More Dynamic Programming
Floyd-Warshall Algorithm
(All Pairs Shortest Path Problem)

A weighted graph is a collection of points(vertices) connected by lines(edges), where each edge has a weight(some real number) associated with it. One of the most common examples of a graph in the real world is a road map. Each location is a vertex and each road connecting locations is an edge. We can think of the distance travelled on a road from one location to another as the weight of that edge.

Given a weighted graph, it is often of interest to know the shortest path from one vertex in the graph to another. The Floyd-Warshall algorithm algorithm determines the shortest path between all pairs of vertices in a graph.

Although I don't expect you all to understand the full derivation of this algorithm, I will go through some of the intuition as to how it works and then the algorithm itself.

The the vertices in a graph be numbered from 1 to n. Consider the subset {1,2...,k} of these n vertices.

Imagine finding the shortest path from vertex i to vertex j that uses vertices in the set {1,2, ...,k} only. There are two situations:

1) k is an intermediate vertex on the shortest path.
2) k is not an intermediate vertex on the shortest path.

In the first situation, we can break down our shortest path into two paths: i to k and then k to j. Note that all the intermediate vertices from i to k are from the set {1,2,...,k-1} and that all the intermediate vertices from k to j are from the set {1,2,...,k-1} also.

In the second situation, we simply have that all intermediate vertices are from the set {1,2,...,k-1}.

Now, define the function D for a weighted graph with the vetrtices {1,2,...n} as follows:

D(i,j,k) = the shortest distance from vertex i to vertex j using the intermediate vertices in the set {1,2,...,k}

Now, using the ideas from above, we can actually recursively define the function D:

$$D(i,j,k) = \begin{cases} w(i,j), & \text{if } k=0 \\ \min(D(i,j,k-1), D(i,k,k-1)+D(k,j,k-1)), & \text{if } k > 0 \end{cases}$$

In English, the first line says that if we do not allow intermediate vertices, then the shortest path between two vertices is the weight of the edge that connects them. If no such weight exists, we usually define this shortest path to be of length infinity.

The second line pertains to allowing intermediate vertices. It says that the minimum path from i to j through vertices {1,2,...,k} is either the minimum path from i to j through vertices {1,2,...,k-1} OR the sum of the minimum path from vertex i to k through {1,2,...,k-1} plus the minimum path from vertex k to j through {1,2,...k-1}. Since this is the case, we compute both and choose the smaller of these.

All of this points to storing a 2-dimensional table of shortest distances and using dynamic programming for a solution.

Here is the basic idea:

1) Set up a 2D array that stores all the weights between one vertex and another. Here is an example:

0	3	8	inf	-4
inf	0	inf	1	7
inf	4	0	inf	inf
2	inf	-5	0	inf
inf	inf	inf	6	0

Notice that the diagonal is all zeros. Why?

Now, for each entry in this array, we will "add in" intermediate vertices one by one, (first with k=1, then k=2, etc.) and update each entry once for each value of k.

After adding vertex 1, here is what our matrix will look like:

0	3	8	inf	-4
inf	0	inf	1	7
inf	4	0	inf	inf
2	5	-5	0	-2
inf	inf	inf	6	0

After adding vertex 2, we get:

0	3	8	4	-4
inf	0	inf	1	7
inf	4	0	5	11
2	5	-5	0	-2
inf	inf	inf	6	0

After adding vertex 3, we get:

0	3	8	4	-4
inf	0	inf	1	7
inf	4	0	5	11
2	-1	-5	0	-2
inf	inf	inf	6	0

After adding vertex 4, we get:

0	3	-1	4	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

Finally, after adding in the last vertex:

0	1	-3	2	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

Looking at this example, we can come up with the following algorithm:

Let D1 store the matrix with the initial graph edge information. D2 will stored calculated information look at D1.

```
For k=1 to n {
        For i=1 to n {
                For j=1 to n
                        D2[i,j] = min(D1[i,j], D1[i,k]+D1[k,j])
        }
        Copy matrix D2 into D1
}
```

Last D2 matrix will store all the shortest paths.

In order to code this up, we could do so in a static method. We need the adjacency matrix of the graph passed into the method as a two dimensional double matrix. Then we need the auxiliary min and copy methods.

As it turns out, you do NOT need to use 2 separate matrices, even though we traced through the algorithm in that manner. The reason for this is that when we look up values in the "current matrix", we know that those values will be at least as good as the values we would have looked up in the "previous matrix." Thus, in essence, we will not be overlooking any possible shortest path.

