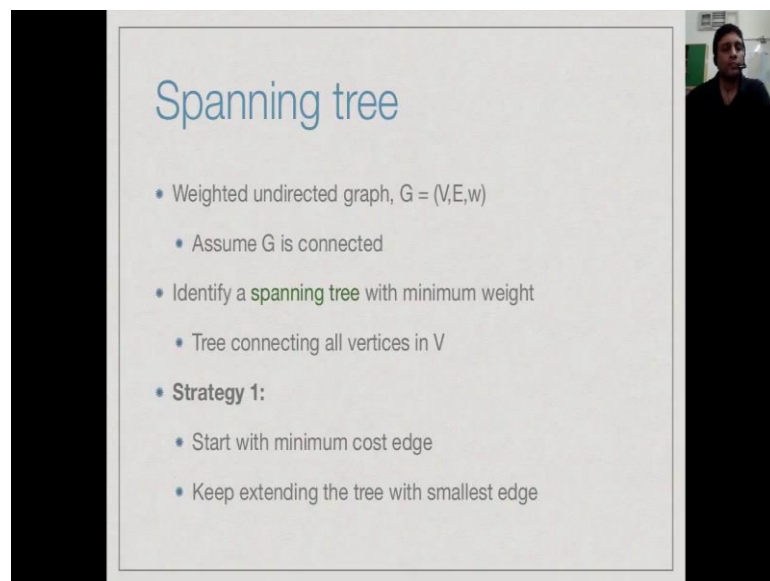


Design and Analysis of Algorithms
Prof. Madhavan Mukund
Chennai Mathematical Institute

Week - 04
Module - 06
Lecture - 30
Spanning trees: Prim's algorithm

So, we are looking at the problem of constructing a minimum cost spanning tree in a weighted graph. We said there we have two basic strategies one could think of to do this. The first one leads to an algorithm called Prim's algorithm, and the second one leads to an algorithm called Kruskal's algorithm. So, in this lecture we would look at Prim's algorithm.

(Refer Slide Time: 00:20)



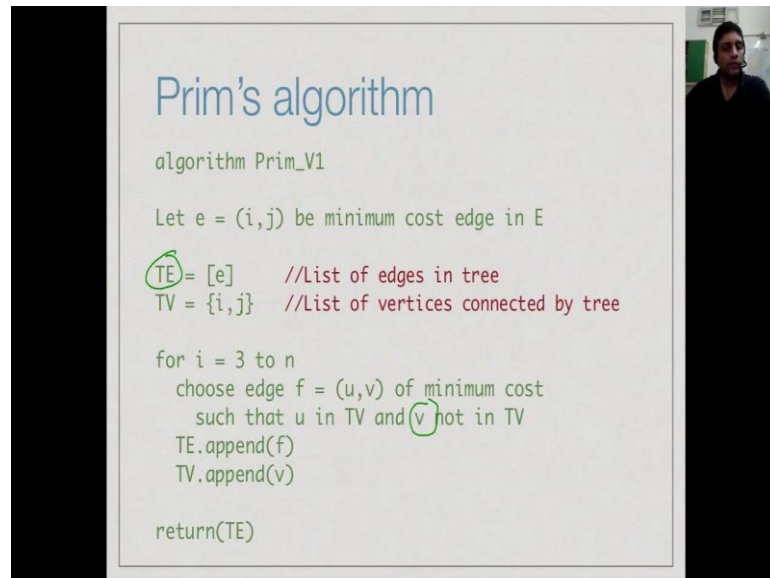
Spanning tree

- Weighted undirected graph, $G = (V, E, w)$
 - Assume G is connected
- Identify a **spanning tree** with minimum weight
 - Tree connecting all vertices in V
- **Strategy 1:**
 - Start with minimum cost edge
 - Keep extending the tree with smallest edge

So, the problem domain is the following. We have a weighted undirected graph. So, V is the set of vertices, E is the set of edges and w is a weight function. We assumed that G is connected, because G is not connected and there is no way to build a spanning tree. Spanning tree, remember is a sub set of the edges which connects all the vertices in g . So, g is not already connected, then there is no way we can actually connect using a subset of edges. So, G is a connected weighted undirected graph, and now we want to identify spanning tree with minimum weight. So, this strategy in Prim's algorithm starts with the minimum cost edge, and keep extending the tree with the smallest edge

connected to the current tree.

(Refer Slide Time: 01:04)



```
Prim's algorithm

algorithm Prim_V1

  Let e = (i,j) be minimum cost edge in E
  TE = [e]    //List of edges in tree
  TV = {i,j}  //List of vertices connected by tree

  for i = 3 to n
    choose edge f = (u,v) of minimum cost
    such that u in TV and v not in TV
    TE.append(f)
    TV.append(v)

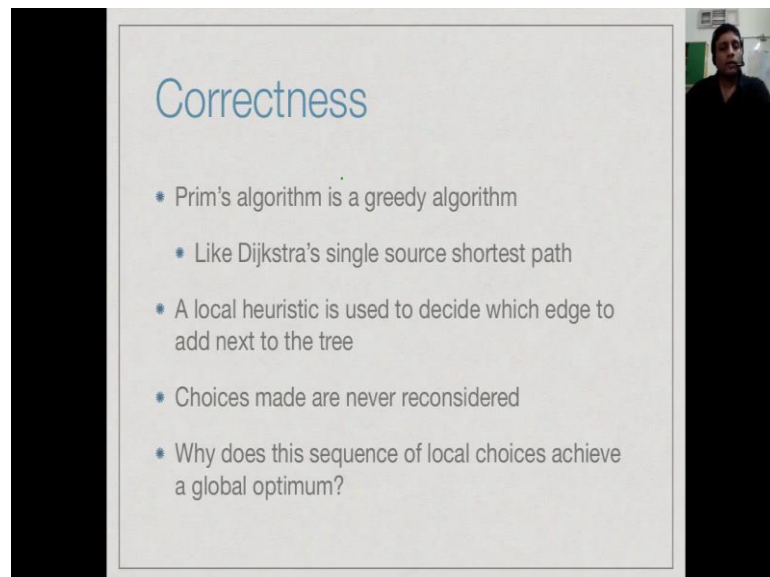
  return(TE)
```

