Lecture Transcript
Shortest Path Algorithm (Dijkstra's Algorithm)

Hello and welcome to this next lecture on Data Structures and Algorithms. In today's session we will discuss a shortest path algorithm, we'll discuss the simplest shortest path algorithm which is based on a single source and which happens to be greedy yet optimal. It's also known as Dijkstra's Algorithm. So, we will discuss shortest path in graph that are weighted and the underline property of some of these algorithm in particular, Dijkstra's Algorithm makes use of a property called the optimal substructure property. And finally we discuss the procedure of edge relaxation. What is the shortest path in weighted graphs? So, given two vertices 'a' and 'g' in this graph below. The problem is to find a path that has minimum total weight between them.

So we have assumed non negative weights. Weights are 7, 7 and 9 along this particular path, a 23 is a total weight of this path. There are also other paths one through d which actually offers slightly different weight 22, so which of these paths will be the shortest. The length of a path is sum of weight of the consequent edges and this has several applications such as routing of vehicles. The weight could denote length of a particular road. It could also denote the time required to get from 'a' to 'b' based on congestion conditions and so on. One could look at flight routing, internet packets in an IP network and so on. Two specific properties that we would like to exploit are as follows. Optimal substructure, given the path between 'a' and 'g. What we would like to discover is subpaths and we would like the subpaths to also correspond to shortest paths . So the subpath of a shortest path is also a shortest path. In fact, now starting with a source node 'a' we can show that a tree of shortest path exists from any start vertex s to every other vertex. So for example, the shortest path from 'a' to 'g' here happens to have this weight 11. Every other path has higher weight.

However, the shortest path to 'f' is indeed through 'd'. So 'a', 'd', 'f'. You don't need to connect 'f' to 'g' as part of the shortest path tree because that's not the shortest path. So what we are developing is an intuition that this subgraph containing all the shortest paths will actually be a tree. It cannot have cycles, primarily because we're talking about unique shortest paths and we're only talking about finding one shortest path. So the optimal substructure property is exploited in a greedy algorithm which is a Djikstra"s Algorithm that the subpath of a shortest path is also a shortest path. So, Dijkstra's Algorithm combines the optimal substructure property with the tree of shortest paths which it keeps track of. So for every node or vertex 'v' it stores the label d(v), which is the distance of 'v' from the source 's', and computes d(v) successively starting from the neighbours of 's'. We are of course assuming that the graph is connected.

We can make small modifications to this algorithm to deal with graph which consists of several connected components, and that we'll leave as a homework problem, to deal with containing several connected components. We are also assuming that edges are undirected, we will leave it is a homework problem to determine how directed edges could be considered.

How to deal with a mixture combination of directed and undirected edges. And finally very critical for our algorithm is that the edge weights are nonnegative. So here is the algorithm by Dijkstra's, given an input graph 'g' and the source node 's'. Dijkstra's keeps track of the shortest path length from 's' to every possible node 'v'. So, SP is basically evolving in size, every time we find a node with a shortest path, we add it to SP. And SP is actually critical to our loop invariant. We'll also keep track of the distance to every possible node in the graph that has been computed to be the minimum so far, so P is an interim state and it's maintained as a min heap. So this is the interim state, the distance that has been found to be shortest so far, but this has populated for every node, in fact, that's what we do here. So, initially we keep track of the shortest path to the source node zero and for every other node we set that to infinity, and for every node we go ahead and set the value of the shortest path so far to be SP that is basically infinity for all nodes but the source node. So, 'P' is a priority queue and we pick the minimum element from 'P' within this loop, 'u' is that minimum which means 'u' is the closest node in 'P' from 'S'. So initially what you'll get from 'P' is the source node itself, so you get rid of that node, the source node, look at all the incident edges and for every incident edge 'e' you find the other vertex 'w'. So pictorially what you've done is found 'u' that has been found to be the shortest from 's' so far, look at all its incident edges, here's one particular 'e' which has 'w'. We compute the shortest path SP from 'S' to 'u', to that we add the weight of edge 'e'. This is the step called relaxation.

