Lecture Transcript
Shortest Path Algorithm (All Pair Shortest Path)

Hello and welcome to this next session on Data structures and Algorithms. We'll continue our discussion on Shortest Path Algorithms. Today, we will introduce you to an algorithm that finds shortest path between all pairs of vertices in a graph. So far, the Bellman-ford and the Dijkstra's algorithm that we discussed are based upon a single source vertex. What if you want to find the shortest path between every pair of points? So given a directed graph on each edge v1, v2 and E, you have an associated weight. The problem is to find a shortest path from v1 to v2. We make an interesting observation here before we discuss the exact solution. The observation is that, if d[v1, vk] is shortest path from v1 to vk and d[vk, v2] is a shortest path from vk to v2 then we expect the following: we expect that, if v1 to v2 are directly connected in a graph then the associated weight w, v1, v2 could be the shortest path or the sum of these two could be the shortest path.

So we would expect that d the shortest path from [v1 to v2] is the min of d[v1 to vk] if from and then from [vk to v2] and if there as an edge v1 to v2 you might also want to look at weight of v1 to v2. However, who will give you this shortest path from v1 to vk at the outside, we won't have this. What we of course know are these weights to begin with and instead of looking at the minimum weight from v1 to vk and then vk to v2, what we will do is iteratively construct the weights from v1 to vk and vk to v2 restricted to the first k-1 nodes and that we will denote with this superscript k-1. So what we are going to exploit is that the shortest path from v1 to vk using the first k-1 nodes using the nodes 1 or more precisely the set of nodes 1 to k-1. Similarly, if vk-1, vk to v2 is the shortest path from v1 to vk using nodes 1 to k-1 then we will estimate the shortest path from v1 to v2 using the first k nodes dk, v1, v2 as the min of the sum of the distances using k-1 nodes from v1 to vk and vk to v2 respectively or it could be the direct weight on the edge v1 to v2. This is the basic idea behind the all pair shortest path algorithm.

This has a very specific name, this is called the Floyd-Warshal Algorithm. In fact, this is the first instance of an algorithmic paradigm called dynamic programming. So this is instance of an algorithmic paradigm. What is dynamic programming here? Well, we are actually solving larger problems in terms of smaller problems. So we keep track of distances using up to k - 1 nodes and use that to keep track of distances up to k nodes this it goes on. A substructure gives you solution to the superstructure. So, let's set up the notation before we give you the complete algorithm. So given this weighted graph, we want to represent the weight of each edge in an adjacency matrix representation. So, W is the adjacency-matrix representation. So w between i and j is a weight of nodes of the edge vi to vj. It set to 0 if i equal to j, if i, j is an edge then wvivj is nothing but the actual edge weight and it set to infinity if there is no direct edge between i and j.

Now the distance matrix that we talked about is initialised to W. And, what is D0? D0 is nothing but the shortest path matrix for all pairs with path through, so recall we talked about nodes 1 to k - 1 but here we have k less than 1. So this is basically path through the empty set. We'll also do book keeping using a predecessor matrix. So this is book keeping for shortest path. What this does is, if i, j is indeed an edge then the predecessor for j is i. However, if there is no such edge i, j then the predecessor is set to null, this is the initial configuration. Thereafter, we are going to invoke the dynamic programming update step that we have already motivated. So here is the algorithm with the initialisation that we've already discussed. We are going to iterate through every node, we are assuming that the nodes are numbered 1 to n. So for this, k value will be 1, 2 until n. So, Dk is a new n by n matrix that we want to learn, and this corresponds to the desired shortest path matrix which passes through nodes 1 to k. And, how do you find that? Well, for every node i and every node j we will let ourselves explore the shortest path from i to j that passes through one of the k-1 nodes, one or more of the k-1 nodes, this is the shortest path from vi to vj through one or more of 1, 2, k-1 nodes. With these indices so you might want to specifically write them down as v1, the nodes number v1 to vk-1.