So, here is a kind of high level version of prim's algorithm, right. So, we start with minimum cost edge and we add it to the list. So, T E is a list of edges that form a tree. So, we will describe the tree as a collection of edges, and then we note that we have added i and j to the tree. So, i and j are now connected. So, this leaves n minus 2 vertices which have to be connected. So, we have to do something n minus 2 times. So, n minus 2 times we have to add an edge. Each time we add an edge; one more vertex would be connected. So, we know that after that many edges, you will have a tree. Remember that the tree has totally n minus 1 edge. So, we have added the first edge by starting with a minimum cost edge. So, we can add n minus 2.

So, what we do is, n minus 2 times, we choose the smallest edge which has one end point in the tree and one end point outside the tree. So, this is a vertex v now which is not connected to the tree. So, we connect it. So, we append this new edge to our list of tree edges and we add this vertex to our list of tree vertices, and at the end after doing this n minus 2 times, it claim as we connected all the edges, we have a spanning tree and more over the claim is because we are choosing the minimum cost edge to add a edge point. The overall thing is a minimum cost spanning tree. Of course, we will not prove all this,

but this is what we aim as prim's algorithm, right.

(Refer Slide Time: 02:33)



Correctness

- Prim's algorithm is a greedy algorithm
 - Like Dijkstra's single source shortest path
- A local heuristic is used to decide which edge to add next to the tree
- Choices made are never reconsidered
- Why does this sequence of local choices achieve a global optimum?

So, why do we need to prove something? Well, you can see that like Dijkstra's algorithm, prim's algorithm is a very greedy algorithm, right. At each point, we have to decide how to extend the tree. So, we look in the neighbourhood that the current tree, we look for the nearest vertex which is connected to the tree, but the shortest edge and we add it. So, this is a local choice, and then we keep making these sequences of local choices and ultimately, we arrive globally at a spanning tree and our claim is globally we have built the best possible tree, right. So, this is always an example of a greedy algorithm where you make a sequence of local choices. Never go back and reconsider them and finally, achieve a global optimum and very often as we mentioned before with the Dijkstra's algorithm, such a strategy may not give you the right thing. So, you have to always justify that this works.

(Refer Slide Time: 03:23)

Minimum separator lemma

- Let V be partitioned into two non-empty sets U and $W = V - U$
- Let $e = (u, w)$ be minimum cost edge with u in U and w in W
- Assume all edges have different weights (relax this condition later)
- Then every minimum cost spanning tree must include e

So, in order to prove prim's algorithm correct and indeed, we will also use this to prove Kruskal's algorithm, correct later on. We prove a very useful remark all the minimum separator lemma. So, let us assume that we have this weighted undirected graph and we look at the set of vertices V , right and we assume that it is divided into two parts, right. So, this is called partitioning. So, there are two separate disjoint parts which I will call U and W , and I am assuming that both of these are non-empty. So, there is at least one vertex u and one vertex w . Now, let me look at the smallest edge which goes across this partition. Remember that the whole graph is connected. So, there must be a way to go from u to w . Some of all the ways I can go from u to w . Let me check the smallest edge. Let me call the end point small u and small w .

So, now, the claim is that every minimum cost spanning tree must include this edge. This is a very powerful claim. Of course, there is a side condition which is that we are assuming for a moment at no two edges have the same weight. We will see later on how we will relax this condition, right. So, under the condition that no two the edges at the same weight, the minimum separator lemma is that whenever you separate V into two parts which are not empty, then the smallest edge connecting these two parts must lie in every spanning tree, every minimum cost spanning tree.

(Refer Slide Time: 04:43)

Minimum separator lemma

- Let T be a minimum cost spanning tree, $e = (u, w)$ not in T
- u in U and w in W are connected by a path p in T
 - p starts in U and ends in W
- Let $f = (u', w')$ be the first edge on p such that u' in U and w' in W
- Drop f and add e to get a smaller spanning tree

So, now why is the case? So, let us assume that we have these two parts. So, I will draw one part say this yellow thing and let us call this u and which have not drawn a boundary for, this is v , and this is w .

(Refer Slide Time: 05:00)

Minimum separator lemma

- Let T be a minimum cost spanning tree, $e = (u, w)$ not in T
- u in U and w in W are connected by a path p in T
 - p starts in U and ends in W
- Let $f = (u', w')$ be the first edge on p such that u' in U and w' in W
- Drop f and add e to get a smaller spanning tree

So, now let us look at the smallest edge connecting u and w , right. So, now the claim is

that this must be in every minimum cost spanning tree. So, suppose it is not. So, then suppose there must be some minimum cost spanning tree, because we know that the graph is connected. So, there are many spanning trees unless assume that the minimum cost spanning tree T which does not include this edge.

