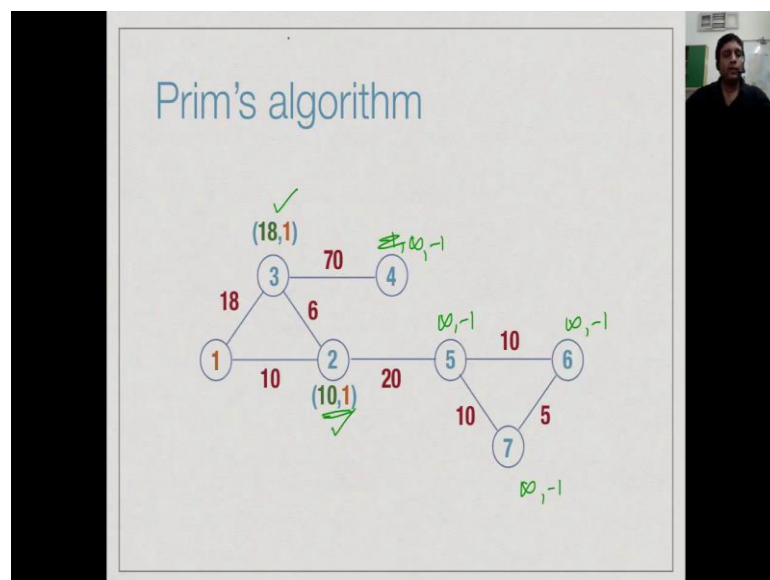


d prime. Suppose d is bigger than v, then now that this is in the tree, now that u has been added in the tree, now v is connected by a smaller edge to the tree, right. So, the distance that I currently have for b is bigger than the weight of u v edge. Then, I will replace that weight by the weight of the u v h and I will say the neighbour of v is now u, so that when I had v to the tree, I will add the edge u.

So, this is exactly what Dijkstra's algorithm does except for this update as this update we had d of u plus the weight of u, right. So, we in Dijkstra's algorithm, we want cumulative distance. Here we want one step distance from the nearest node in the tree, but otherwise prim's algorithm is basically a restatement or Dijkstra's algorithm with a different update function and additionally we have this thing that we could have done it in Dijkstra's also. We could have maintained the Dijkstra's algorithm the path, right. So, had we maintained the path, it will be exactly like this neighbour relation here. We want to know when these edges added to my shortest paths set, the burn set, why it was added, right. So, here we are doing that we are adding it and we are also remembering the edge to that.

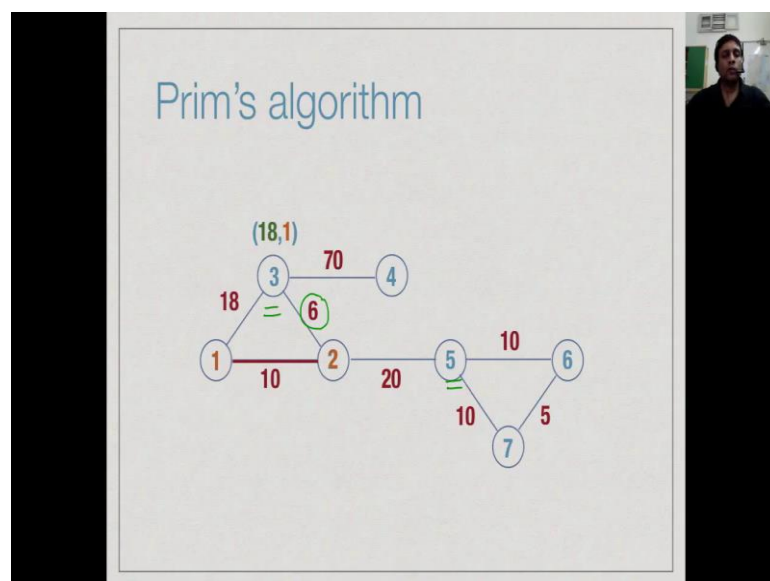
(Refer Slide Time: 13:27)



