I = 1, J = 3	$D^{(0)}[1, 3] = 3$.
	$X = D^{(0)}[1, \mathbf{1}] + D^{(0)}[\mathbf{1}, 3] = 0 + 3 = 3$, so no change.
I = 1, J = 4	$D^{(0)}[1, 4] = 40$.
	$X = D^{(0)}[1, \mathbf{1}] + D^{(0)}[\mathbf{1}, 4] = 0 + 40 = 40$, so no change.

Of course, we know that for I = 1 that $D^{(1)}[1, J] = D^{(0)}[1, J]$, so no surprise here.

I = 2, J = 1	$D^{(0)}[2, 1] = 2$.
	X = $D^{(0)}[2, \mathbf{1}] + D^{(0)}[\mathbf{1}, 1]$ = 2 + 0 = 2, so no change.
I = 2, J = 2	$D^{(0)}[2, 2] = 0$, so no change.
I = 2, J = 3	$D^{(0)}[2, 3] = 40$.
	X = $D^{(0)}[2, \mathbf{1}] + D^{(0)}[\mathbf{1}, 3]$ = 2 + 3 = 5, **a new value**.
I = 2, J = 4	$D^{(0)}[2, 4] = 40$.
	X = $D^{(0)}[2, \mathbf{1}] + D^{(0)}[\mathbf{1}, 4]$ = 2 + 40 = 42, no change.

Here we note one advantage of using very large finite numbers in this algorithm. The sum 2 + 40 has meaning and is equal to 42, while the sum 2 + ∞ has no meaning.

I = 3, J = 1	$D^{(0)}[3, 1] = 40$.
	X = $D^{(0)}[3, \mathbf{1}] + D^{(0)}[\mathbf{1}, 1]$ = 40 + 0 = 40, so no change.
I = 3, J = 2	$D^{(0)}[3, 2] = 7$.
	X = $D^{(0)}[3, \mathbf{1}] + D^{(0)}[\mathbf{1}, 2]$ = 40 + 7 = 47, so no change.
I = 3, J = 3	$D^{(0)}[3, 3] = 0$, so no change.
I = 3, J = 4	$D^{(0)}[3, 4] = 1$.
	X = $D^{(0)}[3, \mathbf{1}] + D^{(0)}[\mathbf{1}, 4]$ = 40 + 1 = 41, so no change.

We could have examined the third row of the matrix and shown that there would be no change in any element by noting that every computation of $D^{(1)}[3, J]$ involved adding a distance to $D^{(0)}[3, 1]$, which we have set to our representation of infinity.

I = 4, J = 1	$D^{(0)}[4, 1] = 6$.
	X = $D^{(0)}[4, \mathbf{1}] + D^{(0)}[\mathbf{1}, 1]$ = 6 + 0 = 6, so no change.
I = 4, J = 2	$D^{(0)}[4, 2] = 40$.
	X = $D^{(0)}[4, \mathbf{1}] + D^{(0)}[\mathbf{1}, 2]$ = 6 + 40 = 46, so no change.
I = 4, J = 3	$D^{(0)}[4, 2] = 40$.
	X = $D^{(0)}[4, \mathbf{1}] + D^{(0)}[\mathbf{1}, 3]$ = 6 + 3 = 9, **a new value**.
I = 4, J = 4	$D^{(0)}[4, 4] = 0$, so no change.

At this point, we have the following intermediate result.

$$D^{(1)} = \begin{bmatrix} 0 & 40 & 3 & 40 \\ 2 & 0 & 5 & 40 \\ 40 & 7 & 0 & 1 \\ 6 & 40 & 9 & 0 \end{bmatrix}$$

Before continuing the example, we present a revision of the algorithm that we shall adopt during the solution of this example.

```
Algorithm Floyd ( W[1..N, 1..N] )
// Implements Floyd's algorithm for shortest paths
// Input:  The weight matrix of the graph
// Output: The distance matrix of the graph.
//
   D = W        // Copy the W matrix, so we don't change it.
//
   For K = 1 to N Do
      For I = 1 to N Do
           If (I ≠ K) And !(D[I, K] == ∞) Then
                For J = 1 to N Do
                     X = D[I, K] + D[K, J]
                     If X < D[I, J] Then D[I, J] = X
                End Do // J
           End If
      End Do // I
   End Do // K
```

K = 2

This loop generates the matrix $D^{(2)}$. The operative code is the following.

```
   X = D^(1)[I, 2] + D^(1)[2, J]
   If X < D^(1)[I, J] Then D^(2)[I, J] = X,
                     Else D^(2)[I, J] = D^(1)[I, J]
```

Row I = 1. $D^{(1)}[I, K] = D^{(1)}[1, 2] = 40$, so no change on this row.

