Lecture Transcript
Shortest Path Algorithm (Bellman Ford Algorithm)

Hi and welcome to this next lecture on Data Structures and Algorithms. We'll continue our discussion on Shortest Path Algorithms. In fact, we'll continue our discussion for shortest path with a fixed source. However, today we'll discuss an algorithm that can deal with negative weighted edges. Recall that we discussed Dijkstra's algorithm, which operates under the constraint that no edge weights are negative. So the Bellman-Ford Algorithm, it computes shortest path from a single source, fix source to all other vertices in a weighted directed graph. However, Dijkstra's algorithm does not work for negative weighted edges. It's a greedy algorithm that makes use of the optimal substructure property. Bellman-Ford solves the problem of negative weighted edges, but it's slower than Dijkstra's algorithm that's a price to be paid and the running time is the order of a number of a vertex is to the number of the edges. So what is the idea?

So let us understand the algorithm with a simple example. So we'll try and illustrate the working of the Bellman-Ford Algorithm through an example. So let's say you've graph with a source S, we'll consider purely directed graph. Let's give names to these vertices V(sub)1, V(sub)2, V(sub)3, V(sub)4, V(sub)5 and V(sub)6. We'll also give names to the edges and their weights W(sub)1, W(sub)2, W(sub)3, W(sub)4, W(sub)5, W(sub)6. It also helps to have an additional edge here to make the graph a directed acyclic graph, so just let say this is W(sub)7. So, Bellman-Ford Algorithm iterates through all the edges of the graph and for every edge, it determines the weight on the two nodes that it's part of, so of course if you have the edges to be undirected. It's possible that shortest path crosses that particular edge in either way in either direction. However, edge is directed you expect the shortest path from one node to the other node if at all to occur only along the direction that is permitted by the edge. So the idea behind Bellman-Ford Algorithm is initially all vertices are infinitely far away from the source, the source is of course, at distance zero from itself.

Now you iterate over all edges for each edge 'e', you're going to check if the let say 'e' is going from u to v, you'll check if the distance a shortest path distance to v existing distance is greater than equal to a distance to u plus the weight on the edge u v. If this is the case, then you update d[v] to this new-found shortest path, it says I am going to look at the shortest path to u and then append the edge (u,v) to the shortest path to u. So this is the basic idea and you are going to do this for every edge. How many times will you have to do? Well, when you start the iterations initially you determine that for V(sub)1 and V(sub)4 or that is the edge is W'(sub)1 and W(sub)4. You are able to update the weights for V(sub)1 and V(sub)4 to say W(sub)1 and

W(sub)4 respectively. However, for all the other vertices other than source of course, in the first iteration over all the edges there is nothing to update. However, in a subsequent iteration, you will also update the shortest path for V(sub)2 say which is W(sub)1 + W(sub)2 and that for V(sub)5 which is W(sub)4 + W(sub)5. So you will have to iterate over all the edges number of times that equals the maximum number of hops to a node from S. So, number of times to iterate over all the edges, what I mean by this is this iteration.

This is the width of the graph, which is max number of hops from source node S to any vertex, say this let's say V, by the way, we don't need V(sub)6, V(sub)3 this going to be V(sub)3. So, the complexity is basically number of edges into this maximum number of hops. But I will convince you that this maximum number of hops in the worst case actually happens to be a number of vertices minus 1. I'll construct an example to show this. So let's consider a simple linear graph S goes to V(sub)1 goes to V(sub)2 goes to V(sub)3. So my first scan over the edges I'll be able to update V(sub)1 the second scan I'll be able to update V(sub)2 the third will be V(sub)3. So if there are four vertices and I need three scans over all the edges till I am able to reach V(sub)3 and update it's shortest path. Though I can update the shortest path for V(sub)3 when the shortest path for its immediate neighbors are updated. Let's look at the actual algorithm as pointed out the first thing you do is iterate over all the vertices set the depth of every vertex the shortest path depth infinity predecessor for all the nodes is not set but the predecessor for the source is null and the depth for the sources is zero. And then as suggested, we are going to iterate over all the edges these in a loop iterate over all the edges and we're able to do that for a number of times that equals number of vertices minus one.