(Refer Slide Time: 05:28)

Minimum separator lemma

- Let T be a minimum cost spanning tree, $e = (u, w)$ not in T
- u in U and w in W are connected by a path p in T
- p starts in U and ends in W
- Let $f = (u', w')$ be the first edge on p such that u' in U and w' in W
- Drop f and add e to get a smaller spanning tree

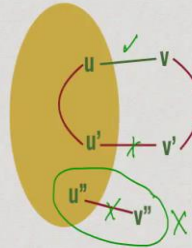
The diagram shows a yellow oval representing set U . Inside U , there are vertices u and u' . Outside U , there are vertices v and v' . A green edge (u, v) is shown with a checkmark, representing edge e . A red path p is shown starting from u and ending at w (not explicitly labeled but implied). The edge (u', v') is shown with a red 'X', representing edge f to be removed. The new tree T' is indicated by the equation $T' = T - (u', v') + (u, v)$.

So, in that tree u must be connected to the tree, because any spanning tree connects all the vertices. So, there is a red path from u to w in my hypothetical spanning tree T which does not include this edge. So, now the claim is if I take that particular tree, and then I remove the edge u' v' and replace it by the edge u, v , we get a new tree, right. I get tree T' . So, T' is T minus edge u' v' plus the edge u, v , but now by assumption u, v was the smallest vertex, smallest weight edge going from inside U to outside U , right. So, therefore, u, v has weight strictly less than u', v' . Therefore, T' has a weight strictly less than T and you can check that everything else is connected because anything which have connected, so u' is now connected v' is long term and therefore, all other vertices which connect by T remain connected by T' . Therefore, T' is a valid spanning tree. It is of smaller cost and therefore, T could not have been a minimum cost spanning tree. So, this is a proof that the smallest cost edge from inside the partition to outside the partition must lie in every minimum cost spanning tree.

(Refer Slide Time: 06:45)

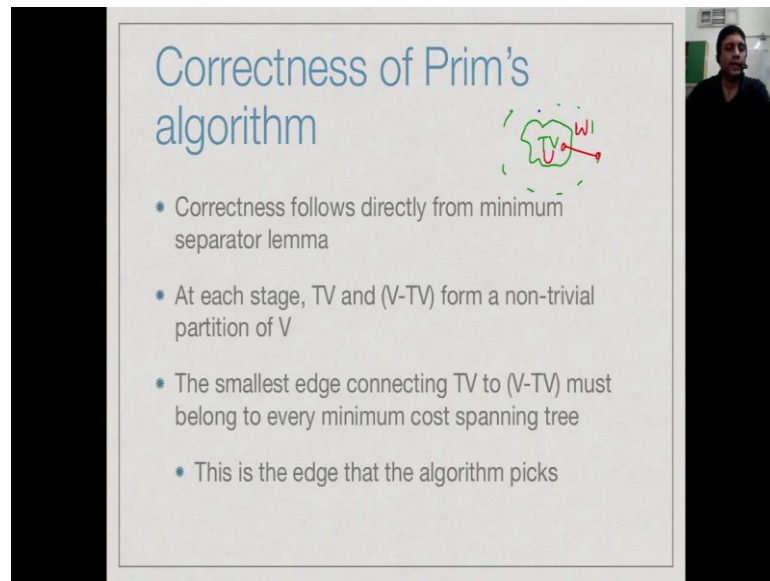
Minimum separator lemma

- Proof of the lemma is slightly subtle
- Not enough to replace **any** edge from U to W by $e = (u,v)$
- Need to identify such an edge on the path from u to v



Before we move ahead, I just want to make one small remark. So, we would have to be careful when we prove this lemma a little bit. So, it is true that among all the edges going from inside to outside, $u v$ is the smallest one. So, we might be tempted to just say, so uv is the smallest one, pick any edge in my given tree T prime given tree T which goes from inside to outside, then replace. So, for instance, we might accidentally pick up this edge and replace it with this edge. Notice by picking up this edge and replace it with this edge, then perhaps there is no other way to get a double prime, right. So, it is very crucial that we choose that correct edge to replace. So, we have a target and we want to introduce u to v . Therefore, we must follow the path in T from u to v and that path must start inside and go outside. So, it must cross the boundary somewhere, and this is the edge to replace. So, we should not make the mistake of replacing some arbitrary edge. We must replace that edge which allows us to go from u to v in the hypothetical tree, not at to make the new tree.

(Refer Slide Time: 07:50)

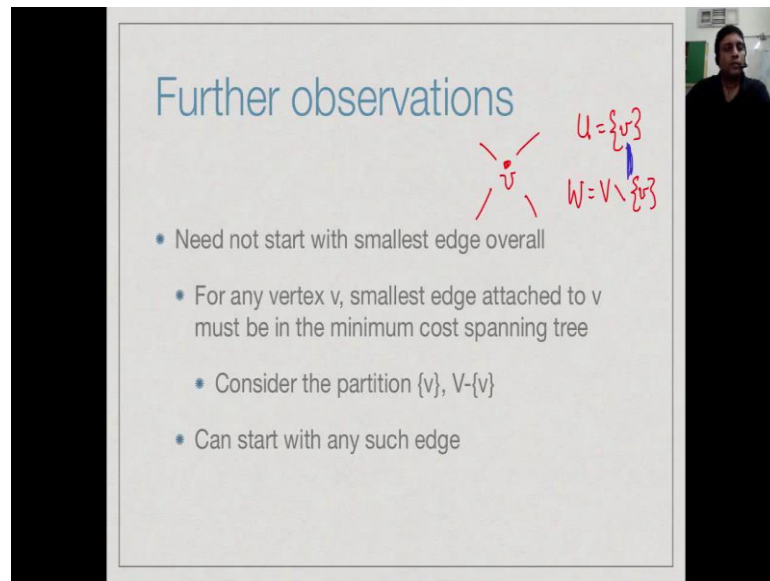


Correctness of Prim's algorithm

- Correctness follows directly from minimum separator lemma
- At each stage, TV and $(V-TV)$ form a non-trivial partition of V
- The smallest edge connecting TV to $(V-TV)$ must belong to every minimum cost spanning tree
- This is the edge that the algorithm picks

