Lecture Transcript
Spanning Tree Algorithm (Kruskal's Algorithm)

Hi and welcome to this next lecture on Data Structures and Algorithms. In this session, we'll continue our discussion on finding weighted minimum spanning trees. We'll discuss a greedy algorithm that is based on greedy edge selection, it's called the Kruskal's Algorithm. So recall that we've already discussed a greedy approach using the prim's algorithm. The prim's algorithm is greedy but it is greedy on vertices. The key based on which you pick the next vertex is the weight, the minimum weight of an edge incident on that vertex to a node or vertex which is already a part of the existing minimum spanning tree, so that's how you greedily grow. Here weight edge that does not cause a cycle in the minimum spanning tree. So it's an edges selection rather than vertex selection strategy. However, both of these strategies are based on edge weights. So, the specific strategy is to find an edge of the least possible weight that connects any two sub-trees in the forest. So instead of building a single tree at any point of time you have a bunch of trees, tree 1, tree 2, and so on ,T3.

So set of trees, and as far as possible you try and collect these trees and you want to connect these trees in a way that the edge connecting them has least possible weight. So this edge should have least possible weight and you basically add increasing cost at each step, add 1 edge at a time. So initialisation is based on a forest which consists of singleton sets each for each tree will be a single node. So initialization, your set of tree is T is vertex 1, vertex 2, vertex n. This basically your forest. You have no edge added to begin with and then you introduce edges will pick the edge, the least possible weight that connects two different trees. Of course, in this process you'll merge these two trees, so we'll get a new tree v1, v2. The others will remain what they were. In this way you grow, you keep growing till you have found a single tree or you're not able to find any edge that connects two trees, so grow. How can we implement this? Well, we'll need to make use of some union structures, where you can maintain unions of trees and you also need to keep track of these individual sets.

We are going to maintain these individual sets as sequences. So what we are going to do is have SSets as an array of sequences. So SSets[1] will be sequence sorted in increasing id of vertices in tree 1. In general, thus SSets for i will have sequence of vertices in tree i. So as pointed out we hope that eventually either there is only one of these sets or sequences that is non-empty or we are not able to find the edge connecting two different sequences. There is two different trees represented a sequences of vertices. We've bented in these vertices and sorted order primarily because we want to be able to merge them in linear time. So sorted in order to facilitate merging

of two trees that is sequences, tree represented sequences in linear time. We will come to that step later on here, merge(SSets[u], SSets[v]). So continuing, we're also going to maintain a mapping. This mapping is from every node in the every vertex in the graph to the set of the tree that is corresponds to or it is a part of at any point of time. So SSet id for node i, vertex i is a sequence SSets[j], that i is part of and again the purpose to avoid to be able to find the tree containing i so that you only add edges that connect two different trees, two different sequences. So this will become handy when we look at addition of an edge that connects two different trees in this step.

So let's continue, next we initialize both these data structures, this iterate to all the vertices, again we assume that the vertices v are numbered say 0 to n, n-1. Assume that v is 0, 1 to n-1, just makes our indexing easier. So we're going to insert v into the set v that's the only element of that tree, these are basically the singleton trees and also set, the, set corresponding to be set to be itself. Next, we sort the edges of G.edges into non-decreasing order by weight w. So this is going to help us in our greedy selection strategy. So then we iterate over all the edges. Now we check if this edge indeed connects two different trees. So if the Set id of u is not equal to Set id v that means they are part of two different trees then this T which we haven't describe so far. This T basically is a data structure that keeps track of the trees so far, so this T is set of edges in tree T so far. So it starts with an empty set but if we find that this new edge does connect two different trees, we basically add this edge to the tree that is being grown. We've basically merge the two trees now.

We have now merged the tree corresponding to s set v into s sets u so that we reflect in the update for t and well we also explicitly update our tree data structure itself, our forest data structure in the next step so explicitly update the forest data structure which means you also have to empty v, so your s sets have been merged into u. So s sets u has been set to s sets u union s sets v, however you merge them maintaining the sorted order and then you empty s sets v, this is the next step and then update the set for v to be u. The set index for v is now set to u. Let's see Kruskal's Algorithm in action and I'll also suggest that think about the conversions of this algorithm using the following loop invariant. The loop invariant will be that the tree at any point of time is basically going to contain the edges that belong to the minimum spanning tree and they're the smallest weighted edges, whatever remains to be scanned from the sorted list of edges are those with larger weights and either they connect two different sub tress in the forest that is being grown or they don't connect basically they belong to the same tree they will land up creating a cycle. So you can think about this loop invariant.

So the loop invariant is as follows t contains all edges of the minimum spanning tree with weights less than edges to be scanned or get listed and that the edges that are yet to be listed. These in turn either connect two different trees or form cycle within a tree, in which case the set id's will match. The Kruskal's Algorithm in action, so imagine that we have assigned numbers to these nodes, vertices so a is number 1, b is number 2, 3, 4, 5, 6, 7, 8, 9. The next step for us will be to consider 9 different trees so t will basically consist of 1, 2 and so on until 9. We are also going to sort the edges in increasing order of weights. So the first weight will come across is, beg your pardon, it is between h and g has weight 1. So we are going to add 1 that will lead to the merging of h and g. So now we have fewer trees and we are going to keep track of this, h, g in 1, of course, maintained the sorted order a, b, c, d, e, f, g,h will again maintain the increasing order, i. The next vertex to be chosen should be between, either between g and f or between i and c.

Let's pick this i and c. Now the process continues, we can look at g and f, and then the next edge we consider is between a and b, none of this has actually resulted in an edge within the same tree, so we keep adding them, we have added c,f. What about the next one? It's between c and d. So we have added the a and h. How about h and i? Why didn't we add h and i? So we know that, if, well h and i actually belong to the same tree so (h,i) connects i and h belonging to the same tree. So recall that we had set id, so set id[i] by this point of time will point to h, that is the smaller of the two. In fact, it could be any of the other, so in fact set id[i] will just suffices to say will equal set id[h]. So we will not add this edge h, i but we'll add a, h next, doesn't lead to any cycle, they belong to two different trees, then d, e and so on. So, have we got a tree? Well, we have actually covered all the nodes. Is this minimum spanning tree? Yes, the loop invariant, the initialisation, maintenance and termination if proved for the loop invariant

we are guaranteed that is a minimum spanning tree, and it isn't very difficult to prove those properties for the loop invariant. Analysis of the Kruskal's algorithm. So we have a very specific implementation where we've mentioned the union using list of sequences. So for this the analysis is very straight forward Insertion of the V nodes will take c1 into V times, the sorting will be an order —E—log—E— operation in the sorting on the edges, increasing order of ways, beg your pardon. And then we are going to iterate over these edges in increasing order of weights, for each edge we are going to check if the set ids are different, set ids for the two end points of the edge if they are different we add it to the tree.

This basically is an order E operation over all, we are going to do this for every edge, the worst case for all edges. The merge is an order E operation again because in the worst case the two sets that you have will need to be merged. And finally, we have some other order E operations over all the edges. So one can show that this over all complexity is dominated by this order —E—log—E— sorting step, and the worst case E is order V squared so therefore order —E—log—E— can also be written as —E—log—V—.

Thank You.