So, let us try and execute before during the complexity analysis of these things. So, remember we can start anywhere. So, let us start at 1, right. We start at 1 and we mark our tree consisting of form. Now, since this is an edge start at 1, we have to update the

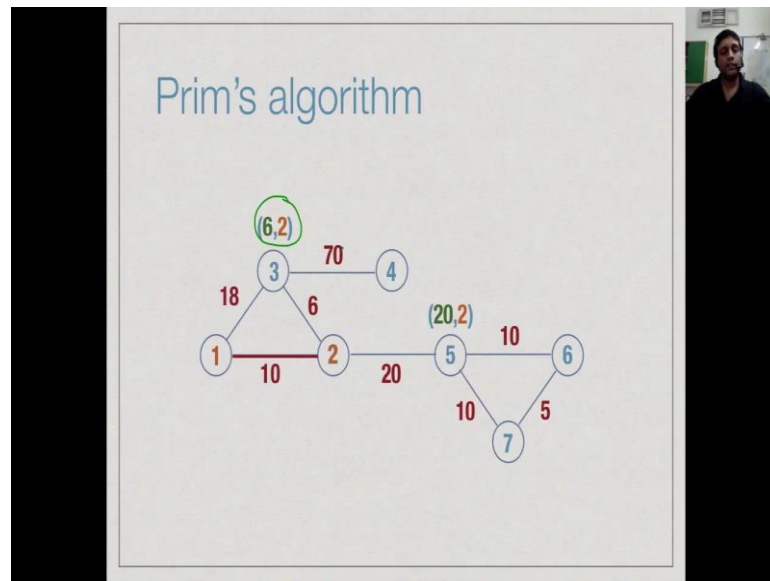
values in the neighbours of 1, namely at 2. So, we mark for 3. We say that is the distance which we mark in green, so the tree is 18 because the tree consist vertex 1 and it is neighbour in the tree which is at the distance is the vertex 1. Similarly, the other neighbour of 1 is 2. So, we will say is its distance is 10, and its neighbour is 1, right. So, everywhere else I have not mentioned it explicitly, but everywhere else the values are minus 1 and sorry, infinity and minus 1. So, this is infinity and neighbour is minus 1. So, this is the default value. So, wherever the default value is present, we will just leave it out indicating that the value is effectively not been cyclic. We know it is a connected graph. So, we will eventually set it. So, you do not have worry about it, but in this discussion we just leave it out. So, we have these two candidates now which are not visited and which have some reasonable distance associated. So, we will pick smaller than 2. So, we pick this one which is 10, and therefore at a next step we visit the vertex 2 and we add this edge 1 to 2.

(Refer Slide Time: 14:46)



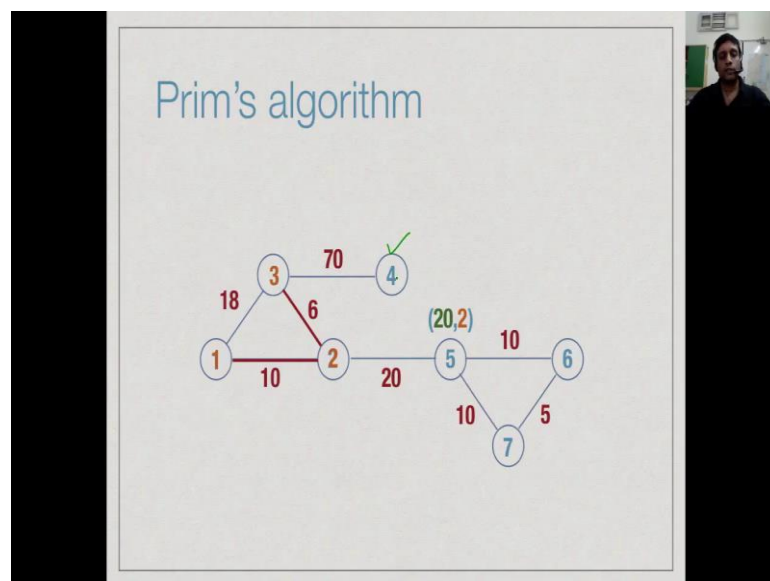
Now, having added 2, we have this update. So, we look at the neighbours. So, the neighbours of 2 are the vertex 3 and vertex 5. So, for the vertex 2, we have a new distance 6. So, if you go via to the distance of 3 to the tree 6 and there it could be connected to 2 which is 6 is smaller than 18, right. So, 18 was earlier best estimate of how far 3 were in the tree.