So, once we had this lemma, the correctness of prim's algorithm is very obvious. So, at every stage remember in prim's algorithm, we have built this tree TV which consists of a few edges, and then we have everything just lying outside and now among these we want to connect one of that, right. So, if you think of this said the set inside is my u and the set outside is my w , and we are picking by assumption in prim's algorithm, the smallest weight edge which connects u to w . By this minimum separator lemma, this edge must line every spanning tree. So, the algorithm, the prim's algorithm, the edge that the prim's algorithm picks is in fact the edge that the lemma forces us to pick. So, therefore, prim's algorithm is definitely correct.

(Refer Slide Time: 08:35)



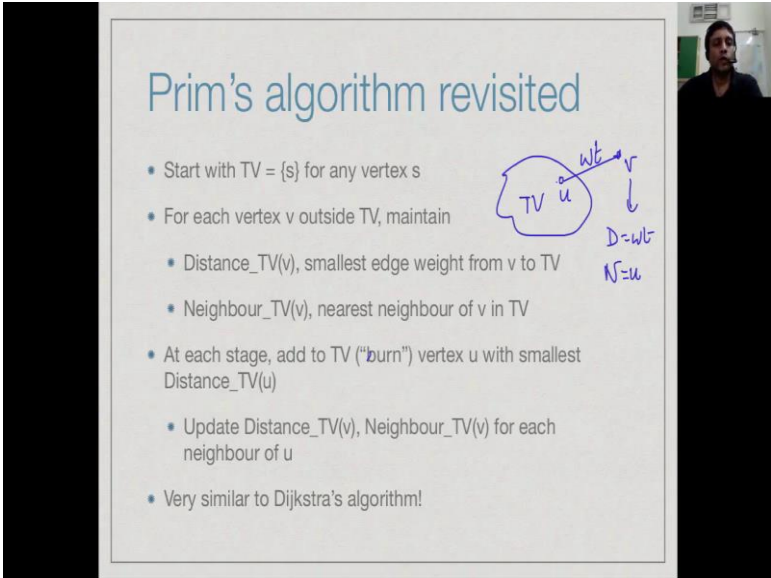
Further observations

- Need not start with smallest edge overall
- For any vertex v , smallest edge attached to v must be in the minimum cost spanning tree
- Consider the partition $\{v\}, V-\{v\}$
- Can start with any such edge

$U = \{v\}$
 $W = V \setminus \{v\}$

So, in fact we can use the lemma to make prim's algorithm little more relaxed. Recall that narrow regional formulation we started with the smallest edge, but now it is easy to see that if I take any vertex, right and I look at all the edges going out of it, then I can take u to be the vertex itself and I can take w to be everything else. We set minus this vertex. Then, I know that the smallest edge which goes from v to this edge must be in every spanning tree. In other words, if I start at any vertex and look at the smallest edge attached to it, I can start with that because that must be in every spanning tree by the minimum separator lemma.

(Refer Slide Time: 09:18)



Prim's algorithm revisited

- Start with $TV = \{s\}$ for any vertex s
- For each vertex v outside TV , maintain
 - $Distance_TV(v)$, smallest edge weight from v to TV
 - $Neighbour_TV(v)$, nearest neighbour of v in TV
- At each stage, add to TV ("burn") vertex u with smallest $Distance_TV(u)$
- Update $Distance_TV(v)$, $Neighbour_TV(v)$ for each neighbour of u
- Very similar to Dijkstra's algorithm!

Handwritten diagram: A cloud-like shape labeled TV contains a vertex u . An edge connects u to a vertex v outside the cloud, labeled w_{uv} . To the right of the diagram, the following is written:
 $D = w_{uv}$
 $N = u$

So, this gives us the following algorithm for prim's strategy. So, we start with any vertex. Now, for each vertex which is not in our current set of tree vertices, we maintain the smallest edge weight from that vertex to some tree vertex is called that distance of v and also because we want to build up the tree the set of edges, we remember where that edge goes to. So, we remember it does have a neighbour, right. So, if I have some tree at a given point and I know that for this v , this neighbour u is the smallest edge connecting into v , then here I will keep its as distance as the weight of the edge, I will keep it as neighbour as u . This will allow me to keep track of which edges are added. So, now, at every stage I look for the smallest vertex which is outside in terms of the distance. Then, I add it to the set and I update it as neighbours distances and values, right. So, this is very similar to Dijkstra's algorithm, right. The only thing is the update of the distance does not involve adding my distance plus the weight. It only involves considering the weight. So, when see the algorithm itself, we will see the parallel Dijkstra's algorithm even clearer.

