LESSON: GR3: Minimum Spanning Tree

***

(Slide 1) Greedy Approach

We're going to look now at the greedy approach for optimization problems.

So we're going to take a locally optimal move. In particular, we have a partial solution and we're going to take the optimal move for the next step and the question is, when does this approach lead to the global optimum?

Now, we saw earlier for the knapsack problem, that this greedy approach doesn't work. There are examples where the greedy does not lead to the optimal solution. Nevertheless, we were able to use Dynamic Programming to find the optimal solution for the knapsack problem.

What we're going to work on now, is the minimum spanning tree problem and we're going to see that for this problem, that the greedy approach does work, and the particular algorithm that we're going to use for the minimum spanning tree problem is the Kruskal's algorithm.

Many of you may have seen Kruskal's algorithm before, but the important thing, that we're going to stress in this lecture, is the proof of correctness of Kruskal's algorithm for the Minimum Spanning Tree problem.

Why exactly does the greedy approach work for this problem? We're going to see a general lemma known as the **cut property**. This lemma is going to imply that Kruskal's algorithm works correctly for the Minimum Spanning Tree problem. And it's important that we understand the statement of the cut property and also the proof has some useful ideas that I

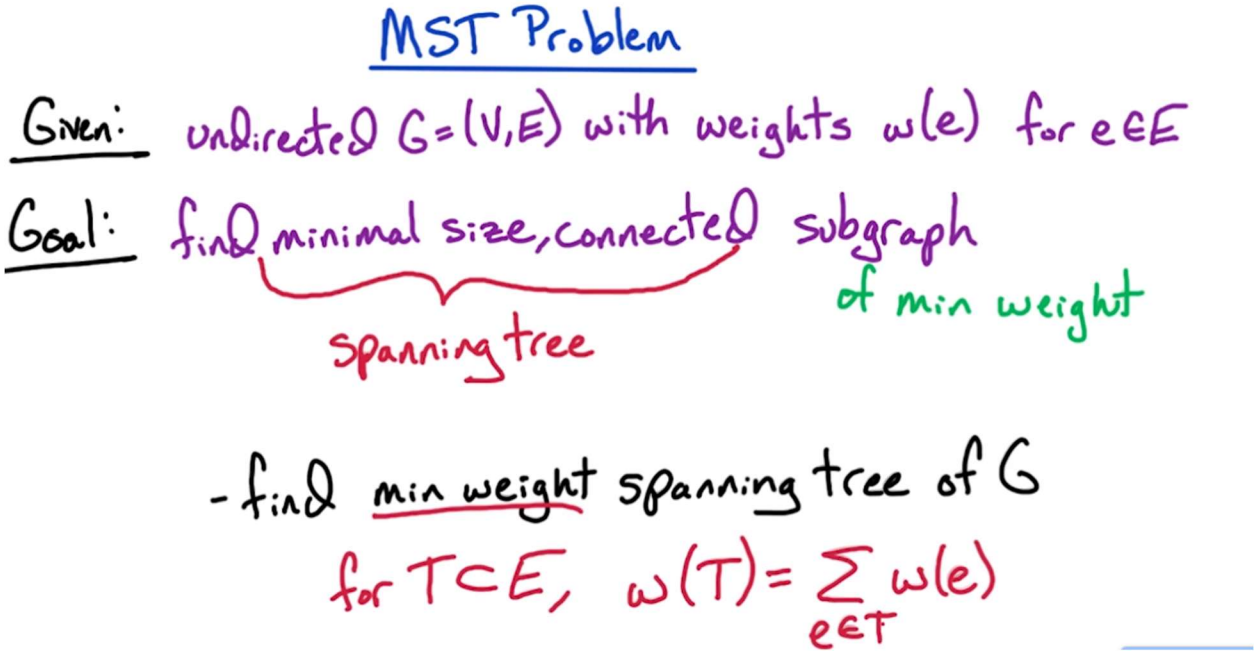
want you to understand. And we're going to see is a nice by-product of this general lemma.

We're also going to see that Prim's algorithm for the Minimum Spanning Tree problem also works correctly. So, this general lemma implies that Kruskal's algorithm works correctly and Prim's algorithm works correctly.

Let me remind you of the precise formulation of the Minimum Spanning Tree problem and then we'll look at the general formulation of the cut property - its statement and its proof - and then we'll see that it immediately implies that Kruskal's algorithm works correctly and Prim's algorithm also works.

***

(Slide 2) MST Problem



Footnote: See [DPV] Chapter 5.1 (Minimum Spanning Trees) and Eric's notes: https://cs6505.wordpress.com/schedule/mst/

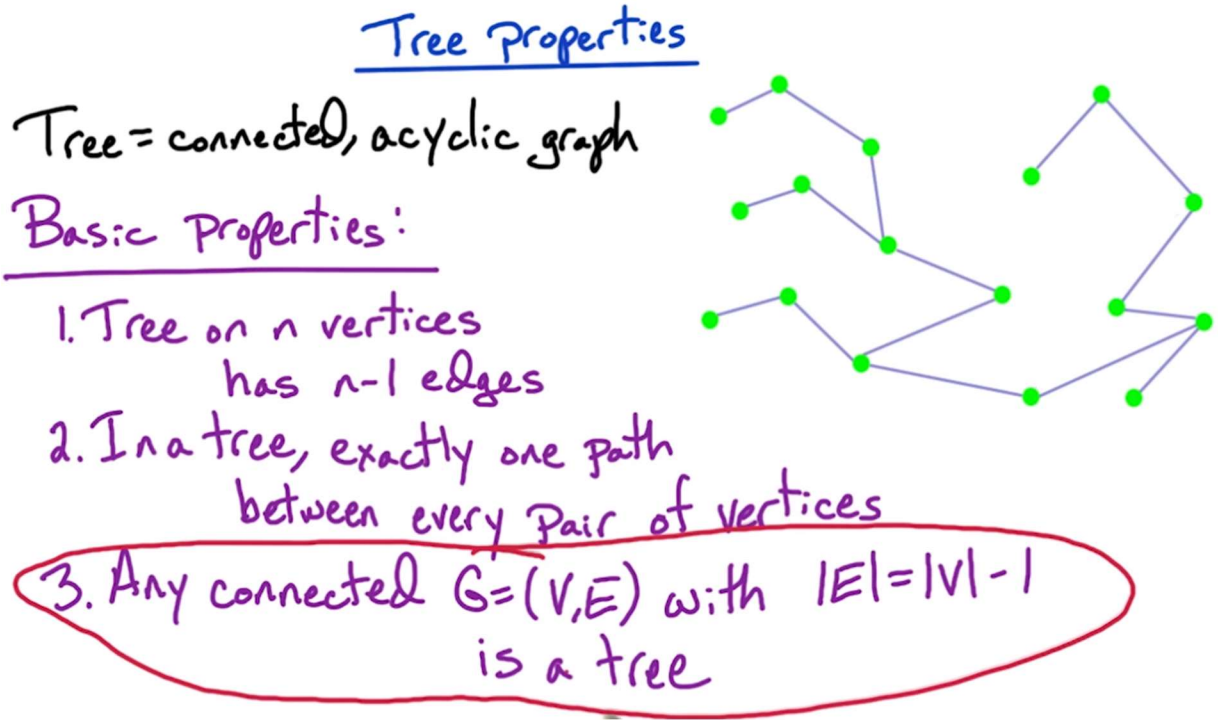
The input to the minimum spanning tree problem - also known as the MST problem - is an undirected graph, G, and each edge, e ϵ E, has a weight w(e). Our goal is to find the subgraph of minimal size which is still connected.

Now, the connected subgraph of minimal size is of course a tree. We call it a **spanning tree** to stress the fact that it's connected, that it's a tree and not a forest. And we want to find such a subgraph of minimum weight.

In summary, our goal is to find the minimum weight spanning tree of G. And, just to remind you what do we mean by **minimum weight**, well, for a particular tree, T, its weight is the sum of the weights of the edges in that tree.

***

(Slide 3) Tree Properties



Before we dive into the algorithm and the proof of correctness, let's review some basic graph theory about properties of a tree:

- Now, I would remind you, a **tree** is a connected acyclic graph.

