Lecture Transcript
Spanning Tree Algorithm (Prim's Algorithm)

Hi and welcome to this next session on Data structures and Algorithms. In this session, we'll discuss another problem with graph and that is of finding spanning trees. A tree is basically a sub-graph which spans all the vertices of the original graph but using only a subset of the edges in such a way that there is no result and cycle. We'll discuss Prim's Algorithm for finding a spanning tree for a graph. However, it's possible that a graph has weighted adjust. So in such a situation you would be interested in finding a tree that has the smallest sum of weights over its edges and that basically amounts to finding a minimum spanning tree. So minimum spanning tree T is basically a subset of the graph G with min value of summation over all the vertices, more specifically over all the edges E of a tree of the weight of the edge, such that for every vertex V in G there exists and E in T that covers V. We'll be considering weighted undirected graphs.

Now the first algorithm we'll discuss is the Prim's Algorithm. It's a greedy algorithm, very much like the Dijkstra's Algorithm. Recall, that the Dijkstra's Algorithm also discovers a tree. However, Dijkstra's Algorithm is for a single source and the goal there is to find the shortest path from every node to that source. Of course, the Dijkstra's Algorithm assumes that all edge weights are non-negative. So here's the basic idea behind the Prim's Algorithm. It is greedy in the sense of finding the next vertex to add which has the smallest edge weight to any of the existing vertices in the sub-tree that is being grown so far. So the key idea is as follows : 1) Grow the minimum spanning tree T, as you grow, 2) keep track of the smallest edge weight from each vertex V which is in the set of vertex of G, but not in a set of vertices of T to a vertex W that is part of the existing minimum spanning tree T and the greedy step is 3) Add V from V(G) - V(T) to the tree T with smallest value of such edge weight.

So let's try and map the key ideas into the algorithm described here. We are going to keep track of the smallest edge weight from each vertex V(G) - V(T) to a vertex in V(T) and basically this will form what we call the array of keys, so here is the key and we initially set the key value for every vertex to infinity. However, we need some route. We can arbitrarily choose a route because we are dealing with undirected graphs, so arbitrarily choose a route. So, all other key weights are large.

Now, we are going to iterate over the set of vertices in P, so make a note here that forevery vertex in G we add it to P. P is basically maintained as a min heap. However, at initialisation, you specify that the key that the min heap should use is this array is based on this array that

element the next smallest element should be retrieve. So, now you iterate over this min heap, get the smallest element from the min heap and that corresponds to the step number three, add vertex V to T with thesmallest value of such a weight and that's exactly what we do here. However, once we have identified the next node to add and initially we'll actually addrout r. You're also going to update the key values for nodes or vertices that adjacent to u. So this step which iterates over all the adjacent vertices basically corresponds to the step of update, updating the key array and this is done for adjacent vertices to u. You also do a small book keeping here through the predecessor. The predecessor array is what gives you access to the edges that are included in minimum spanning tree.

So, for all (u,v) such that predecessor of u is v or predecessor of v is u, edge (u,v)will belong to the edge set of T. What ensures correctness of this algorithm? Well, the loop invariant for this algorithm is as follows. You can guaranty that in each iteration of the while loop the vertices that are already placed into the minimum spanning tree are basically those which are not in the priority queue P but are in V. So the loop invariant has a bunch of conditions, one of them is that vertices in V(G)minus those in the priority queue P are already part of the minimum spanning tree, of course, we keep track of the minimum spanning tree to the predecessor and a termination will what does it mean, it means that P is empty, you have actually spanned all the vertices of the graph. So you've basically completed the minimum spanning tree. There are also couple of other conditions which basically ensure that you have added edges which should actually belong to the minimum spanning tree. So, for all vertices v which are part of the priority queue P. If the predecessor of v is not nill, then the key corresponding to [v] should be finite and key [v] is weight of the edge going between v and pi v, where pi v is basically already a part of the minimum spanning tree constructed so far.

So, one can again ensure or prove that this condition holds, this second condition was the motivation behind the Prim's Algorithm in the first place. Let's see Prim's algorithm in action. We are going to arbitrarily choose a root. Let's say we choose 'a' to 'b' the root. The what this is, that are adjacent to 'a', will have their keys updated. So, 'b' will have its keys updated, the smallest path to 'a' is actually 4th. The smallest path to any node in the existing spanning tree is 4. So, this is how you grow the tree starting with the root 'a' and you add 'b' as well. What do you do next, is update the key for all the other nodes and here are the new keys. 'h' has the shortest but 'x' to 'a' as compared to that to 'b'. So, will basically select will update this. 'c' has a part to 'b' but has no direct edge to 'a'. but all the other what this is, there is actually no direct edge incident on the node added so far to minimum spanning tree. So, the next step all you can do is, choose between 'c' and 'h'. Arbitrarily break pi and choose 'c'. What are the new keys to be updated? So we can update the key for 'i', we can update the key for 'd' and for 'f'.

Now, amongst all the new keys and the old keys, we find that 'i' should be pignets that what is done. Again you update the keys for other nodes connected to the 'i' and we find that in the next iteration it makes sense to include 'f'. So, we go ahead and again update keys 'g' has a new key and looks like 'g' should be picked up next. Again you update keys well certainly 'h' can be picked up next. The some nodes still an unexplored and refined the amongst though unexplored nodes, we can pick 'd' next because it has the smallest key 7 and then amongst the rest 'h' is already part of the vertex schedule not added. Only thing to consider is 'e'. On what basis do you add 'e', well this is the shortest. This comes right of the key array. Okay, so we have found

the minimum spanning tree. What was the complexity? So, the first initialization requires access to all the vertices and inserting for each into the mean heap. We know that the heapification can happen in order the number of entries being added. So, we can recall the efficient version of heapification. So, this is basically c1 into c2 times 'V'.

The next step is to find the smallest element in the Min-heap and you need to keep doing this for as long as the heap doesn't get empty. So, this will happen order 'V' times and every time you are going to do a c3 times log order v access into the Min-heap. So, this gives you c3 times c6 —V— times log —V—. The inner loop is interesting so the inner if loop is called for each node. For each node, you are basically accessing entire adjacent the list. This is basically summation over v of degree of v. This is nothing but the number of edges. However there is a very interesting implicit call being made to the heap which is not a trivial operation and that is updating the key. So, moment you update the key we're left implicit but you might want to explicitly make a call saying update heap for P. So, this will take order log V time and remember all this happens for time proportional to the some of the links of all the adjacency list. That gives you c4 times c5 times log V multiplied by summation we are all know all vertices of the degree of that vertex. This is nothing but order of e times log v + order of V times log —V—.

Thank you.