(Refer Slide Time: 10:36)

Prim's algorithm, refined

```
function Prim
  for i = 1 to n
    visited[i] = False; Nbr_TV[i] = -1; Dist_TV[i] = infinity

  TE = [] //List of spanning tree edges
  visited[1] = True
  for each edge (1,j)
    Nbr_TV[j] = 1; Dist_TV[j] = weight(1,j)

  for i = 2 to n
    Choose u such that Visited[u] == False
    and Dist_TV[u] is minimum

    Visited[u] = True
    TE.append{(u,Nbr_TV[u])}
    for each edge (u,v) with Visited[v] == False
      if Dist_TV[v] > weight(u,v)
        Dist_TV[v] = weight(u,v); Nbr_TV[i] = u
```

Handwritten notes on the slide include:

- $d(u) + w(u,v)$ (written vertically next to the last line of code)
- $d > d'$ (written next to the first diagram)
- $N=1$ (written next to the first diagram)
- d and d' (written next to the second diagram)

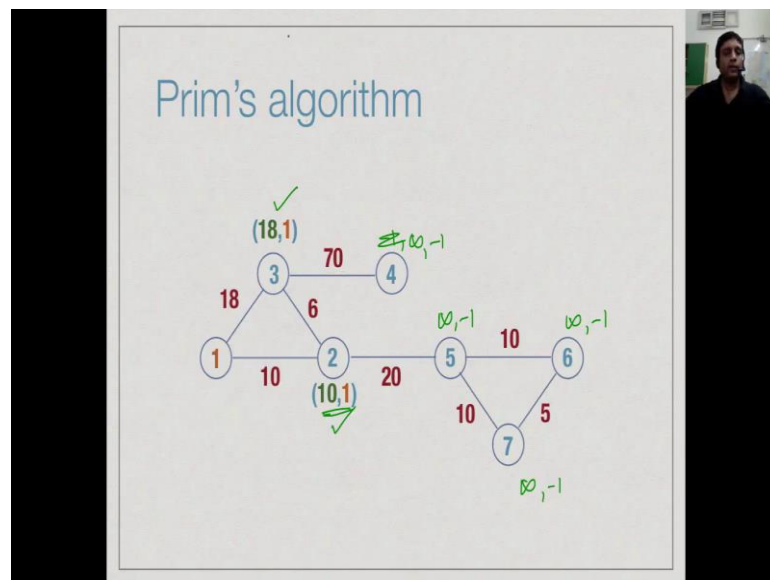
So, here is the final algorithm for prim's shortest or minimum cost spanning tree right. So, you initialize all vertices to be unvisited. This is the bond thing. We miss them all to have no neighbours, because nothing is in the tree. So, they have no neighbours in the tree and they are all distance infinity. This initializes the tree. Now, I pick some initials starting vertex a 1 and mark it to be visited, but I do not have any edges, right. So, now for each edge going out of 1, I update its status. So, I say for every edge one of the form 1, j, I said that the neighbour of j in the tree, so the tree now consists of just this one vertex and trivial tree which has one vertex and therefore, 1 -1 0 8, right. So, the neighbour of j is 1 because that is its connection and the distance is the weight of this edge, right. So, this is my first step.

Now, I have to add the remaining n minus 1 edges to my tree. So, n minus 1 times I do what you do in Dijkstra's algorithm in different format. You pick that u which is not visited and whose distance is minimum. Mark it as visited. Now, you know how it is connected to the tree. So, you add the edge which tells us how it is connected u and neighbour of u. This edge we add to the set of tree edges. Now, for every edge out of u whose neighbour is not visited, if the current distance to the tree is more than the weight of this edge, so basically I had now added this u and there is another vertex v and it claims to be connected somewhere else, right. So, maybe this distance d and this distance

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.

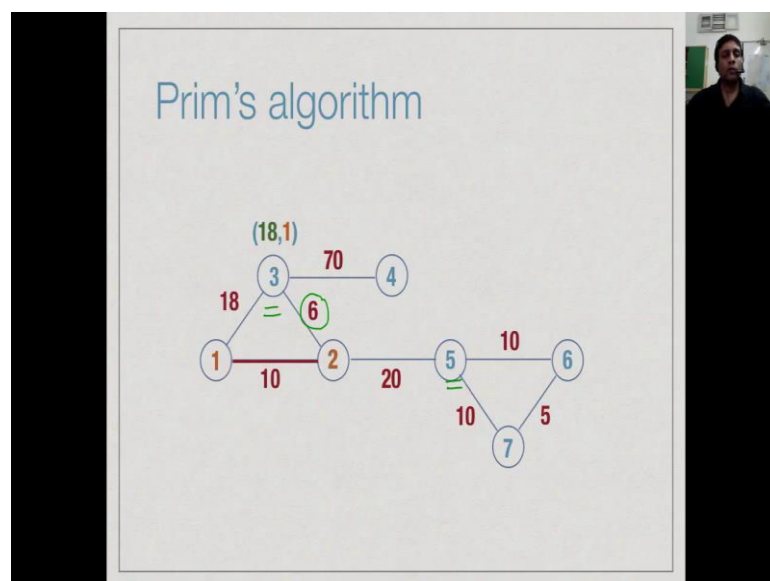
(Refer Slide Time: 13:27)