  Here's an example of a graph on 17 vertices with a tree denoted by the blue edges. The first basic property is that a tree on n vertices has exactly n - 1 edges. In this example, it has 17 vertices and there are 16 edges. To see why this property holds, you'll notice that you need a least n - 1 edges to connect up the n vertices. And if you have more than n - 1 edges, then you'll have at least one cycle.

- The next property about a tree is that, in a tree, there is exactly one path between every pair of vertices.

  Clearly, if there are zero paths between a pair of vertices, then that graph, that tree is not connected. And also, if there are two or more paths between a pair of vertices, then we have a cycle in this graph. So, it's not acyclic. So, in order to be connected and acyclic, there has to be exactly one path between every pair of vertices.

- Now, the final property is the one that we're going to use in our proof of the cut property. We're going to construct a sub-graph and we want to establish that it's a tree.

How are we going to prove there's a tree? - well, we're going to show that it's connected and that the number of edges is exactly n - 1.

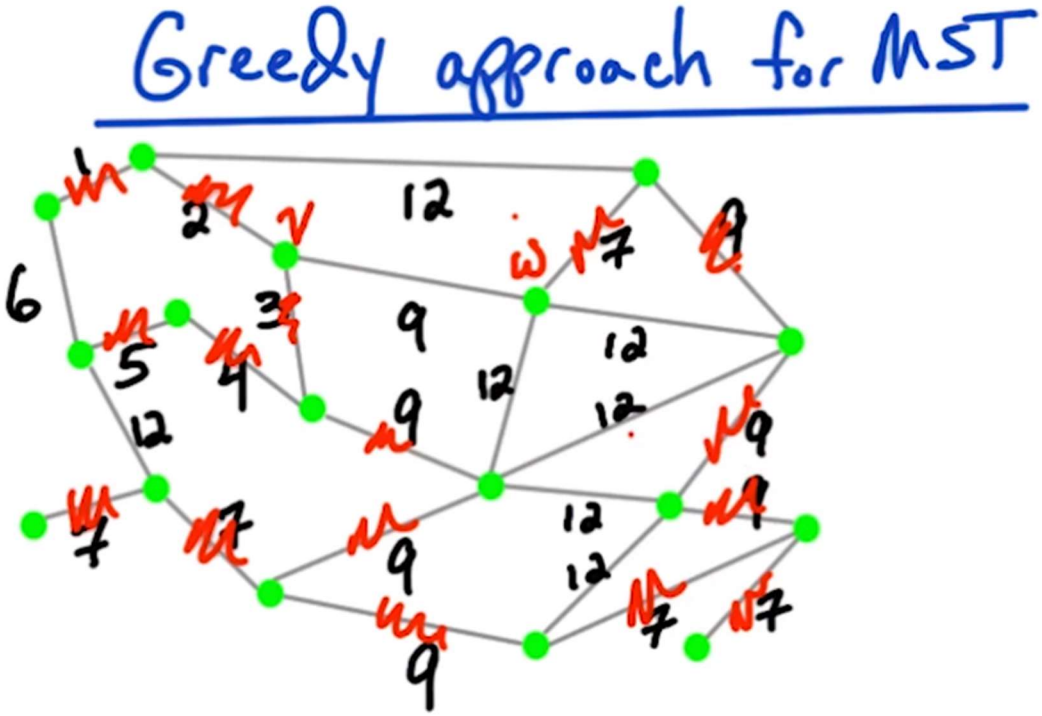
Any connected graph with exactly n - 1 edges is a tree.

Now, this third property follows from combining the first property and the second property together and apply the third property. This last property is the one that we're going to utilize later.

So, let me repeat it again. In order to establish that a graph is a tree, I just have to show you that the number of edges is exactly n - 1 and that this graph is connected.

***

(Slide 4) Greedy Approach for MST



Now, here's a particular graph. Let's put some weights on these edges and then let's see how we would approach the MST problem using a greedy approach.

Now, let's give the edges of this cycle the smallest weight, so we can focus on the behavior on this cycle. For the rest of the edges, I don't want to worry about so much. So, let me give you a bunch of edges weight 7, and I'll give a bunch of edges weight 9, and the remaining edges I'll give weight 12. Now, most of the weights don't matter for this example. I want to focus on this particular cycle. So, really, the first six weights are the only ones that matter.

Now, let's look at how I would approach the MST problem from the greedy perspective:

- In the greedy approach, the first edge I would consider is the edge of weight 1. I would start off with the empty graph. So, my current MST is empty and I'm going to build up the MST one edge at a time.

  I'll consider the edges by increasing weight. So, I'll start with the first edge of lowest weight, which is this edge of weight one. Now, I'm going to add it in if I can. Well, currently I have the empty graph as my current MST. And then, when I consider this

edge of weight 1, can I add it in? Yes, of course I can add it in because it doesn't create any cycles. So, let's add it in.

- Then I consider the edge of weight 2. Can add that in to my current MST? Yes, I can add it in because it doesn't create any cycles.

- Similarly, for the edge of weight 3, and edge of weight 4, edge of weight 5.

- Finally, I consider the edge of weight 6. Can I add that into my current MST? No, I cannot add this edge into my current MST because it creates a cycle.

- After that, I'll consider the edges of weight 7, in some order. But both of these edges can get added in. And similarly, these other three edges of weight 7 can also get added in.

- Now, let me consider these other edges of weight 9. This edge I can add and without creating a cycle. Similarly this edge.

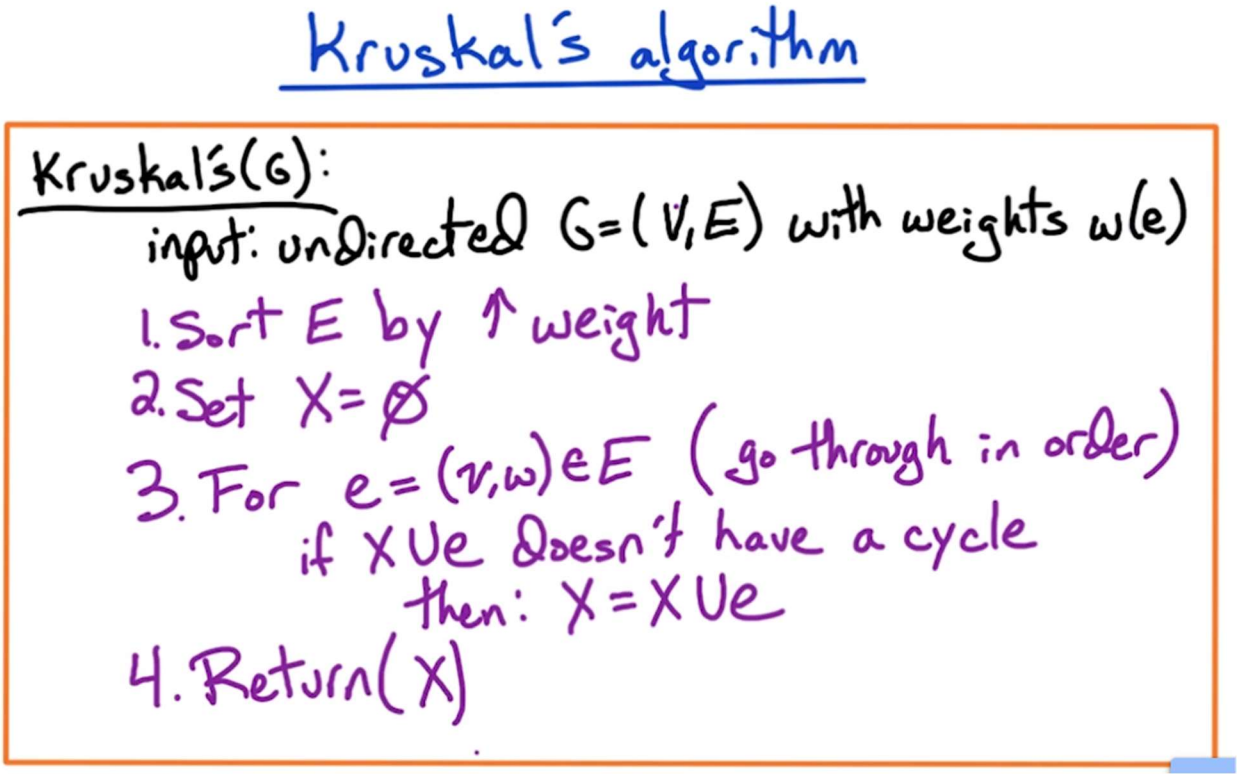
  Finally, when I consider this last I edge of weight 9. Can I add it in without creating a cycle? We'll say the end points of this edge are v and w. Notice that v and w are already connected. They lie in the same component. See, I have this path from v to w. So, since v and w lie on the same components, they are already connected. So, if I add an edge between them that creates another path and therefore I have a cycle which contains this edge. So, I don't add in this edge.