Here is the code for these methods as well as a main that runs this example given above.

```
public class floyd {

    // Runs Floyd Warshall algorithm. Returns the matrix of
    // shortest paths.
    public static int[][] shortestpath(int[][] adj) {

        int n = adj.length;
        int[][] m = new int[n][n];

        // Initialize m to be graph adjacency matrix
        copy(m, adj);

        // Add in each vertex as intermediate point.
        for (int k=0; k<n;k++) {

            // Recalculate estimate for distance from vertex i to j.
            for (int i=0; i<n; i++) {
                for (int j=0; j<n;j++)
                    m[i][j] = min(m[i][j], m[i][k]+m[k][j]);
            }
        }
        return m;
}

 public static void copy(int[][] a, int[][] b) {

     for (int i=0;i<a.length;i++)
         for (int j=0;j<a.length;j++)
             a[i][j] = b[i][j];
}
```

```
public static int min(int a, int b) {
     if (a < b)
         return a;
     else
         return b;
  }

  public static void main(String[] args) {

    // Hard code in example adjacency  matrix.
    int[][] m = new int[5][5];
    m[0][0] = 0; m[0][1] = 3; m[0][2] = 8; m[0][3] = 10000;
    m[0][4] = -4;
    m[1][0] = 10000; m[1][1] = 0; m[1][2] = 10000; m[1][3] = 1;
    m[1][4]=7;
    m[2][0] = 10000; m[2][1] = 4; m[2][2] = 0; m[2][3] = 10000;
    m[2][4] = 10000;
    m[3][0] = 2; m[3][1] = 10000; m[3][2] = -5; m[3][3] = 0;
    m[3][4] = 10000;
    m[4][0] = 10000; m[4][1] = 10000; m[4][2] = 10000;
    m[4][3] = 6; m[4][4] =0;

    int[][] shortpath;
    shortpath = shortestpath(m);

    for (int i=0; i<5;i++) {
        for (int j=0; j<5;j++)
            System.out.print(shortpath[i][j]+" ");
         System.out.println();
    }

  }
}
```

Path Reconstruction in Floyd's Algorithm

When you run Floyd's, what you're really doing is initializing your distance matrix and path matrix to indicate the use of no immediate vertices. (Thus, you are only allowed to traverse direct paths between vertices.)

At each step of Floyd's, you essentially find out whether or not using vertex k will improve an estimate between the distances between vertex i and vertex j. If it does improve the estimate here's what you need to record:

1) **record the new shortest path weight between i and j**
2) **record the fact that the shortest path between i and j goes through k**

#1 is done in the edited adjacency matrix. Hopefully this update (with the if statement) is fairly easy to understand.

Here is how #2 is done:

First, it's important to understand what the path matrix stores. In particular, when

path[i][j] = k, that means that in the shortest path from vertex i to vertex j, the LAST vertex on that path before you get to vertex j is k.

Based on this definition, we must initialize the path matrix as follows:

path[i][j] = i if i!=j and there exists an edge from i to j
= NIL otherwise

The reasoning is as follows:

If you want to reconstruct the path at this point of the algorithm when you aren't allowed to visit intermediate vertices, the previous vertex visited MUST be the source vertex i. NIL is used to indicate the absence of a path.

<u>Now, once the algorithm starts, here is how we update the path matrix</u>

If vertex k doesn't improve our path estimate, then we don't want to change our path at all and do not update the path matrix.

Now, let's say that vertex k improves our distance estimate in between i and j. We want to store the last vertex from the shortest path from vertex k to vertex j. *This will NOT necessarily be k, but rather, it will be path[k][j].*

Using this information, we have the following rule to update the path matrix during the k^{th} iteration of Floyd's:

$path_k[i][j] = path_{(k-1)}[i][j]$, if vertex k doesn't improve the estimate

$= path_{(k-1)}[k][j]$, otherwise

But, just as you only need to store one copy of the distance matrix while you run floyd's, you only have to store one copy of the path matrix as well! Thus, codewise, we'd do the following update:

if (D[i][k]+D[k][j] < D[i][j]) { // Update is necessary to use k as intermediate vertex

```
    D[i][j] = D[i][k]+D[k][j];
    path[i][j] = path[k][j];
}
```

Now, the once this path matrix is computed, we have all the information necessary to reconstruct the path. Consider the following path matrix (indexed from 1 to 5 instead of 0 to 4):

Nil	**3**	**4**	**5**	**1**
4	**Nil**	**4**	**2**	**1**
4	**3**	**Nil**	**2**	**1**
4	**3**	**4**	**Nil**	**1**
4	**3**	**4**	**5**	**Nil**

where the vertices are numbered from 1 to 5. To reconstruct the path from 1 to 2, for example, we look at path[1][2]. It has a 3. This signifies that on the path from 1 to 2, 3 is the last vertex visited before 2. Thus, the path is now, 1…3->2. Now, look at path[1][3], this stores a 4. Thus, we find the last vertex visited on the path from 1 to 3 is 4. So, our path now looks like 1…4->3->2. So, we must now look at path[1][4]. This stores a 5, thus, we know our path is 1…5->4->3->2. When we finally look at path[1][5], we find 1, which means our path really is 1->5->4->3->2.

One more example: find the path from vertex 2 to vertex 5.

Path[2][5] = 1
Path[2][1] = 4
Path[2][4] = 2

So, the path is 2->4->1->5.

Transitive Closure

The transitive closure of a directed graph G contains an edge in between each pair of vertices in G that are connected. (Connected means that there is a path in G in between the two vertices.) In essence, computing a transitive closure of a graph gives you complete information about which vertices are connected to which other vertices.

We can use a variant of Floyd-Warshall's algorithm for shortest paths to determine the transitive closure of a graph G as follows:

FloydWarshall(G)
1) $G_0 = G$
2) For k=1 to V do
a) $G_k = G_{k-1}$
b) For i=1 to V do
i) For j=1 to V do
if edge (v_i, v_k) and (v_k, v_j) are edges in G_{k-1}
Add (v_i, v_j) to G_k.
3) Return G_V.

This is the SAME as the other Floyd-Warshall Algorithm, except for when we find a non-infinity estimate, we simply add an edge to the transitive closure graph. Every round we build off the previous paths reached. After iterating through all vertices being intermediate vertices, we have tried to connect all pairs of vertices i and j through all intermediate vertices k.