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

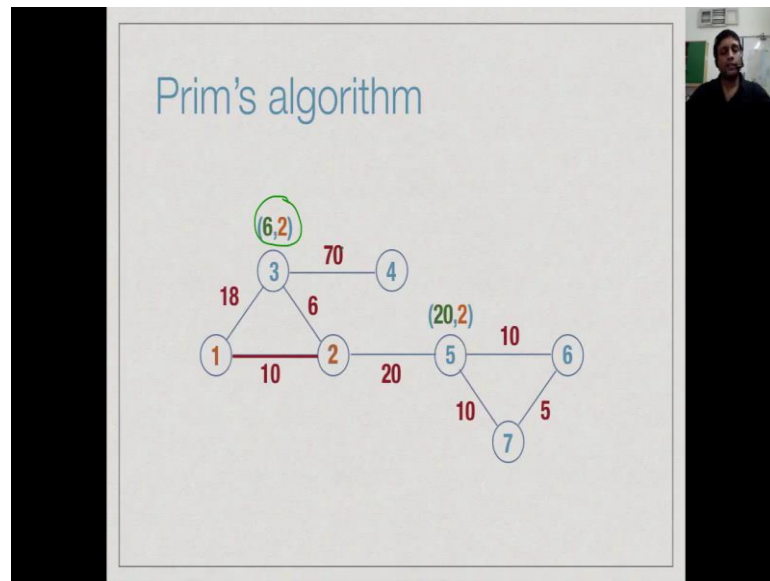
values in the neighbours of 1, namely at 2. So, we mark for 3. We say that is the distance which we mark in green, so the tree is 18 because the tree consist vertex 1 and it is neighbour in the tree which is at the distance is the vertex 1. Similarly, the other neighbour of 1 is 2. So, we will say is 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)



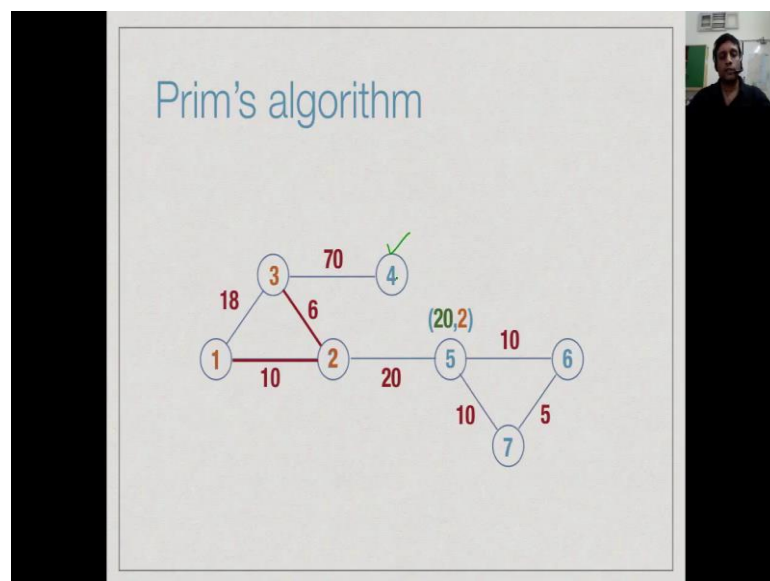
Now, having added 2, we have this update. So, we look at the neighbours. So, the neighbours of 2 are the vertex 3 and vertex 5. So, for the vertex 2, we have a new distance 6. So, if you go via to the distance of 3 to the tree 6 and there it could be connected to 2 which is 6 is smaller than 18, right.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

(Refer Slide Time: 15:16)



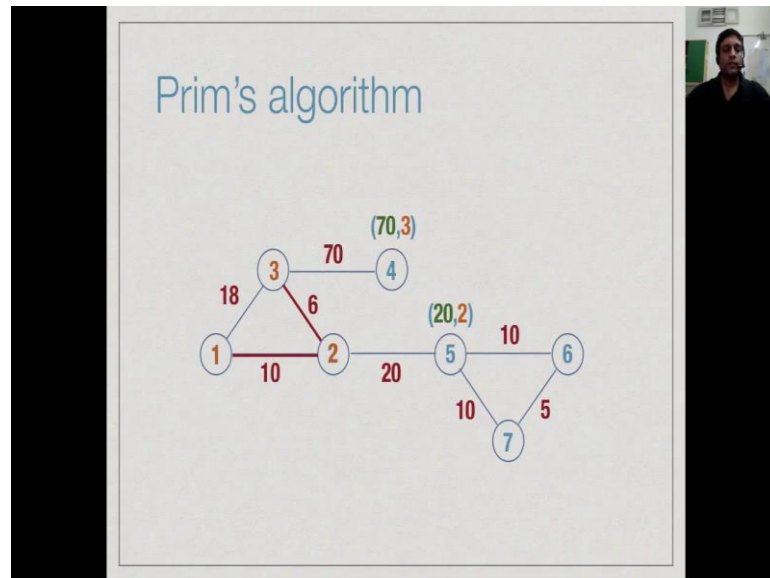
So, you will replace 18, 1 by 6, 2 indicating now the vertex 3 is 6 distance away from the tree, and if it were to be connected at the distance, it could be connected to 2. Similarly, 5 which was earlier unlabelled, now becomes labelled as 22 indicating that its distance is 20 from the tree and its neighbour 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)