- And, similarly, any of the edges of weight 12, I don't add in.

And this is in fact an MST of this graph.

***

(Slide 5) Kruskal's Algorithm



If we formalize the greedy approach that we just outlined on the previous slide, this gives us Kruskal's algorithm for the MST problem.

So the input to Kruskal's algorithm is an undirected graph with weights on edges. Now we want to consider these edges of this graph in increasing order of weight. We want to consider the lowest weight edge first, and the highest weight edge last.

Kruskal's(G):

1. So our first step of our algorithm is to sort the edges by increasing weight. To do this, we can use something like mergesort.

2. We're going to use a set X to keep track of the set of edges that we've inserted so far into our MST. So we initialize X to the empty set.

3. Now we're going to go through the edges one by one:

   And we are going to go through these edges in order. The order is a sorted order by increasing weight. So the first edge we consider is the one of the lowest

weight and the last edge that we consider is one of the edges of highest weight. Now for a particular edge between V and W, when do we add this edge E into our current tree? We add it in if it doesn't create a cycle.

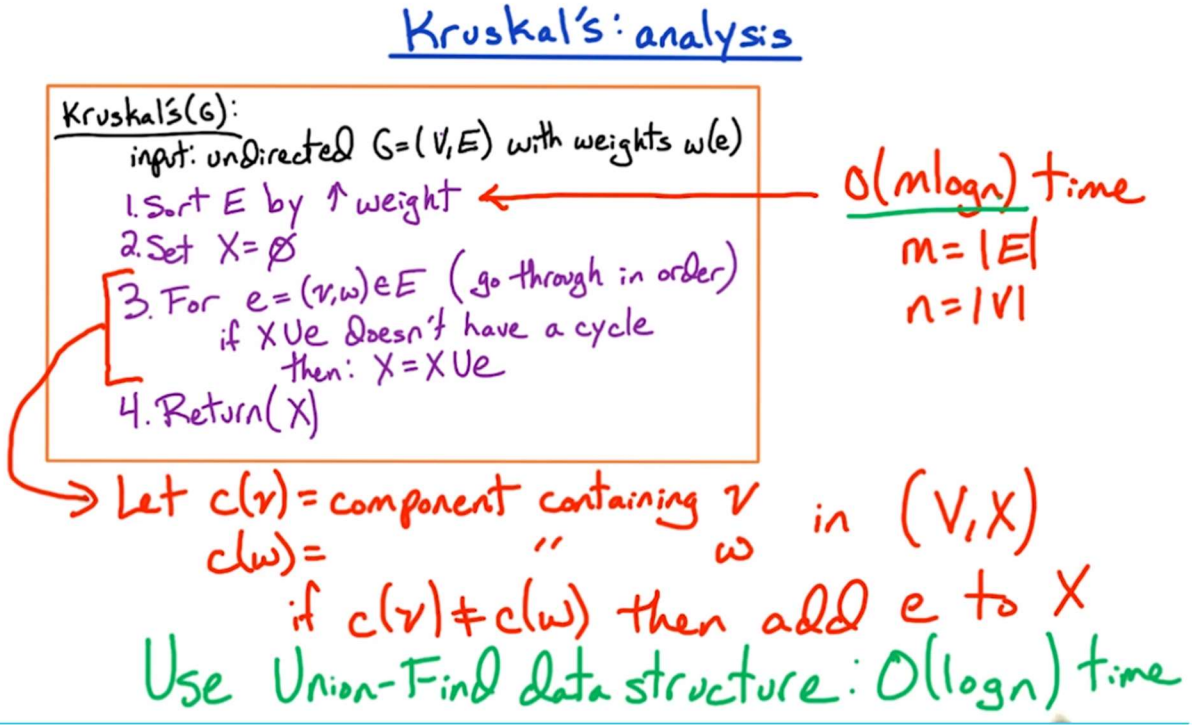
If adding this edge doesn't create a cycle, then we add this edge into our current tree X.

4. Finally, return X.

This will be an MST at the end of the algorithm.

***

(Slide 6) Kruskal's Analysis



Now let's take a look at the running time of Kruskal's algorithm and then we'll look at the proof correctness of Kruskal's algorithm.

1. (Step 1) How long does it take to sort the edges by increasing weight? This takes O(m log n) time. m is the number of edges and n is number of vertices. Note of course (m log n) is the same as (m log m).

2. (Step 3) Now how long does it take to do this step? How long does it take to check whether adding edge E into X creates a cycle or not? What exactly do we want to check to see whether it creates a cycle or not?

   Look at the subgraph {V,X} on this set of edges X. Now in this sub graph, let c(v) be the component containing v, and c(w) is a component containing w.

   Now to see whether the edge e creates a cycle when we add it into this subgraph, we want to check whether there's already a path between v and w. If there already is a path between v and w, then the components containing v and w are the same - v and w are in the same component. So what this step does is, it checks whether the component containing v and the component containing w are different. So if v and w are in different components, then we add the edge E into X.

Now how do we check the component containing v and the component containing w? Well, we use this data structure, the simple data structure known as Union-Find data structure. Using this data structure, it takes O(log n) time per operation. So it takes O(log n) time to check the component containing v and the component containing w and then we can see whether they're the same or different components.

And then, once we add this edge e into X then we can update the component containing v and the component containing w. We can merge those two components in O(log n) time.

So the Union-Find data structure takes O(log n) time for each check operation, in order to check whether the component containing v and the component containing w are the same or not.

And it takes the O(log n) time to do a merge operation, where we merged the component contained v and the component containing w because we added edge e into X.

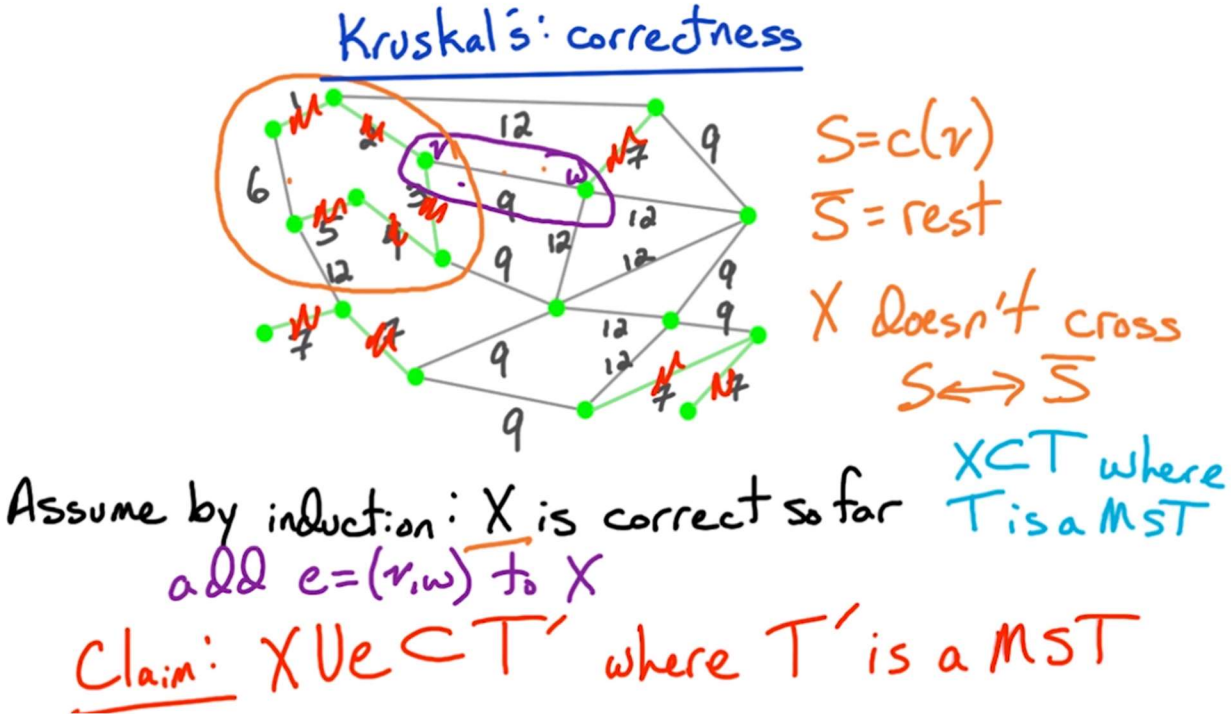
Now I'm going to skip the details of the union-find data structure, since many of you may have seen in the data structures class before and if you want, if you haven't seen it before then you can review it in the textbook. But it's a very simple data structure using this notion of rooted directed trees.