The other possibility however is that there is a shortest path which includes k and therefore we compare this shortest path through v1 to vk-1 with the new shortest path, and this is the shortest path from vi to vk through v1 to vk-1 plus the shortest path from vk to vj, this is again passing through one of these nodes, one or more of these nodes. What are we saying? We are saying that the shortest path vi to vj either passes through vk or doesn't pass through vk but the nodes v1 to vk-1. This is no point in repeating vk more than ones, because if that happens you would have a cycle, and in fact we will see that a cycle with negative weights is undesirable will need to infinite shortest path cost or weights and that is something we will actually discover at the end of the algorithm. So for the time being we are talking about the desirable case where vk doesn't repeat. What is the complexity of the algorithm where you are going to iterate over k which varies from 1 to n, which is nothing but more than ones, because if that happens you would have a cycle. And in fact, we will see that a cycle with negative weights is undesirable will lead to infinite shortest path cost or weights and that is something we will actually discover at the end of the algorithm. So for the time being we are talking about the desirable case where vk doesn't repeat. What is the complexity of the algorithm where you are going to iterate over k which varies from 1 to n, which is nothing but a number of vertices, and you are going to do that thrice so it basically —V— into —V— into —V—, so each of this is order V. So this is basically an order —V— cube algorithm.

Let's see this algorithm in an action. So this is the graph with k=0, where D0 is just set to W, and what is W? If there is direct edge, we have the corresponding weight else you set the weight to infinity, the weight, the shortest path from a node to itself is 0, has weight 0. What next? So we are going to assume that a corresponds to k=1, b corresponds to k=2, c is k=3 and d is k=4. So we start with k=1. To remind you k=1 corresponds to the node or vertex a, and we'll start with vi=b. We are certainly not interested in the row corresponding to a, because we are looking at paths going through a. So we look at 'b'. Now the scan the row corresponding to 'a' that is this row this is the column under consideration and we scan the row corresponding to Vi and we will basically vary j. So j will be scanned for, in this row and what are going to see. We going to basically check for different values of j whether 'v' 'i' 'k' + 'v' 'k' 'j' is less or 'v' 'i' 'j' is less.

So we start with v, i , j where j = a, vj = a and we find that well 6 + 0 is certainly equal to 6 itself, so we persist with the 6. Again we're not interested from b to b. b to c well we're going to look at the shortest path from b to a and from then a to c.b to a is 6. a to c is 4 and these gives you the value of 10. So this is actually less than d to c the earlier value of d to c which was basically infinity. You can do the same thing for b to d but there is no path through a so this is the only update for vi = b and k = 1. Then loot at k = 2 so basically for k = 1 there's nothing else that's possible. 'c' 'a' is infinity and 'd' 'a' is also infinity. k = 2 we started with vi = a so recall we're going to decide on the column based on k and that happens to be this one because this corresponds to b and row will correspond to vi = a. This is a row and we're going to now scan for j in this row. So we certainly we'll start with c, vj = c and that will mean comparing ab that is 12 with bc that is 10. So 12 + 10 is 22 which is actually greater than existing weight 4 on aj the dij. So we leave this unaltered then we look at the next one 12 + 8 and we compare that again ad which was actually infinity. So this leads to an update. So we continue with k = 2 which corresponds to b and we'll change our choice of i to the third row.

This will correspond to vi = c and as before we're going to scan this from left to right for different values of j. Our comparisons again reveal the following, so we look at cb + ba. cb is 3 and ba is 6. So this leads to an update of 9 for ca. We also look at cb which is 3 plus bd which is 8 and that is 11 and that's actually less than the earlier value of cd which was infinity. So this gives you two updates for vi = c. Continuing with k = 2, we now turn on attention to the last row. We again perform a scan for different values of j and we compare db with ba. db is 10. ba is 6. And compare with the earlier value of da. The earlier value of da was infinity and the 16 is certainly less than infinity. This is an update. There is actually nothing to do for the last two because weights are always, anyways going to be smaller than what we would get using any possible update relaxation. So this was for k = 2 for k = 3, so this is the choice of c.

Now I've shown one example, now we're started looking at longer now we'll start looking at longer hauls. vi = a, vj = b. We're going to quickly jump to our results so we look at the existing shortest path between a and b which was 12 and then we look at the shortest path from a to c which is 4. The shortest path from c to b which is 3 and find that well 7, 4 + 3 is certainly less than 12 so we'll update. So you go on up to end. We've just change the value of j, vj = d and you compare ac 4 + cd which is 11 and contrast as with the earlier value of ad, ad was 20 so 4 + 11 less than 20. That you thus you update 15. So this is as far as the illustration goes how do you show the correctness of this algorithm? So one can show the correctness of this algorithm using the following loop invariant. You can show that the following loop invariant is initialise maintained and wholes a termination which is what we started off with that d (k - 1) (vi, vj) is shortest path from vi to vj using exactly one or more of one to k - 1.

Thank you.