(Refer Slide Time: 15:16)



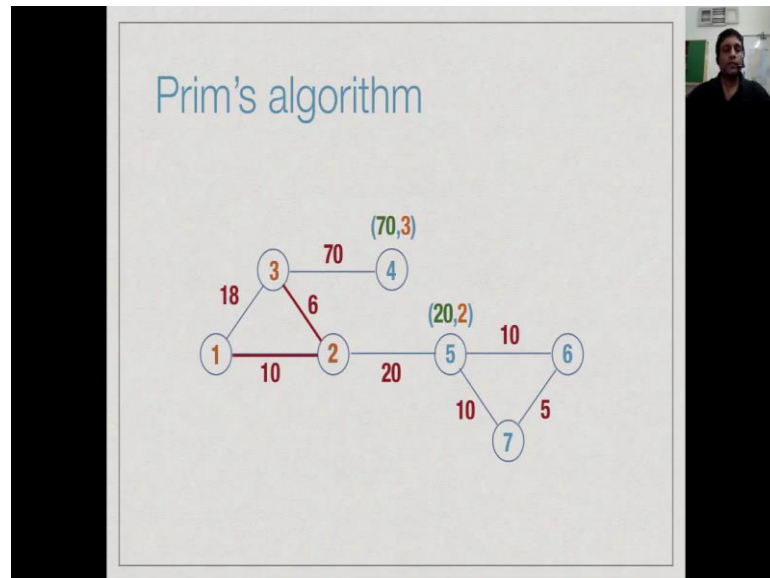
So, you will replace 18, 1 by 6, 2 indicating now the vertex 3 is 6 distance away from the tree, and if it were to be connected at the distance, it could be connected to 2. Similarly, 5 which was earlier unlabelled, now becomes labelled as 22 indicating that its distance is 20 from the tree and its neighbour in the tree is the label vertex v. Now, we again pick the smaller of the two. So, we will pick this vertex 3 to add to the tree, right.

(Refer Slide Time: 15:46)



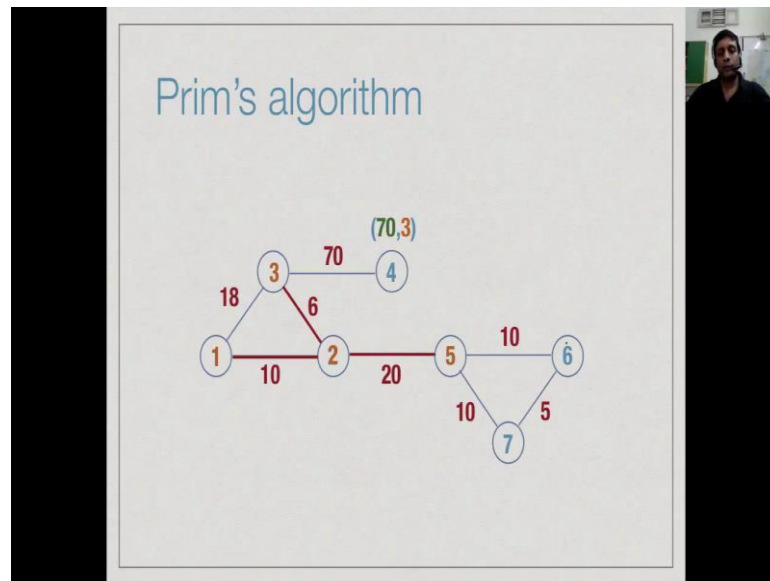
Once we add it, you will obtain the status of 4 because that is the only new label we do not update the status of 1 because 1 is already been added to the tree. We only look at those neighbours of tree which are not visited. So, now 4 gets the distance 70 with the neighbour 3, and then among these two, now 20 is smaller than 70.