3. Once we have this data structure which has O(log n) per operation, then the key fact is that we're doing at most m operations and then since there are O(m) operations each one taking O(log n) time, then the total run time for the Step 3 is O(m log n) again. So step 1 and step 3 both take O(m log n) time. So, the total run time is O(m log n) for the whole algorithm.

Now what I want to focus on in this lecture is not the data structure but the proof of correctness of this algorithm. Why does the greedy approach work for this problem?

***

(Slide 7) Kruskal's Correctness



Now, let's get the idea for the proof of correctness of Kruskal's algorithm by looking at our earlier example.

So, recall what Kruskal's algorithm is going to do. It's going to consider these edges by increasing weight. So, we're going to add in these first five edges. The sixth edge we're going to skip because it creates a cycle. Then we add in these five edges of weight 7. And then finally we're going to consider one of these edges of weight 9.

Now, which edge of weight 9 do we consider first? Well, that's an arbitrary choice. Let's suppose we consider this edge of weight 9 first. Let's mark the end points as v and w. Now in general how are we going to prove that this algorithm is correct? Well, we're going to do it by induction.

So, let's suppose that the algorithm is correct so far, so that the current X is correct so far. What does that mean to be correct so far? That means that these edges are part of an MST. So, there is MST T and X is a sub-graph of T. So, we're on our way to an MST to a solution to this problem. So, we're assuming that these red edges, which I've now highlighted as green edges, are part of an MST.

Now let's suppose that we add this edge that we're considering right now to this sub-graph X.

If we don't add this edge to the sub-graph X, then X is still part of MST. But, if we add this edge e to this sub-graph X, then we want to make sure that X U {e} is part of MST. And that's our claim that we're going to prove.

We're going to prove that $X \cup \{e\}$ is part of MST T'. T' might be different than T. So, we might be heading in the direction of a different MST than we were headed for before, but we're still on our way to an MST.

Now, what is the key property that ensures that adding e to X ensures that we're still on our way to an MST? - well, when are we adding this edge e to X … when we look at the component containing v (which are these six vertices) and we look at the component containing w (which are these two vertices) - these are different components. Therefore, we can add this edge and it doesn't create any cycles.

Now, the key property is that if we consider one of these components … let's say the component containing V … and we let S = c(v), the set of vertices and the component containing v. An $\bar{S}$ is the complement of this set. So that's the rest of these vertices … then, the key property is that our current subset X has no edges which go from S to $\bar{S}$. If there was an edge that goes from S to $\bar{S}$, then that vertex in $\bar{S}$ which is connected to S would be added into S.

So, it would be in the component containing v. This component is a maximal set. So, we know that nobody in $\bar{S}$ is connected to anybody in S. Therefore, there are no edges of X going from S to the rest of the graph - that's the definition of a component.

So X has no edges that go between S and $\bar{S}$. So, X doesn't cross between S and the rest of the graph, $\bar{S}$. But this edge e does cross between S and $\bar{S}$, because $v \in S$ and $w \notin S$.

Now, what is the key property about e? e is a minimum weight edge crossing from S to $\bar{S}$.

Now, there are several edges of minimum weight crossing between S and $\bar{S}$ (two, in particular). We can take any of those minimum weight edges. For any minimum weight edge crossing between S and $\bar{S}$, if we add e to X, then X U {e} will be part of an MST. That's what we're going to prove.

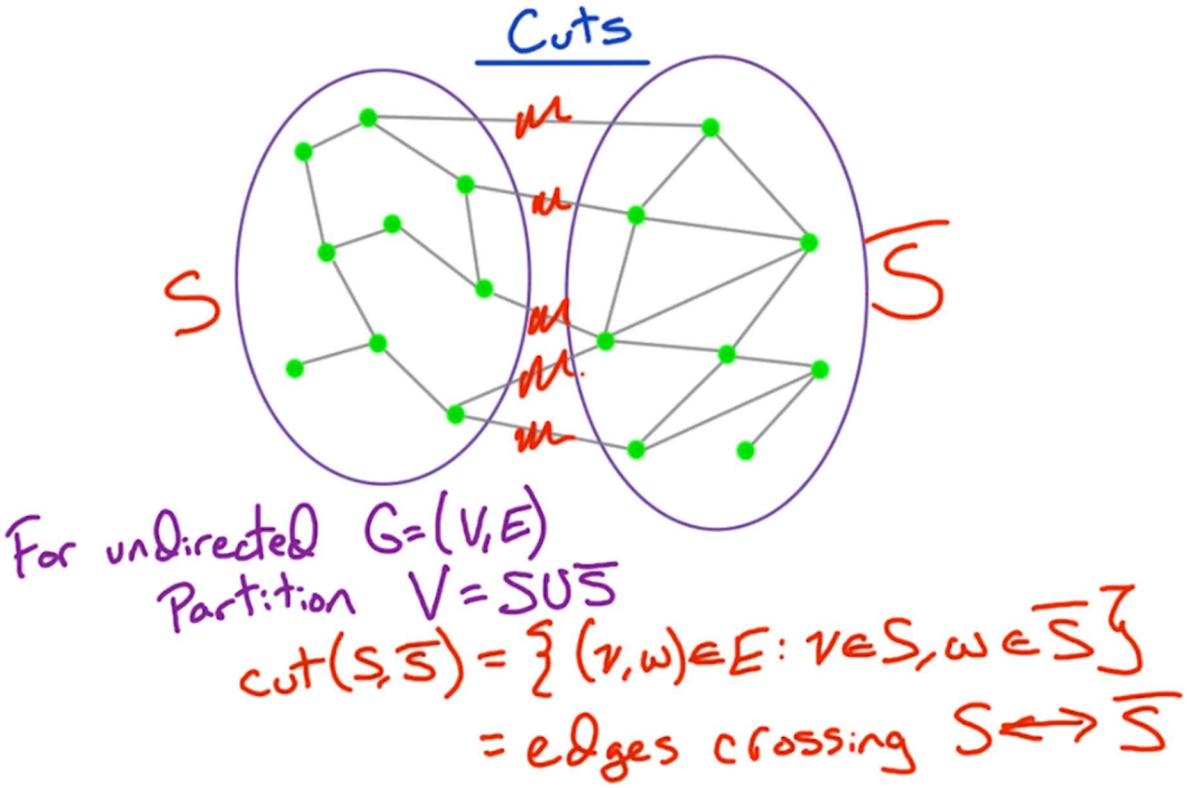
And we're going to prove in general that if we take a subset of edges, which are part of an MST and we take an edge which crosses from S to $\bar{S}$ of minimum weight and no other edges of this current subset cross from S to $\bar{S}$ - So, this minimum weight edge crossing from S to $\bar{S}$ - then we can add this edge into our current subset and that will still be part of MST. This general claim

will be known as the cut property.