Row I = 2. I = K = 2, so no change on this row.

I = 3, J = 1	$D^{(1)}[3, 1] = 40$.
	$X = D^{(1)}[3, \mathbf{2}] + D^{(1)}[\mathbf{2}, 1] = 7 + 2 = 9$, **a new value**.
I = 3, J = 2	$D^{(1)}[3, 2] = 7$.
	$X = D^{(1)}[3, \mathbf{2}] + D^{(1)}[\mathbf{2}, 2] = 7 + 0 = 7$, so no change..
I = 3, J = 3	$D^{(1)}[3, 3] = 0$, so no change.
I = 3, J = 4	$D^{(1)}[3, 4] = 1$.
	$X = D^{(1)}[3, \mathbf{2}] + D^{(1)}[\mathbf{2}, 4] = 7 + 1 = 8$, so no change.

Row I = 4. $D^{(1)}[4, K] = D^{(1)}[4, 2] = 40$, so no change on this row.

At this point, we have the following intermediate result.

$$D^{(2)} = \begin{bmatrix} 0 & 40 & 3 & 40 \\ 2 & 0 & 5 & 40 \\ 9 & 7 & 0 & 1 \\ 6 & 40 & 9 & 0 \end{bmatrix}$$

K = 3

This loop generates the matrix $D^{(3)}$. The operative code is the following.

```
X = $D^{(2)}$[I, 3] + $D^{(2)}$[3, J]
If X < $D^{(2)}$[I, J] Then $D^{(3)}$[I, J] = X,
                  Else $D^{(3)}$[I, J] = $D^{(2)}$[I, J]
```

I = 1, J = 1	$D^{(2)}$[1, 1] = 0, so no change.
I = 1, J = 2	$D^{(2)}$[1, 2] = 40.
	X = $D^{(2)}$[1, **3**] + $D^{(2)}$[**3**, 2] = 3 + 7 = 10, **a new value**.
I = 1, J = 3	$D^{(2)}$[1, 3] = 3.
	X = $D^{(2)}$[1, **3**] + $D^{(2)}$[**3**, 3] = 3 + 0 = 10, so no change.
I = 1, J = 4	$D^{(2)}$[1, 4] = 40.
	X = $D^{(2)}$[1, **3**] + $D^{(2)}$[**3**, 4] = 3 + 1 = 4, **a new value**.

I = 2, J = 1	$D^{(2)}$[2, 1] = 2.
	X = $D^{(2)}$[2, **3**] + $D^{(2)}$[**3**, 1] = 5 + 9 = 14, so no change.
I = 2, J = 2	$D^{(2)}$[2, 2] = 0, so no change.
I = 2, J = 3	$D^{(2)}$[2, 3] = 5.
	X = $D^{(2)}$[2, **3**] + $D^{(2)}$[**3**, 3] = 5 + 0 = 5, so no change.
I = 2, J = 4	$D^{(2)}$[2, 4] = 40.
	X = $D^{(2)}$[2, **3**] + $D^{(2)}$[**3**, 4] = 5 + 1 = 6, **a new value**.

Row 3: I = K = 3, so no change on this row.

I = 4, J = 1	$D^{(2)}$[4, 1] = 6.
	X = $D^{(2)}$[4, **3**] + $D^{(2)}$[**3**, 1] = 9 + 2 = 11, so no change.
I = 4, J = 2	$D^{(2)}$[4, 2] = 40.
	X = $D^{(2)}$[4, **3**] + $D^{(2)}$[**3**, 2] = 9 + 7 = 16, **a new value**.
I = 4, J = 3	$D^{(2)}$[4, 3] = 9.
	X = $D^{(2)}$[4, **3**] + $D^{(2)}$[**3**, 3] = 9 + 0 = 9, so no change.
I = 4, J = 4	$D^{(2)}$[4, 4] = 0, so no change.

