Weighted Graphs: Shortest Distance Algorithms

Floyd-Warshall's Algorithm

This algorithm finds the shortest distance in a weighted directed graph between all pairs of vertices and runs in $O(V^3)$ time for a graph with V vertices.

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) = w(i,j), if k=0
min(D(i,j,k-1), D(i,k,k-1)+D(k,j,k-1)) if k > 0

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.

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

Here is a method that calculates the shortest distances given the adjacency matrix of the graph and doesn't destroy the original data:

```
public static int[][] shortestpath(int[][] adj) {

  int n = adj.length;
  int[][] m = copy(adj);

  for (int k=0; k<n;k++)
    for (int i=0; i<n; i++)
      for (int j=0; j<n;j++)
        m[i][j] = Math.min(m[i][j],m[i][k]+m[k][j]);

  return m;
}

public static void copy(int[][] a) {
  int[][] res = new int[a.length][a[0].length];
  for (int i=0;i<a.length;i++)
    for (int j=0;j<a[0].length;j++)
      res[i][j] = a[i][j];
  return res;
}
```

Path Reconstruction - Floyd-Warshall's

Create a path matrix so that path[i][j] stores the last vertex visited on the shortest path from i to j. Update as follows:

```
if (D[i][k]+D[k][j] < D[i][j]) {
  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

Using this matrix, we can construct the shortest 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.

Dijkstra's Algorithm

This algorithm finds the shortest path from a source vertex to all other vertices in a weighted directed graph without negative edge weights.

Here is the algorithm for a graph G with vertices $V = \{v_1, \ldots v_n\}$ and edge weights w_{ij} for an edge connecting vertex v_i with vertex v_j. Let the source be v_1.

Initialize a set $S = \varnothing$. This set will keep track of all vertices that we have already computed the shortest distance to from the source.

Initialize an array D of estimates of shortest distances. $D[1] = 0$, while $D[i] = \infty$, for all other i. (This says that our estimate from v_1 to v_1 is 0, and all of our other estimates from v_1 are infinity.)

While S != V do the following:

1) Find the vertex (not is S) that corresponds to the minimal estimate of shortest distances in array D. **Use a priority queue to speed up this step.**
2) Add this vertex, v_i into S.
3) Recompute all estimates based on edges emanating from v. In particular, for each edge from v, compute $D[i]+w_{ij}$. If this quantity is less than D[j], then set $D[j] = D[i]+w_{ij}$.

Essentially, what the algorithm is doing is this:

Imagine that you want to figure out the shortest route from the source to all other vertices. Since there are no negative edge weights, we know that the shortest edge from the source to another vertex must be a shortest path. (Any other path to the same vertex must go through another, but that edge would be more costly than the original edge based on how it was chosen.)

Now, for each iteration, we try to see if going through that new vertex can improve our distance estimates. We know that all shortest paths contain subpaths that are also shortest paths. (Try to convince yourself of this.) Thus, if a path is to be a shortest path, it must build off another shortest path. That's essentially what we are doing through each iteration, is building another shortest path. When we add in a vertex, we know the cost of the path from the source to that vertex. Adding that to an edge from that vertex to another, we get a new estimate for the weight of a path from the source to the new vertex.

This algorithm is greedy because we assume we have a shortest distance to a vertex before we ever examine all the edges that even lead into that vertex. In general, this works because we assume no negative edge weights. The formal proof is a bit drawn out, but the intuition behind it is as follows: If the shortest edge from the source to any vertex is weight w, then any other path to that vertex must go somewhere else, incurring a cost greater than w. But, from that point, there's no way to get a path from that point with a smaller cost, because any edges added to the path must be non-negative.

By the end, we will have determined all the shortest paths, since we have added a new vertex into our set for each iteration.

This algorithm is easiest to follow in a tabular format. The adjacency matrix of an example graph is included below. Let a be the source vertex.

	a	b	c	d	e
a	0	10	inf	inf	3
b	inf	0	8	2	inf
c	2	3	0	4	inf
d	5	inf	4	0	inf
e	inf	12	16	13	0

Here is the algorithm:

	Estimates	b	c	d	e
Add to Set					
a		10	inf	inf	3
e		10	19	16	3
b		10	18	12	3
d		10	16	12	3

We changed the estimates to c and d to 19 and 16 respectively since these were improvements on prior estimates, using the edges from e to c and e to d. But, we did NOT change the 10 because 3+12, (the edge length from e to b) gives us a path length of 15, which is more than the current estimate of 10. Using edges bc and bd, we improve the estimates to both c and d again. Finally using edge dc we improve the estimate to c.

Now, we will prove why the algorithm works. We will use proof by contradiction. After each iteration of the algorithm, we "declare" that we have found one more shortest path. We will assume that one of these that we have found is NOT a shortest path.

Let t be the first vertex that gets incorrectly placed in the set S. This means that there is a shorter path to t than the estimate produced when t is added into S. Since we have considered all edges from the set S into vertex t, it follows that if a shorter path exists, its last edge must emanate from a vertex outside of S to t. But, all the estimates to the edges outside of S are greater than the estimate to t. None of these will be improved by any edge emanating from a vertex in S (except t), since these have already been tried. Thus, it's impossible for ANY of these estimates to ever become better than the estimate to t, since there are no negative edge weights. With that in mind, since each edge leading to t is non-negative, going through any vertex not in S to t would not decrease the estimate of its distance. Thus, we have contradicted the fact that a shorter path to t could be found. Thus, when the algorithm terminates with all vertices in the set S, all estimates are correct.

The code for this is very similar to breadth first search and will be shown in lecture examples.

Bellman- Ford Algorithm

This algorithm finds shortest distances from a source vertex for directed graphs with or without negative edge weights. This algorithm works very similar to Dijkstra's in that it uses the same idea of improving estimates (also known as edge relaxation), but it doesn't use the greedy strategy that Dijkstra's uses. (The greedy strategy that Dijkstra's uses is assuming that a particular distance is optimal without looking at the rest of the graph. This can be done in that algorithm because of the assumption of no negative edge weights.)

The basic idea of relaxation is as follows:

1) Maintain estimates for distances of each vertex.

2) Improve these estimates by considering particular edges. Namely, if your estimate to vertex b is 5, and edge bd has a weight of 3, but your current estimate to vertex d is greater than 8, improve it!

Using this idea, here is Bellman-Ford's algorithm:

1) Initialize all estimates to non-source vertices to infinity.
   Denote the estimate to vertex u as D[u], and the weight of an
   edge (u,v) as w(u,v).

2) Repeat the following |V| - 1 times:

```
    a) For each edge (u,v)
            if D[u]+w(u,v) < D[v]
                    D[v] = D[u] + w(u,v)
```

The cool thing is that it doesn't matter what order you go through each edge in the graph in the inner for loop in step 2. You just have to go through each edge exactly once.

This algorithm is useful when the graph is sparse, has negative edge weights and you only need distances from one vertex.

Let's trace through an example. Here is the adjacency matrix of a graph, using a as the source:

	a	b	c	d	e
a	0	6	inf	inf	7
b	inf	0	5	-4	8
c	inf	-2	0	inf	inf
d	2	inf	7	0	inf
e	inf	inf	-3	9	0

Edge	Estimates	a	b	c	d	e
none		0	inf	inf	inf	inf
ab, ae	0	6	inf	inf	7	
bc, bd, be, ec, ed	0	6	4	2	7	
all, with cb	0	2	4	2	7	
all, with bd	0	2	4	-2	7	

The proof for the correctness of this algorithm lies in the fact that after each i^{th} iterartion each estimate is the shortest path using at most i edges. This is certainly true after the first iteration. Now, assume it is true for the i^{th} iteration. Under this assumption, we must prove it is true for the $i+1^{th}$ iteration. Notice that either the shortest path using at most i+1 edges uses at most i edges OR, is a shortest path of i edges with one more edge tacked on. This is because all shortest paths contain shortest paths. BUT, the way the algorithm works, ALL shortest paths of length i are considered, along with all edges added to them. Thus, the algorithm MUST come up with the optimal path using at most i+1 edges.