Before we go ahead and formalize the cut property statement, let me formalize what I mean by a **cut** - this partition of vertices into S and $\bar{S}$.

***

(Slide 8) Cuts



Before we give the statement of the cut property, let's do a little bit of terminology:

- Here again is our running example of a graph. In general, we have an undirected graph, G and a **cut** of the graph is a set of edges which partition the vertices into two sets, S and the complement of S.
- So for a particular set S, the **cut edges** are those edges with one endpoint in S and the other endpoint in $\bar{S}$. In words, what are the cut edges? The cut edges are the edges crossing between S and $\bar{S}$.

  Let's take a look at a particular cut in this graph. So, I partitioned it into S and $\bar{S}$. Now what are the cut edges? The cut edges are the five edges which are crossing between S and $\bar{S}$. These five edges are the cut(S, $\bar{S}$).

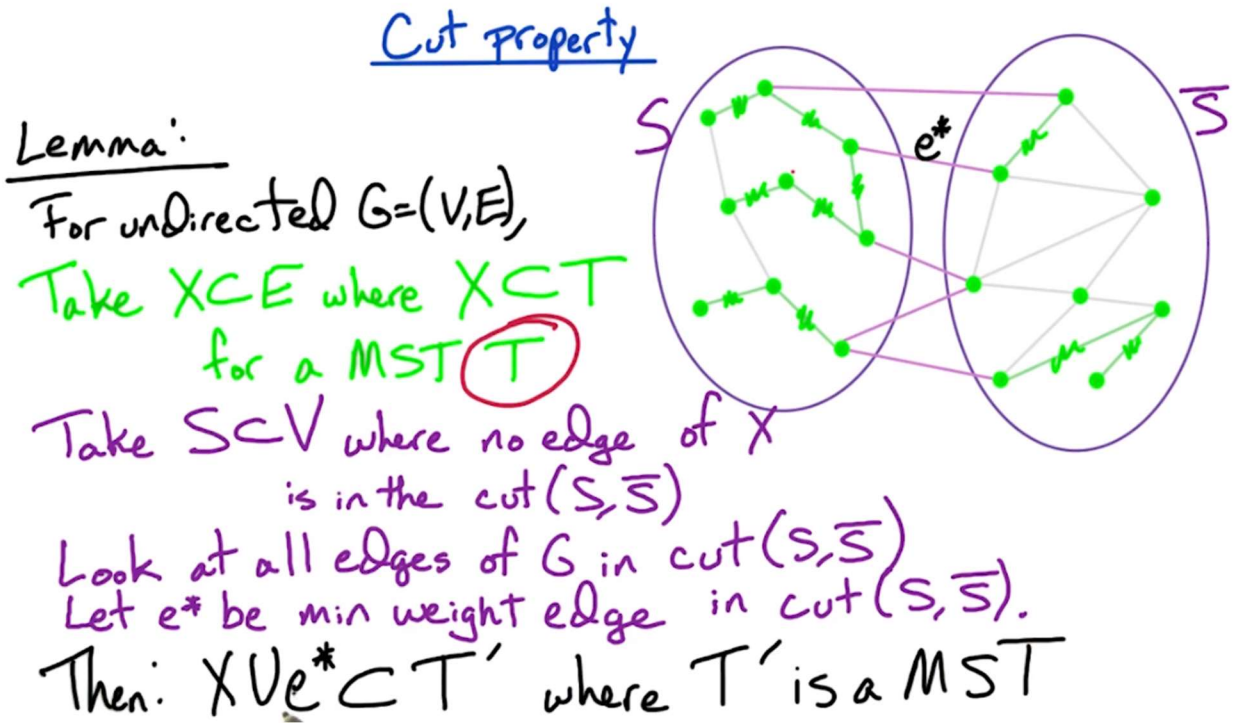
Later in the course, we're going to look at various optimization problems involving cuts. We're going to try to find a minimum cut. So the fewest number of edges in order to disconnect the graph into at least two components.

And we're also going to look at the Max Cut problem where we're trying to find the cut of largest size. But, for now we're just looking at the general notion of what is a cut.

So to summarize, if I give you a set S and $\bar{S}$, then the cut(S, $\bar{S}$) is a set of edges crossing between S and $\bar{S}$.

***

(Slide 9) Cut Property



Finally, I'll state the structural Lemma known as the cut property. This is a Lemma from which we'll derive the correctness of our MST algorithms.

So, the Lemma considers an undirected graph G. So, let's consider the graph that we've been having as a running example.

Now we're going to take a subset of edges. I've marked a particular subset of edges in this graph by the green edges here. Now we're going to assume that this subset of edges is part of an MST T.

> Now you should think of this set X as if you have an algorithm for the MST problem, and you have a partial solution that you're building up edge by edge. X is your partial solution so far. And you assume by induction that this partial solution is correct so far.

So to be correct so far - that means that this subset X that you've constructed, so far, is a subset of MST T. Now, you don't know what T is. But you know that your solution is correct so far. So it is a subset of some MST T.

So, for this particular example, I have the green edges, which are marked here. And these are

what my algorithm has constructed so far. And we're assuming that our partial solution X, the green edges, are part of an MST T, which I marked by red edges here. So the red edges are an MST T, and we don't know this MST. But we have the green edges. And we just know by induction that the green edges are part of an MST.

So we have a partial solution, which is correct so far. And our algorithm is working edge by edge. So, we're going to consider the next edge that we're going to add into X. And we want to prove correctness of this next edge being added into X.

Now, what are we going to assume about the edge that we're trying to add into X? We're going to assume that it crosses a cut of this graph. So in particular, we're going to take a subset S, and we're going to look at the cut between S and $\bar{S}$.

So, here's a particular set S, and here's a compliment of S. Our assumption is that no edge of our partial solution (X), so far, crosses this cut(S, $\bar{S}$). So, notice that no green edge crosses between S and $\bar{S}$.

> I notice there are many sets S where this is true. But we get to choose any set S where no green edge crosses between S and $\bar{S}$.

Now, there are no green edges crossing between S and $\bar{S}$. But, there is at least one edge of the graph G which crosses from S to $\bar{S}$ - because it's a connected graph. So we're going to look at all edges of the graph G, which cross the cut(S, $\bar{S}$).

So I've marked, in pink or purple, the five edges of the graph G, which cross between S and $\bar{S}$. Now, of these five edges, I want to take a minimum weight edge.

So, let e* be a minimum weight edge. Now, there might be multiple edges of minimum weight. It doesn't matter. You get to choose which one you want. So let e* be any minimum weight edge across this cut. So, let's mark this particular edge as e*. So, we're assuming that these four other edges crossing the cut, have weight which is at least the weight of e*.

Now, finally, what's our conclusion? Our conclusion is that we can add e* to our current MST construction, and we'll still be on our way to an MST. So, if we look at our partial solution X, and we add e* to it - so we look at X U {e*} - then this is a subset of T', where T' is an MST.