(Refer Slide Time: 15:55)

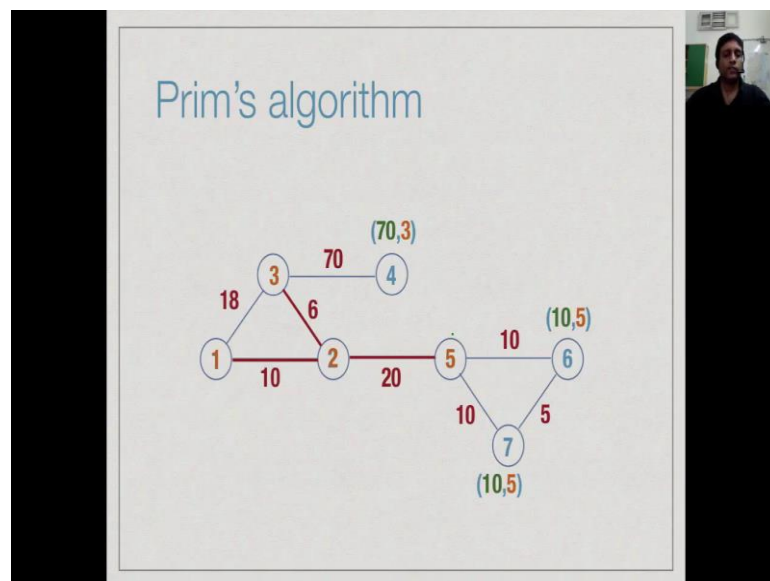


So, we will add 5 to our tree, and then we will update the status of 6 and 7.

(Refer Slide Time: 16:05)



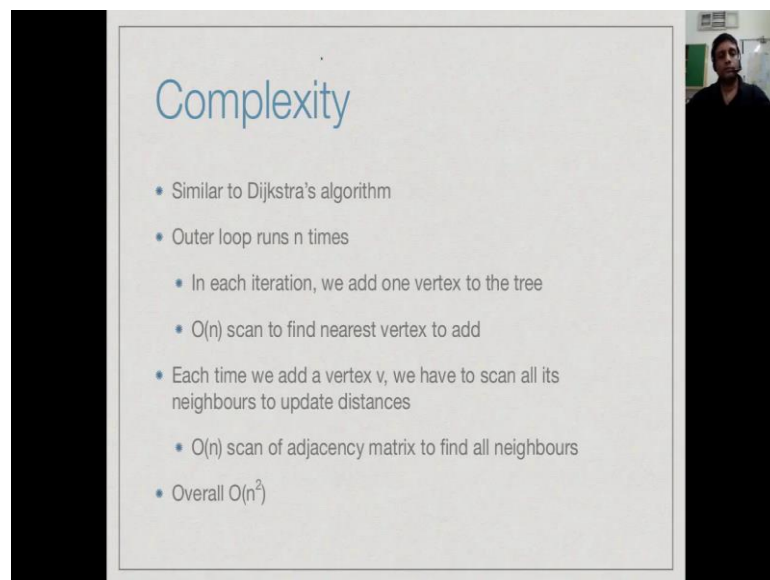
(Refer Slide Time: 16:10)



So, 6 is now distance 10 with neighbour 5, 7 is also is distance 10 with neighbour 5. Now, we have two vertices with distance 10. We could pick either one. So, let us for example pick 7. If we pick 7, then we add it to the tree and now, we update the status of the 6. Earlier it was a distance 10 with neighbour 5, but now it is a distance 5 with neighbour 6. So, we reduce it with neighbour 7. We reduce its distance and we change its

neighbour. Now, among 5 and 17, we have 6 as the vertex, 6 as the newer one, and then we had finally 4 and this is the tree that we get. This is how prim's algorithm works. It is very similar to Dijkstra's principle but it uses a very different update.

