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.

Longest Path in a DAG

Since a DAG doesn't have cycles, there are well-defined longest paths in DAGs. We can use Bellman-Ford's algorithm to find the longest path in a DAG from a chosen source vertex. *Basically, just negate all the edge weights. Now, the negative of the minimum cost for a path in this adjusted graph is the same as the maximum cost path in the original graph!*

Another way to solve this problem would be to use memoization. Recursively, a solution looks like this, assuming that a path always existed:

```
int longpath(ArrayList[] graph, int source, int end) {

    if (memo[source] != -1) return memo[source];
    if (source == end) return 0;

    int best = Integer.MIN_VALUE;

    for (int i=0; i<graph[source].size(); i++) {

        int cur = graph[source].get(i).weight +
         longpath(graph, graph[source].get(i).vertex, end);

       best = Math.max(best, cur);
    }

    memo[source] = best;
    return best;
}
```

Each array list must be an array list of objects that have the instance variables weight and vertex that store those corresponding items for that connection from the source variable and the memo array has to be declared as a static class variable.

In essence, the code just says, "Try all paths that lead from source. The longest these paths could be is the sum of the edge from source to the next vertex, plus the longest path from that next vertex to our end vertex. Of all of these, take the longest!"