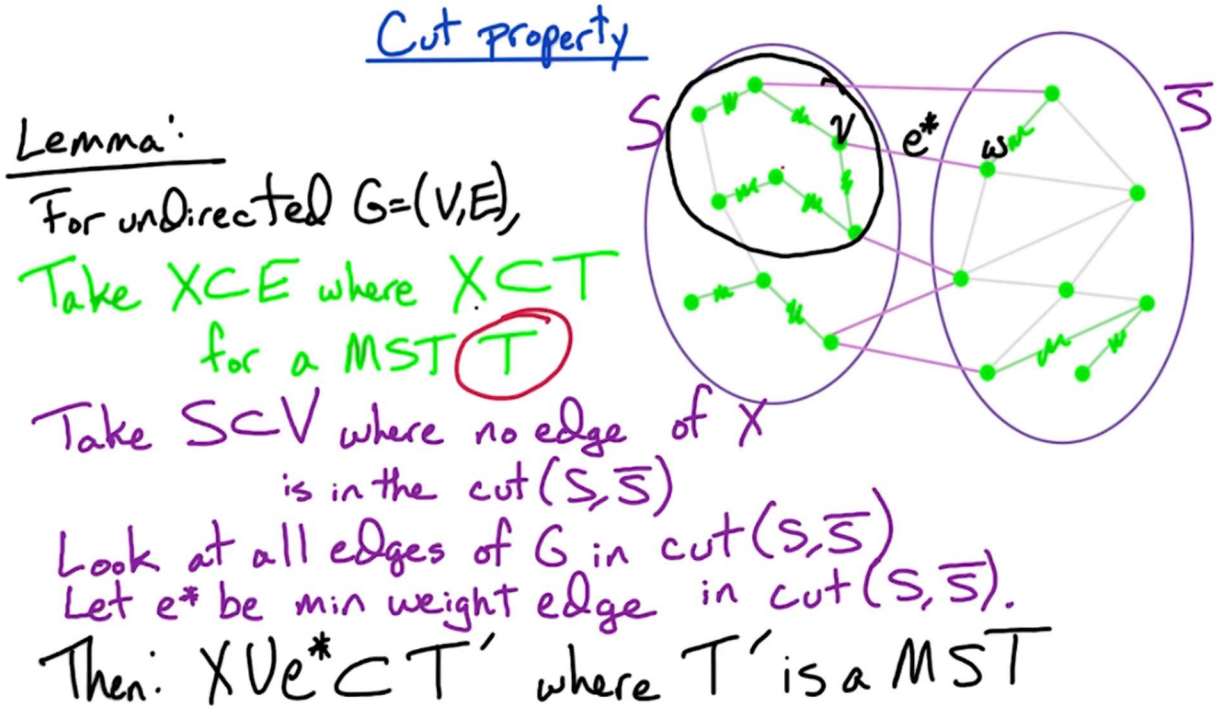
So if our partial solution X was on its way to an MST T - we didn't know T - but we just knew that it was correct so far. Then if we add an edge - which is the minimum weight edge across

the cut - and no edge of the partial solution crosses this cut so far - then, if we add this edge e*, to our partial solution, then we'll still be on our way to an MST T'.

Now, this MST that we're going towards now, it might be different than the previous MST. But our goal is just to find a MST. We're not looking for a particular MST. So, if we were on our way to an MST before (i.e., our partial solution was correct so far), then we'll be on our way to an MST by adding this edge e* in.

***

(Slide 10) Cut Property: Kruskal's



Now, how does this apply for Kruskal's algorithm?

For Kruskal's algorithm, we said this edge e* was between vertices v and w. We added this edge e*. If the component containing v, one of the endpoints, was not the same as a component containing w - so v and w were not connected in the current subgraph - So what was the set S in this example? The set S in this example was the component containing v or the component containing w. Let's just say that S was a component containing v and therefore since the component is a maximal set of connected vertices, we know that there is no edge in the partial solution which crosses from S to $\bar{S}$. And then since we consider the edges in increasing order - in sorted order - then we know that e* must be the minimum weight edge crossing from this component to outside.

So this edge e*, that Kruskal's algorithm considers, satisfies the hypothesis of the Lemma. So if we add an e* to our partial solution, we'll still be on our way to an MST. So this proves correctness of Kruskal's algorithm. Kruskal's algorithm uses a particular type of set S, but this lemma is true in general for any cut S.

Now it's important to understand the statement of the cut property and also to understand the proof. The statement, the main idea is that any edge which is minimum weight across a cut is

going to be part of some MST. Why? Because if you give me a tree T which does not contain this minimum weight edge across the cut then I can add this edge into the current tree and I can construct a new tree which is of smaller weight.

Actually, to be precise, I'm not necessarily going to construct a new tree with a smaller weight, but if you give me a tree T which does not contain this edge e*, or construct a new tree, T', where the weight of T' is at most the weight of T. So the weight doesn't increase. And, that's going to be the intuition for the proof.

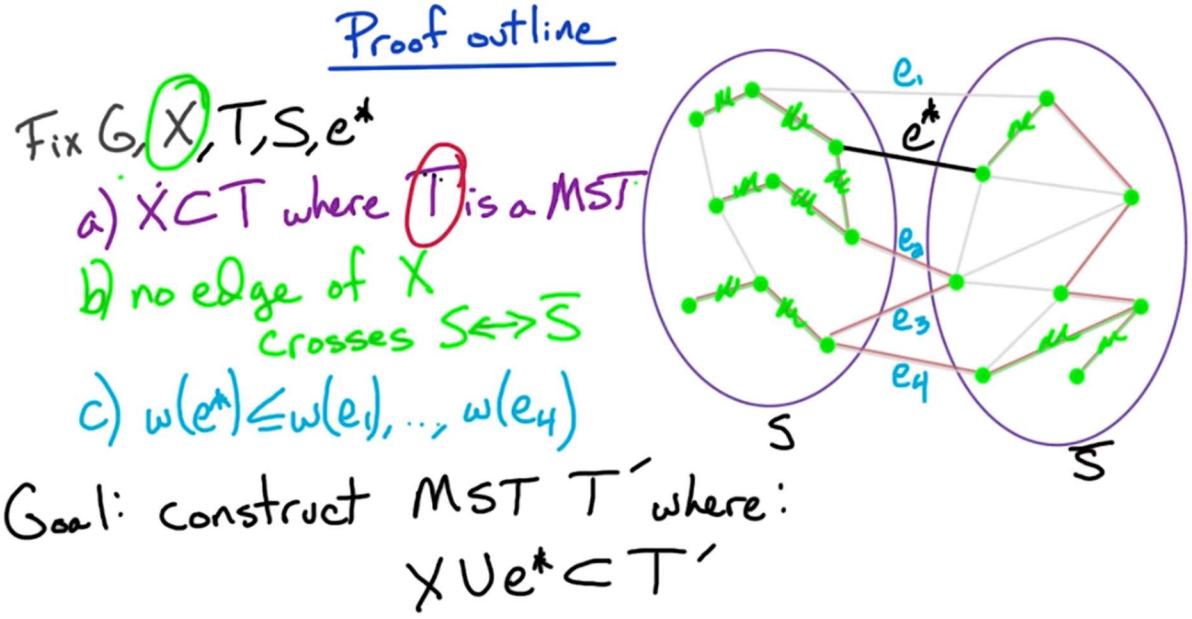
We're going to take this tree T, we don't know this tree T but we know there exists an MST T which contains X, our partial solution so far. And now we're going to add this edge e* into X. If this MST T contains X U {e*} then we're all done.

Now, there's no reason why this edge e* has to be in this tree T. So what happens if edge e* is not in the tree T? Well, then, we're going to construct a new tree, T', where the weight of T' is at most the weight of T, and in this tree, T', will be the edge e* and also X, and we'll construct this tree T' by modifying the tree T slightly by adding in the edge e* and then removing an edge in order to construct a new tree.

So, let's go ahead and dive into the proof of the cut property.

***

(Slide 11) Proof Outline



Now let's outline the proof of the cut property.

Fix G, X, T, S, e*:

- Let's first fix a graph G. Here's our running example of a graph.
- Let's fix the subset of edges X which are past solutions so far. These are the green edges here. Our assumption is that X is a subset of T, where T is an MST.
- Let's mark this tree T by the red edges. Now we don't know these red edges. We don't know this MST T; but, we know there exists such a tree T. So even though our algorithm doesn't know this tree T, we can use it in the proof because we are assuming the existence of this tree T.
- Now, in addition to this tree T, I want to fix the subset of vertices which form the cut which no edge of x crosses. So, I want to fix this subset S.

  In this example, this is X and this is $\bar{S}$. Now, our assumption was that none of the green edges - none of the edges of X cross between S and $\bar{S}$.

  Now, here are the green edges once again. And notice none of these green edges cross between S and $\bar{S}$. Now there are edges of T which cross between S and $\bar{S}$. Now that has to happen because T as an MST, is connected. So it has to connect S to $\bar{S}$.. But we're just asking that no edge of the partial solution crosses between S and $\bar{S}$. So no edge of X crosses S to $\bar{S}$. No edge of X is part of the cut between S and $\bar{S}$.

- Now we're going to take a particular edge e* which crosses this cut.