(Refer Slide Time: 16:57)

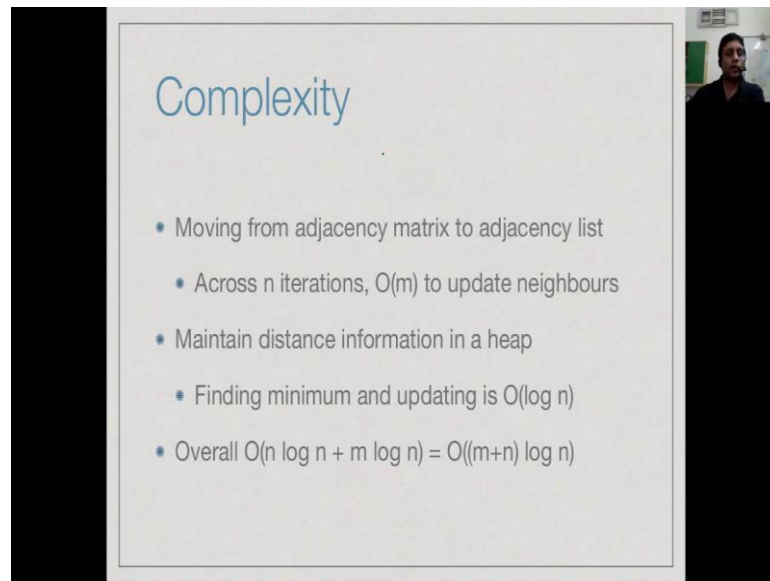


Complexity

- Similar to Dijkstra's algorithm
- Outer loop runs n times
 - In each iteration, we add one vertex to the tree
 - $O(n)$ scan to find nearest vertex to add
- Each time we add a vertex v , we have to scan all its neighbours to update distances
 - $O(n)$ scan of adjacency matrix to find all neighbours
- Overall $O(n^2)$

So, the complexity also is similar to Dijkstra's algorithm. We have an outer loop which runs n times order n times because we have to add n minus 1 edge to form that tree and each time we add vertex with the tree. Now, there is this order n scan in order to find the minimum cost vertex to add. So, we already saw this is what dijkstra's algorithm to find the minimum distance vertex to add, and then when we add a vertex, we have to do and again a scan to update all the entries. So, we have an adjacency matrix. This will again take order n time and therefore, overall it takes order n square.

(Refer Slide Time: 17:36)