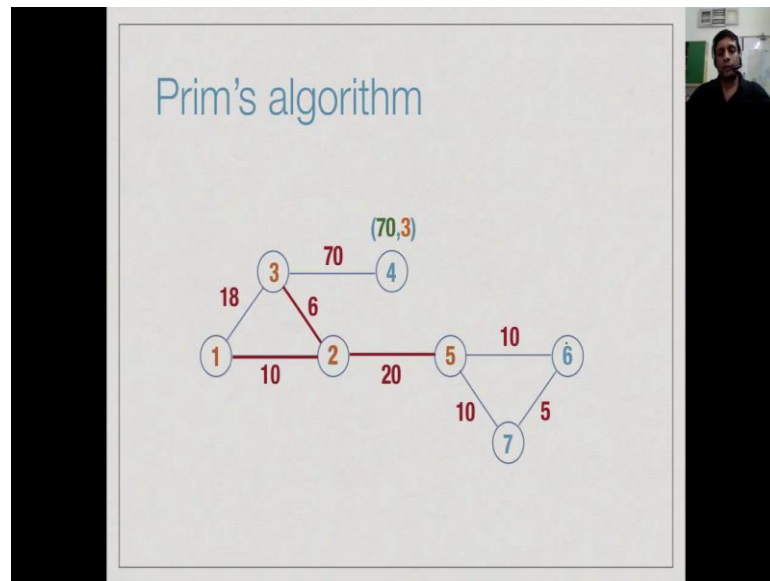
Once we add it, you will obtain the status of 4 because that is the only new label we do not update the status of 1 because 1 is already been added to the tree. We only look at those neighbours of tree which are not visited. So, now 4 gets the distance 70 with the neighbour 3, and then among these two,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

(Refer Slide Time: 15:55)

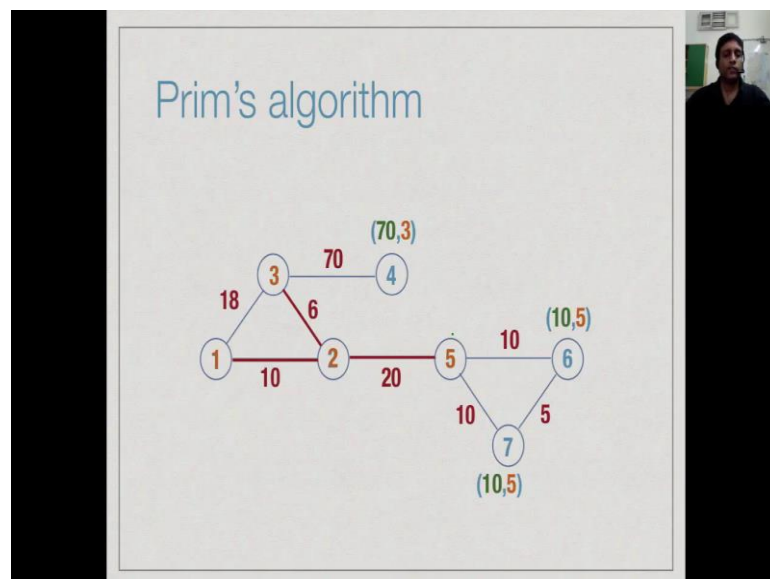


So, we will add 5 to our tree, and then we will update the status of 6 and 7.

(Refer Slide Time: 16:05)



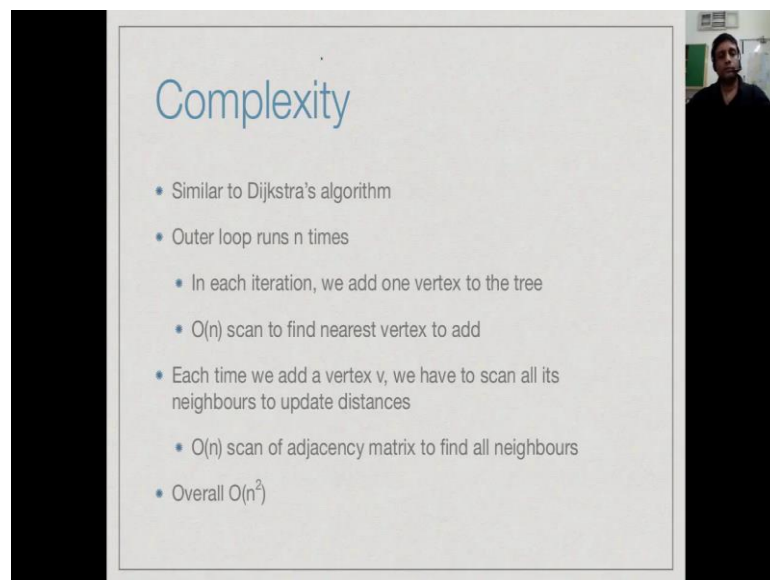
(Refer Slide Time: 16:10)



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)

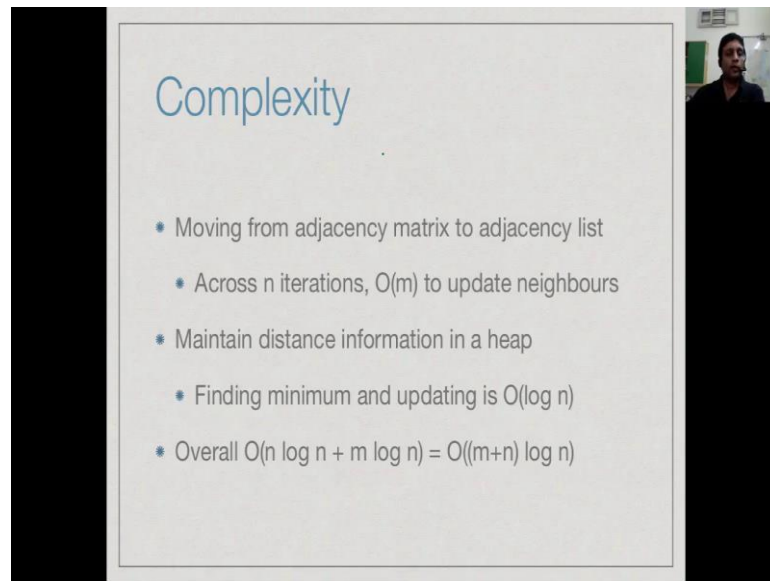


Complexity

- 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)