So we are basically relaxing 'e', absorbing it into the path to 'w' from 'S'. Now, if this relaxed weight 'r' turns out to be less than the shortest path to 'w' computed so far, we conclude that 'r' is the relaxed weight is indeed the shortest path. So we set SP[w] = r, and we update the same within P. So note that the shortest path for all the nodes was initially set to infinity except for the source node. So, the first time you hit upon a node that has been visited for the first time, obviously the shortest path to that node will be infinity and you'll plan updating it. Question is whether you will get to update the weight for a node again, and that's precisely the loop invariant. The loop invariant for the Dijkstras algorithm is as follows. So for every node that gets removed from 'P', we can be sure that we've actually found the shortest path to that node. So loop invariant is for each node 'u' that ceases to remain a part of 'P', the shortest path to 'u' is indeed SP[u]. So, in fact, that is why we keep iterating till 'b' becomes empty. And we can easily prove this as follows. So, the last time that 'P' was updated for that particular node 'u', we computed a new shortest path to 'u' based on its neighboring node, been found to have the shortest distance from the source 'S'. And indeed it was found that the new path to 'u', the relaxed path to 'u' was shorter than the earlier path that was registered for 'u'.

So this just means that we can actually not find a shorter path to 'u' than what has already been registered. One can very formally provide the proof for this loop invariant that the loop invariant is actually satisfied at every iteration using contradiction, it is a very straightforward proof. So, let's see the Dijkstra algorithm in action. We start with a source node 'd' and we look at all its neighbors 'a', 'b', 'e' and 'f'. So recall that the shortest path to 'd' is registered to be 0, whereas shortest path to every other node 'a', 'b' and so on is basically infinity. And this is exactly what

the priority queue, the min heap is going to carry in the beginning. So, P.min is just going to look at 'P', so P.min will actually turn out to be 'd' itself. Next you scan the neighbours of 'd' 5, 9, 15 and 6 are the weights on the paths to the neighbours. So you will update the weights, SP[a] will be 5 + 0, which is less than existing infinity. So SP[a] is 5, SP[b] is 9, SP[f] is 6, and the shortest path to 'e' is 15. You are going to update the min heap, so what happens. And the next P.min should get you 'a' with its corresponding value 5. Thereafter you would again go ahead and update all its neighbours, let's see what that mean.

So, yes, we got 'a' and now what you have done is updated the weights of its neighbours, you find that the new path through 'a' gives a weight of 8 which is less than the earlier value of 9. 'a' has only one neighbour which has not yet explored because the other neighbour is 'd' itself. Now look at the min heap. What is the next element to pick? Well, the next element to pick is 'f' with a value of 6, so you pick 'f'. What do you do next is scan its neighbours, look for their shortest paths and see if the relax path is better than a shortest path so far. So 6 + 8, 14 for 'e' is less than 15. For 'g' again we find 6 + 11, 17 to be less than infinity, so you will have 17 and 14 respectively. The next element however has to be picked up and that is 'b' with weight of 8. Again you go ahead and scan its neighbours and you'll find that 8 + 4, 12 is less than 14 so you could update the shortest path for 'e' and likewise for 'c', it's 16 to be updated. And the next shortest node from the min queue, the priority queue is going to be 'e' with a weight of 12, scan its neighbours, is there need to update the weights in the neighbours? Well, 17 was a 16, 16 is already shorter, 12 + 4, 16, yes, you can update 'g' and so on. So again, we recall that the loop invariant was that every node removal from the priority queue indeed has a shortest path, and that's what we've been doing. We have been actually moving forward in a greedy manner and that is courtesy and never emptying P.

So loop invariant and the greedy nature of the algorithm are very closely tied through the ever emptying P. Analysis, so let's look at how much time it takes to initialize the shortest path for every vertex it's just a scan over all the nodes. So, it's c1 times order V is linear in the number of vertices. Now, for every scan for getting the min element of 'P', you will incur log times order V times, so worst case when 'P' is complete, of course, subsequently 'P' gets depleted. This is using the standard min heap implementation. Once you remove that element you are going to scan all the neighbours of 'u', that is going to be cumulated across all the nodes. So we have already accounted for that 'c4' times a degree of 'v', the neighbour of 'v', and this is going to happen across for every node. And then we also have an update call where you update the weights corresponding to all the neighbours. So this overall is basically for every node so log—V— times. So the multiplication is between this log—V— and the degree summed across all the nodes. So more specifically the time required is c3 into c4, the multiplication of these two log of degree of v times and this is basically summed over all the nodes of degree of v.

So please note that you have accounted for the worst case here and made it independent of v. We could make it a little bit more complicated and make this dependent on the node, the size of 'P', however that is needless because we can always come up with a worst case complexity in the specific case. So, we have this summation over all the nodes, degree (v) plus well the overall cost incurred across all the outer iterations which is across all the nodes. So overall what we get is a summation of degree V times log —V— + log —V— times the size number of vertices themselves. So this is order of E + V times log —V—.

Thank You.