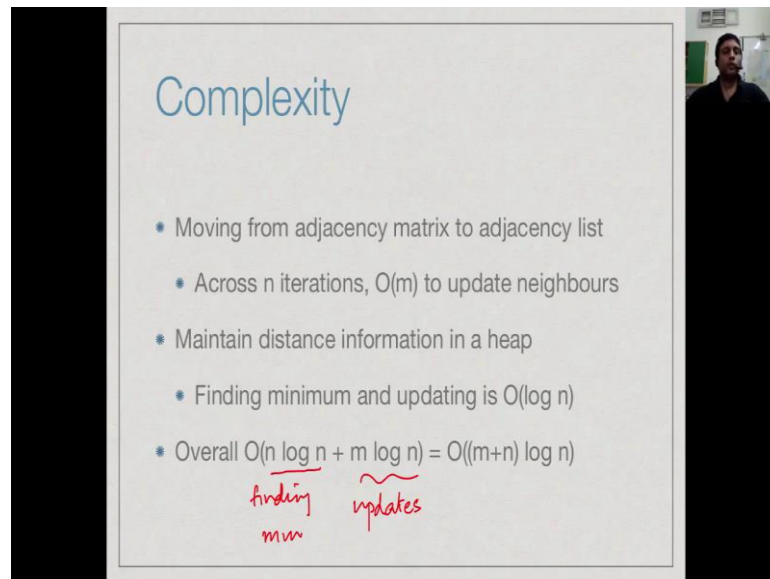


Complexity

- Moving from adjacency matrix to adjacency list
- Across n iterations, $O(m)$ to update neighbours
- Maintain distance information in a heap
 - * Finding minimum and updating is $O(\log n)$
- * Overall $O(n \log n + m \log n) = O((m+n) \log n)$

So, exactly as Dijkstra's algorithm moving an adjacency matrix to adjacency list representation of the edges allows us to reduce the complexity of the updates. So, across the n iterations, we do a total of order m updates because update only according to the neighbours, the degree, the sum of the degrees of all the vertices. However, in order to bring that the order n square, we also need to be able to compute the minimum distance efficiently for which we need a heap, right. So, once we have a heap which we will examine in a later lecture, the claim is we can find the minimum and update the distance in $n \log$ time. So, this gives us overall complexity exactly let x of $m \log n$ plus $m \log n$.

(Refer Slide Time: 18:19)

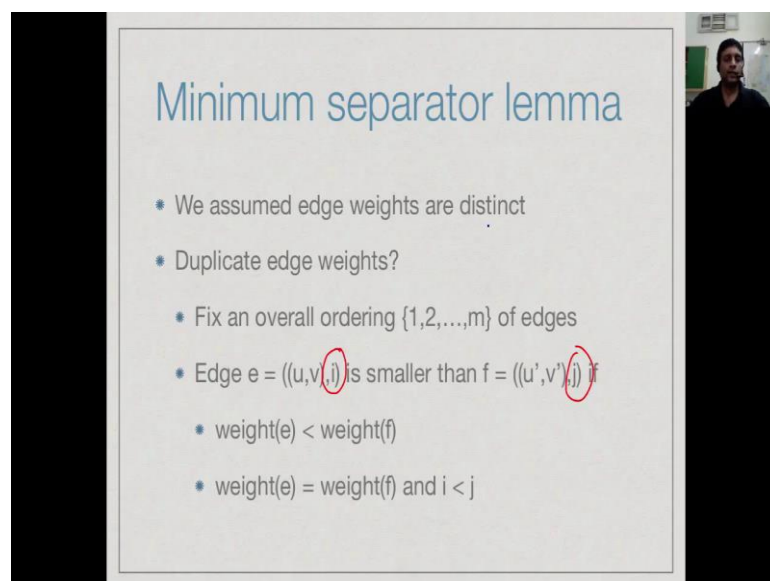


Complexity

- Moving from adjacency matrix to adjacency list
- Across n iterations, $O(m)$ to update neighbours
- Maintain distance information in a heap
 - Finding minimum and updating is $O(\log n)$
- Overall $O(n \log n + m \log n) = O((m+n) \log n)$
 - finding min* (under $n \log n$)
 - updates* (under $m \log n$)

So, this comes from finding the minimum because n time we have to find the minimum and this comes from the updates, because we have to do m updates overall. Each updates takes $\log n$ times, so we get n plus $n \log n$ exactly as we did for Dijkstra's algorithm.