So, recall that this is in the worst case the so called width of the graph and then it's a check that we already anticipated if d[u] + w, w is the weight of edge e. If this happens to be less than the existing shortest path distance to V then you update the shortest part distance at V. You also update the predecessor of [v] to point to u. So, this is already motivated and explained, what is the path here at the end? Well, so far we have assumed that our graph is a directed acyclic graph. What if the graph has cycles? And is therefore not a directed acyclic graph abbreviated of as [DAG]. Even in the case that the graph has no directed edges it might just have cycles and here is the test. If the shortest path can be further reduced even after V minus 1 iterations then it appears at there is a cycle that is there is a path from S to a node any particular node that involves a cycle. Let's illustrate this with an example, so we have S, V(sub)1 let say this is weight 5 this is say 4 to V(sub)2, V(sub)3 minus 3. We already pointed out the negative weight at edges are possible and say minus 2, minus 1. Now as you update the weights originating with S, 0, V(sub)1 is 5, V(sub)2 is 9, V(sub)3 is 6 and then you have 4 and then you have -1, 3. However, the next iteration when you update V(sub)2 we will get 7 and then you will be able to update V(sub)3 to 4 and that at this node V(sub)4 here we able to update the weight for V(sub)4 further and get it to 2 and then at V(sub)1 get it to 1. You could do this again go over this 1 + 4 is 5, 5 - 3 is 2, 2 - 2 is 0, 0 - 1 is -1 and so on.

So, just by going around this loop you can actually get very, very negative weight at paths from S to all of these nodes. So, what is a characteristic here? Well, it turns out that this cycle itself a negative weight. What is a weight of all paths on cycle (-1) + (4) + (-3) + (-2) the whole weight is basically -2. So the more you go around in this loop. The more you will be able to reduce the shortest path in this you can do add infinity term. So this is what we are checking if after so many

iterations if you're still able to reduce the shortest path distance to any particular node then you basically can't find the shortest path. So, you just return a false that's what we see here. Returns false for negative weight cycle. Let's see this in action, we start with infinite weighted shortest path to all the nodes. In the first iteration over all the edges, we find that only the immediate neighbors of the source can be updated and we've done that. Then again we look at all edges, we're only able to update now immediate neighbors of these nodes that were updated.

So, it turns out that b is also neighbor to a and you're able to get shortest path to be through a. This is the latest. We continue our book-keeping. Find that the shortest path to c and d has been updated. c and e has been updated and then to g and it turns out that you can actually find another shortest path to g through c. And in fact, this also lets you update the shortest path to e through g. So, let us analyze this bellman shortest path algorithm. We look at two characteristics, one is its correctness and the other will be its complexity. So, we prove the correctness of this algorithm by claiming the following loop invariant and you set that for every vertex v within k hops from the source node S. The value d[v] is indeed the shortest path from S after k outer iterations. So, we're claiming that at the kth iteration of this outer loop, T value that gets updated in d[v] at the end of this kth iteration. d[v] contain the correct value, the shortest path. And it's easy to show that at initialization this is correct because the only node that is within zero hops from the source itself and indeed the depth of shortest path of source is set to zero. The next iteration we can easily see that we only deal with the immediate neighbors of the source and they get updated correctly. In fact, the maintenance can be easily proved because we know that to get two vertex which is k + 1 hops from the source S. You need to go through a vertex within k hops from the source S.

So, maintenance can be proved with the key observation that a vertex at k + 1 hops from S is reached through vertex at k hops from S. And this can, therefore, extend even at termination. What about time complexity? Well, the run time complexity is again straightforward as we anticipated we're going to iterate over all the edges for a fixed outer iteration and number of outer iterations is basically the number of vertices of order of number of vertices . We're actually split the cost here for each of the if blocks and the for loops and so on. But it essentially number of edges into number of vertices. In the final check is only order of number of edges which is basically to check for negative weight cycles. So over all complexity is order of V E is the dominating term.

Thank you.