  Which edge are we allowed to choose? We're allowed to choose any edge of minimum weight across this cut. So there are four other edges crossing this cut, and our assumption is that the weight of e* is at most the weight of any of these other edges. Now, they may be equal; but, no other edge crossing this cut has strictly smaller weight than e*.

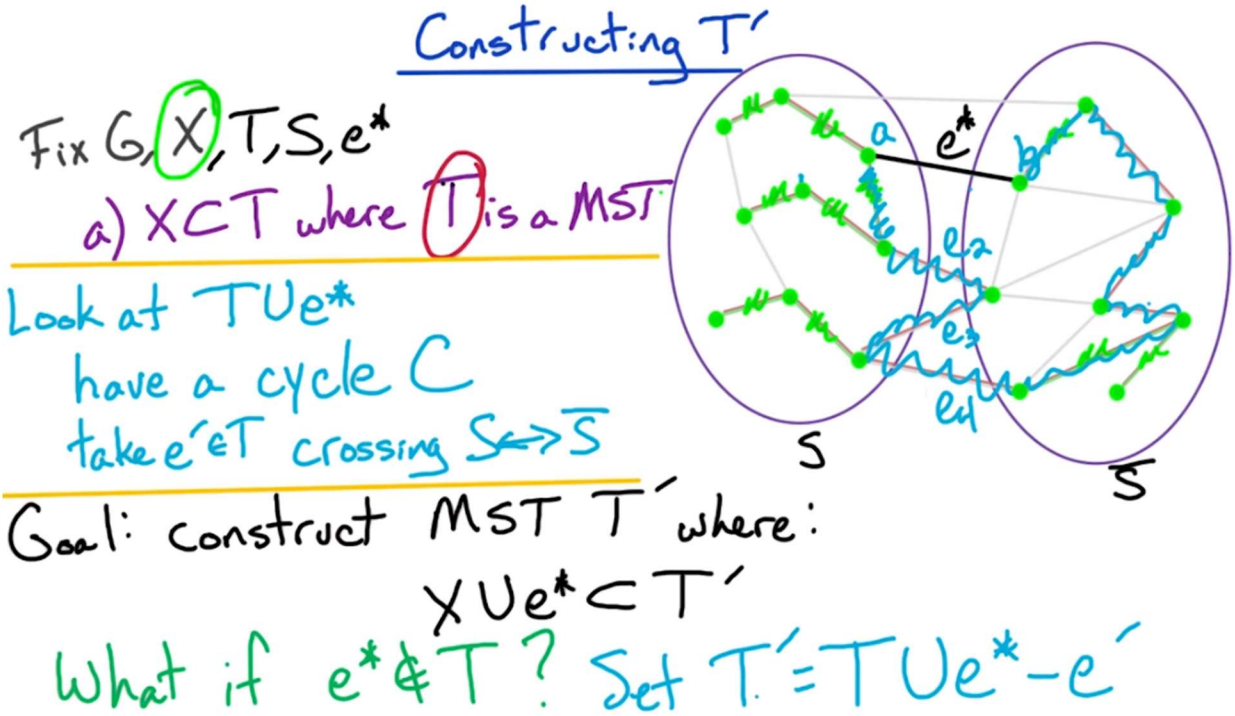
Now, finally, what's our goal in this proof? - our goal is to construct a tree T', where T' is a minimal spanning tree and where X U {e*} are all part of T'. So X U {e*} are a subset of T'.

Now how can we possibly construct this T'? - well, we have this tree T. T is an MST and T contains the partial solution X. Now the problem is that this tree T might not contain this edge e*. For example, in this case, e* is not in one of the red edges. So what do we do? Well we try to modify this tree T to construct a new tree, T', and we'll show that this new tree T' is a minimal spanning tree.

So the weight of T' is at most the weight of T - actually the weights must be equal - and this new tree T' is modified a little bit from T so that it contains e star in addition to containing the partial solution X.

***

(Slide 12) Constructing T′



Once again, our goal is to construct this MST T′, which contains X and e*.

Now, know there is an MST T which contains X. Now, there are two cases to consider either this edge e* is a part of T, or is not part of T:

1. The easy case is suppose that e* happens to be part of T. Then, in this case, what is T′?

   T′ is the same as T. There's nothing to do in this case. Notice that we have that X is part of T. And we're supposing that e* happens to be part of T as well.

   What's our goal? Well, our goal is to show a tree, T′ which is an MST while T is an MST. If we set T′ to be T then we have that T′ is an MST. And we know that X is part of this T or T′. And also e* is part of it. So we have that X U e* is part of T′. So there's nothing to prove in this case. Done.

2. The hard case is, what if e* is not part of T?

   In this particular example, this e* is not part of this particular MST T. So what do we do in this case? Well, we have to modify T in order to add edge e* into T and construct a new MST T′.

How do we construct this tree T′ in this case? Well, let's take this tree T which contains X, and let's add in the edge e* to it. Let's look at T ∪ {e*}.

We're taking the red edges in this graph and we're adding in this edge e*. Let's say that the endpoints of e* are a and b. Now, T is a tree. What happens when we add an edge to this tree? Well, it creates a cycle. It's going to be a cycle which contains this edge.

Now, this tree contains another path between a and b. The path between a and b in the tree T goes along here, along the blue edges. And then, when we add in this edge e*, we get a cycle containing edge e*. Let's call this cycle C.

Now, I want to drop an edge from this cycle in order to make a tree T′. We'll show that we can drop any edge from this cycle and we'll have a tree T′. But we want it to be a minimum spanning tree so we want the weight of the T′ to be minimum.

How do we show there's a minimum weight spanning tree? - well, we know that the weight of T is minimum. If we show that the weight of T′ is at most the weight of T then T′ is also of minimum weight. We need that the edge of the graph that we drop has weight at least that of e*.

Which of these blue edges has weight which is at least that of e*? - well, what was special about e*? e* was chosen to be the minimum weight edge across this cut S to $\bar{S}$. In this particular example, there are three other edges - three other blue edges crossing this cut S to $\bar{S}$.

So what do we know about these three blue edges crossing this cut compared to e*? - we know that the weight of e2, e3, and e4 is at least that of e*. So we can drop any of these three. We take any of these three edges, let's call it e′, which is in the tree T and which crosses S, $\bar{S}$. Now, in this example once again there are three edges crossing this cut S S-bar which are in this tree T.

Now, in general, why do we know that there is an edge e′, which is in the tree T which crosses S, $\bar{S}$? - where this edge e* has one end point in S and one end point in $\bar{S}$? - now, this tree T - it's a tree! Once again, that means it's connected. That means there's got to be at least one path between A and B.

Now, that path has to cross from S to $\bar{S}$ at some point. In this example, it crosses multiple times, but it's got to cross at least one time from one side to the other side. Now, when it crosses from one side to the other side, it has to have an edge, which crosses from one side to the other side. So take one of those edges, which crosses from

one side to the other side. Call that e′.