(Refer Slide Time: 18:36)



Minimum separator lemma

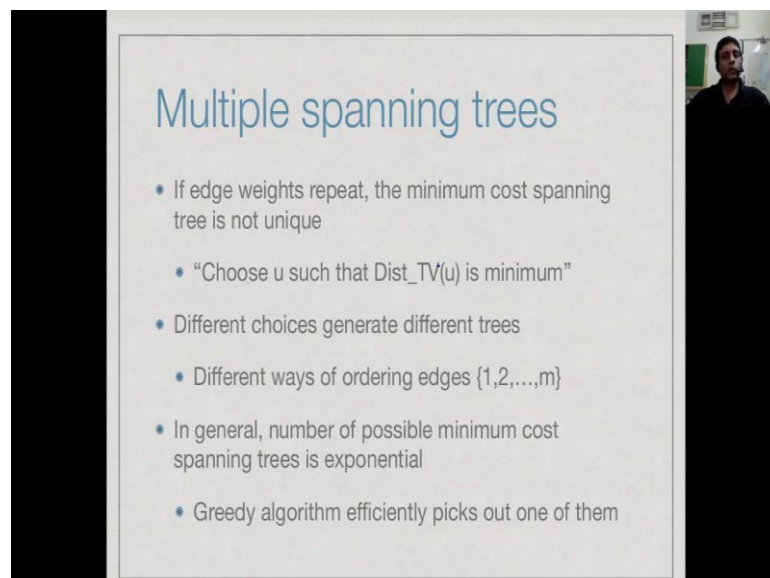
- We assumed edge weights are distinct
- Duplicate edge weights?
 - Fix an overall ordering $\{1, 2, \dots, m\}$ of edges
- Edge $e = ((u, v), i)$ is smaller than $f = ((u', v'), j)$ if
 - $\text{weight}(e) < \text{weight}(f)$
 - $\text{weight}(e) = \text{weight}(f)$ and $i < j$

So, one last point before we leave prim's algorithm. Remember that in the correctness we

have to use that minimum separator lemma in which we had assumed that edge weights are distinct. So, of course we have seen in the example that we executed, we could have edges, multiple edges with the same weight. So, how do we deal with this in the lemma? Well, we could argue with that you can make that cost to be not just exactly the weight, but the weight plus some other term. So, in general we could say that we fix some overall ordering of the edges. There are m edges. So, we just number the edges arbitrarily 1 to n and we say that one edge is smaller than the edges smaller way if either the weight is actually is smaller or the weights are equal, but the index in the ordering is small, right. So, e and f we have the weight of uv and the weight of u' and v' , but we also have the index i and j .

So, this is some i between 1 to m and this is some j between 1 to m . So, either weight of e must be smaller than weight of f or the weights are equal, then i must be smaller than j , right. So, this gives us a tie breaking rule. So, this will now basically tell us that we can always compare two edges and declare 1 is smaller than the other and what prim's algorithm will do is pick the smaller, right.

(Refer Slide Time: 19:49)



Multiple spanning trees

- If edge weights repeat, the minimum cost spanning tree is not unique
 - "Choose u such that $\text{Dist_TV}(u)$ is minimum"
- Different choices generate different trees
 - Different ways of ordering edges $\{1, 2, \dots, m\}$
- In general, number of possible minimum cost spanning trees is exponential
- Greedy algorithm efficiently picks out one of them

So, we want this corresponds to saying is that we are actually giving a strategy for choosing when we have two equal things. So, the algorithm says choose that distance

which is minimum and if multiple use with the same minimum distance, we pick an arbitrary point. So, what is meant is to pick an arbitrary point whether it means some sense to choose an order among them and go in that order. Therefore, if you choose different orderings, then we get different trees. So, therefore we have multiple edges in a tree which have the same weight. In general, we may not get a unique spanning tree. In fact, you can check if you have all weights the same for example. Basically you have to keep adding or dropping different edges and you will have an exponential number of trees because an edge could be there in one tree and may not be there in another tree and so on, right.

So, overall the number of possible minimum cost spanning tree could be very large. What prim's algorithm does and what Kruskal's also will do when we look at in the next lecture is to use a greedy strategy to efficiently pick out one of these possible things. Now, if the edge weights are unique, there is not much choice. You have to pick up same tree. If the edge weights are duplicated, you can definitely have multiple trees and this strategy of picking out the smallest one at each stage will give us a quick way to identify one of the smallest ones, but not a unique one.