At this point, we have the following intermediate result.

$$D^{(3)} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 9 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

K = 4
This loop generates the matrix $D^{(3)}$. The operative code is the following.

```
X = $D^{(3)}$[I, 4] + $D^{(3)}$[4, J]
If X < $D^{(3)}$[I, J] Then $D^{(4)}$[I, J] = X,
                      Else $D^{(4)}$[I, J] = $D^{(3)}$[I, J]
```

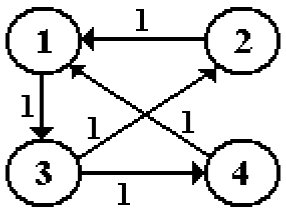
I = 1, J = 1	$D^{(3)}[1, 1] = 0$, so no change.
I = 1, J = 2	$D^{(3)}[1, 2] = 10$.
	$X = D^{(3)}[1, \mathbf{4}] + D^{(2)}[\mathbf{4}, 2] = 4 + 16 = 20$, so no change.
I = 1, J = 3	$D^{(3)}[1, 3] = 3$.
	$X = D^{(3)}[1, \mathbf{4}] + D^{(2)}[\mathbf{4}, 3] = 4 + 9 = 13$, so no change.
I = 1, J = 4	$D^{(3)}[1, 4] = 4$.
	$X = D^{(3)}[1, \mathbf{4}] + D^{(2)}[\mathbf{4}, 4] = 4 + 0 = 4$, so no change.
I = 2, J = 1	$D^{(3)}[2, 1] = 2$.
	$X = D^{(3)}[2, \mathbf{4}] + D^{(2)}[\mathbf{4}, 1] = 6 + 6 = 12$, so no change.
I = 2, J = 2	$D^{(3)}[2, 2] = 0$, so no change.
I = 2, J = 3	$D^{(3)}[2, 3] = 5$.
	$X = D^{(3)}[2, \mathbf{4}] + D^{(2)}[\mathbf{4}, 3] = 6 + 9 = 15$, so no change.
I = 2, J = 4	$D^{(3)}[2, 4] = 6$.
	$X = D^{(3)}[2, \mathbf{4}] + D^{(2)}[\mathbf{4}, 4] = 6 + 0 = 6$, so no change.
I = 3, J = 1	$D^{(3)}[3, 1] = 9$.
	$X = D^{(3)}[3, \mathbf{4}] + D^{(2)}[\mathbf{4}, 1] = 1 + 6 = 7$, **a new value**.
I = 3, J = 2	$D^{(3)}[3, 2] = 7$.
	$X = D^{(3)}[3, \mathbf{4}] + D^{(2)}[\mathbf{4}, 2] = 1 + 16 = 17$, so no change.
I = 3, J = 3	$D^{(3)}[3, 3] = 0$, so no change.
I = 3, J = 4	$D^{(3)}[3, 4] = 1$.
	$X = D^{(3)}[3, \mathbf{4}] + D^{(2)}[\mathbf{4}, 4] = 1 + 0 = 1$, so no change.

Row 4: I = K = 4, so no change.

The final result is thus

$$D^{(4)} = \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix}$$

Unweighted Graphs
If we consider a simple graph as a weighted graph in which all edges have weight 1, we can easily apply Floyd's algorithm.



The graph at left is a simple undirected graph in which each edge has been given a weight of 1. We can apply Floyd's algorithm.

The weight matrix for this graph is

$$D^{(0)} = W = \begin{bmatrix} 0 & 40 & 1 & 40 \\ 1 & 0 & 40 & 40 \\ 40 & 1 & 0 & 1 \\ 1 & 40 & 40 & 0 \end{bmatrix}$$

The distance matrix finally computed is

$$D^{(4)} = \begin{bmatrix} 0 & 2 & 1 & 2 \\ 1 & 0 & 2 & 3 \\ 2 & 1 & 0 & 1 \\ 1 & 2 & 2 & 0 \end{bmatrix}$$

Each weight in the distance matrix is the number of edges in the path from the source vertex to the destination vertex.