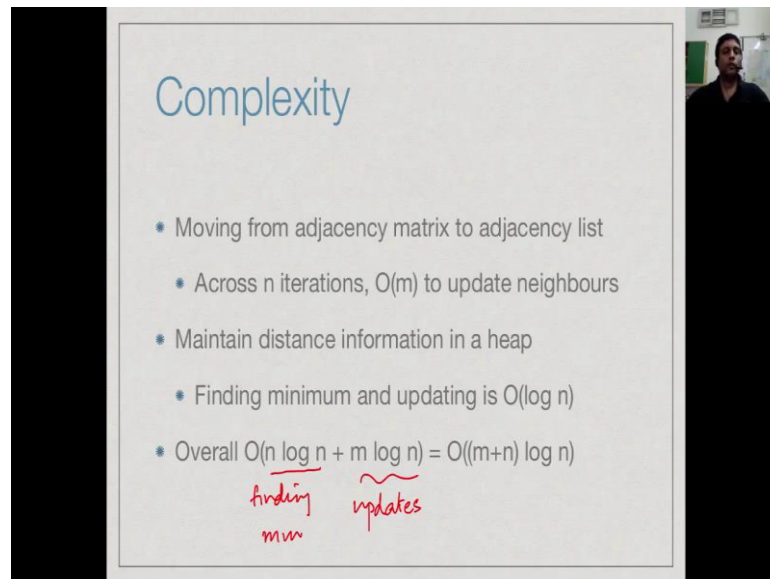


Complexity

- 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)

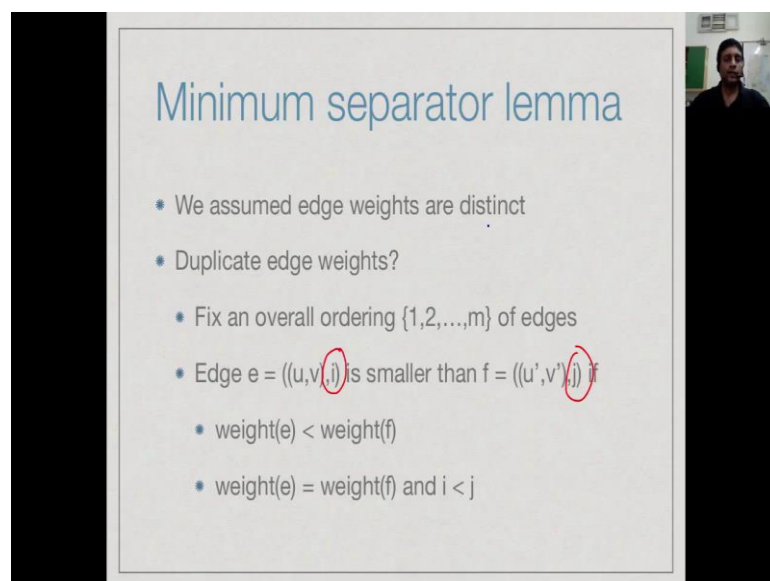


Complexity

- 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

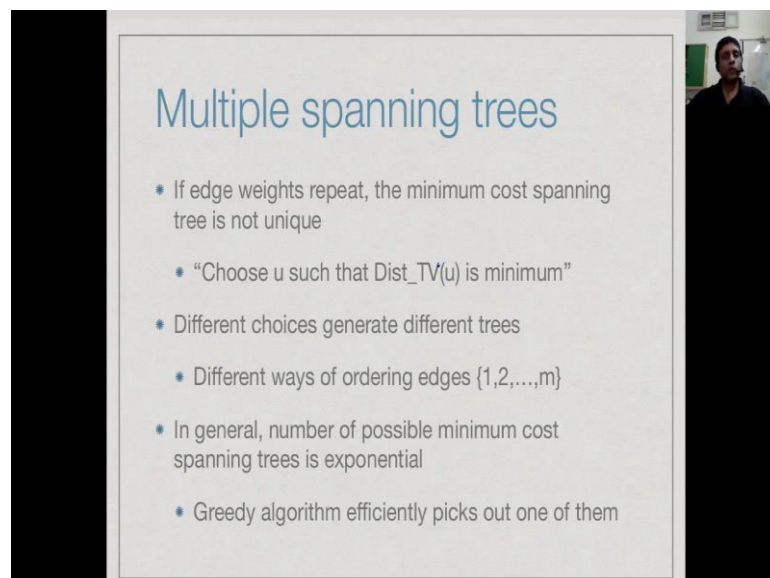
- We assumed edge weights are distinct
- Duplicate edge weights?
 - Fix an overall ordering $\{1, 2, \dots, m\}$ of edges
- Edge $e = ((u, v), i)$ is smaller than $f = ((u', v'), j)$ if
 - $\text{weight}(e) < \text{weight}(f)$
 - $\text{weight}(e) = \text{weight}(f)$ and $i < j$

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.