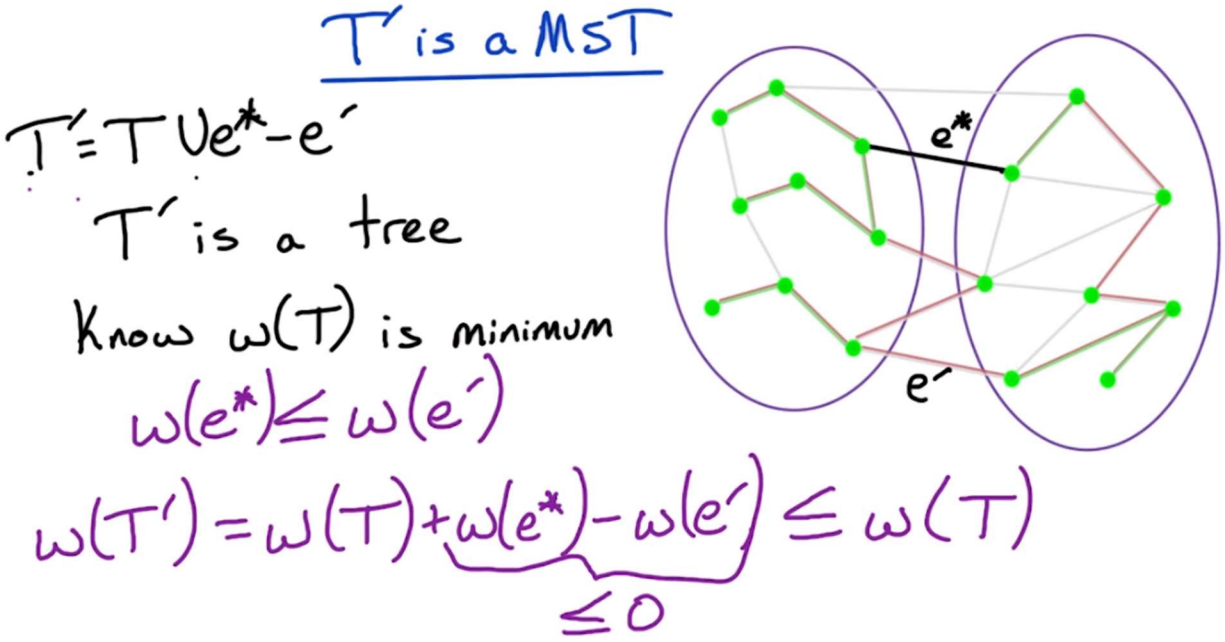
And finally, we have our tree T′. We're going to set T′ to be T, which is MST, which contains the partial solution X. We're going to add in this edge e*, which is the edge we're trying to add into our partial solution. And then we're going to subtract - we're going to take away any edge which crosses from S to $\bar{S}$, which is part of this tree T.

And we know there exists at least one such as e′. Take any such edge e′ and take it out of the tree T, add in the edge e*, and this gives us our tree T′.

What remains? Well, we've constructed this T′ and now we have to prove that T′ is in fact a tree and that is a minimum weight. Therefore, it's a minimum spanning tree.

***

(Slide 13) T′ is a Tree



Now, let's first prove that T′ is a tree.

How do we construct T′ once again? We took the tree T and we added in this edge e* that created a cycle. Now, along that cycle we removed any edge of that cycle. In this case we removed, let's say, this edge and let's call it e′. What we're going to show now is that we can remove any edge of this cycle and this result in graph T′ will be a tree.

Now, how are we going to show that the′ is a tree? Well, we're going to show that T′ is connected and it has exactly n-1 edges. Now, if you recall from the beginning of the lecture, we said that if a subgraph has exactly n-1 edges and is connected, then it must be a tree.

Now the fact that it has exactly n-1 edges is obvious. Why? Because T is a tree so it has exactly n-1 edges, we added one edge in and removed one edge. So we still have exactly n-1 edges.

So what remains? We just have to show that it's connected. So take any pair of vertices y and z and let's show that there's a path between y and z in this subgraph T′. I chose a particular y and z in this example. So y is over here and z is over here.

Now noticing T, there is a path between y and z, of course, because T is connected. The path goes along this edge e′. Now what happens in T′? In T′, I've removed this edge e′. So, this old path between y and z no longer exists in T′.

So let's let P denote the path from y to z in T in the original tree T. And now we have to construct a path in T′ which goes from y to z. Now, recall that when we looked at T U {e*} what do we get? We got a cycle, these blue edges are the cycle in T U {e*}.

So let C denote the cycle in T U {e*}. Let's denote the end points of e′ as c and d. Now e′ is a path from c to d in T. Now, this cycle C has two paths between c and d, e′ is one of those paths, let's take the other path. So let's look at this cycle C and take out the edge e′, what do we have? We have a path and this path goes from c to d and all these edges in this path exist in T′ because T′ contains all these edges except for e′ but we're not using e′, we took e′ out. So we took the cycle, dropped off the edge e′ and this gives us a path from c to d in T′. So, how do we get from y to z in T′? Well, we're going to follow the original path P for whenever we hit this edge e′, we replace this edge e′ by this new path P′.

Now, to show that y and z are connected in T′. So we want to show that there is a path from y to z in T′. Well, we're going to use the original path in T and then whenever we hit the edge e′, we're going to replace the edge e′ by this path P′ because the edge e′ does not exist in T′ but the path P′ does exist.

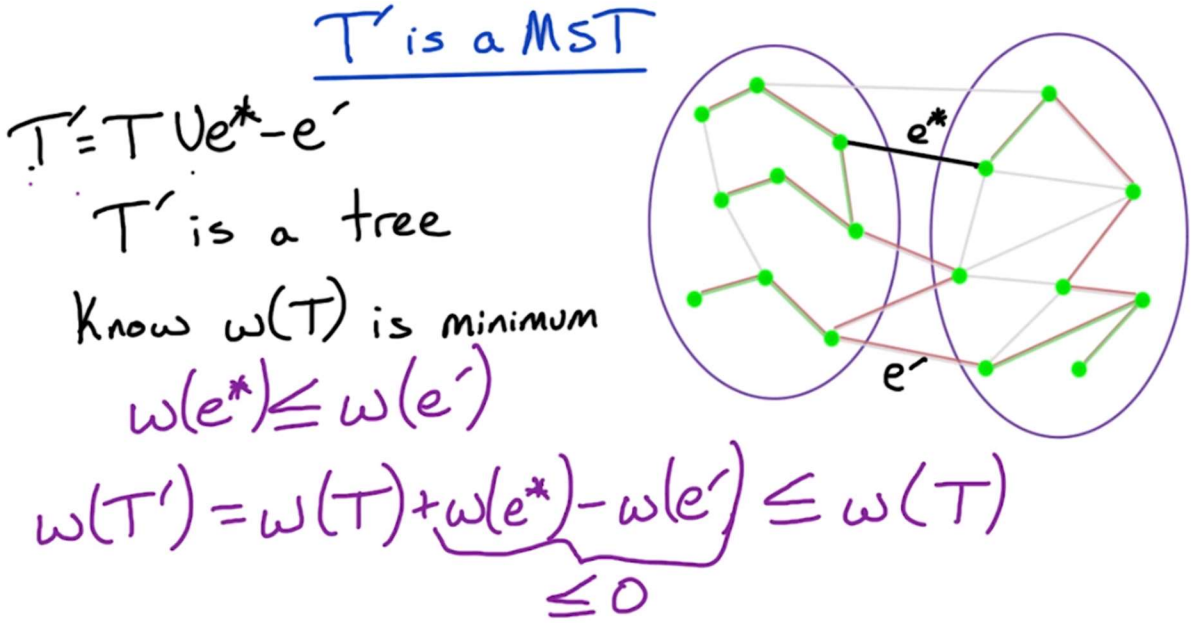
So we're going to go along this path and now the original path uses this edge e′ but it doesn't exist now so we use the rest of the cycle. It gets over to d and then from d we follow the rest of the path P. So, actually, in this example we're backtracking.

So this doesn't actually give us a path, it gives us a walk. But it shows that y and z are connected in T′ and then, of course, we can truncate this walk to make a path. But our whole point is to show that y and z are connected and that's all we need to show.

We've shown that's the case for any y and z; therefore, T′ is connected, and since it has exactly n-1 edges, then T′ is in fact a tree.

***

(Slide 14) T′ is a MST



Now, once again, we have this subgraph T′ which is obtained by taking the tree T, adding in this edge e*, and taking out the edge e′. We've just showed that T′ is a tree and, once again, notice that we could have removed any edge of this cycle and we would have attained a tree. But now we want to show that it's a minimum spanning tree.

How do we show it's of minimum weight? - well, you know that T is an MST, so we know that the weight of T is a minimum over all trees. So to show that T′ is a minimum spanning tree, we have to show that the weight of T′ equals the weight of T.

What do we know about the weight of e* compared to that of e′? - well, we know that the weight of e* is minimum over all edges cross the cut. e′ also crosses the cut(S, $\bar{S}$). Therefore, the weight of e* is at most that of the weight of e′.

Now what is the weight of T′? - it's the weight of T plus the weight of e* and then finally minus the weight of e′. The weight of e* is at most the weight of e′ so this part is almost zero. So this whole thing is at most the weight of T.

So, we've shown that the weight of T′ is at most of the weight of T - T is minimum weight - therefore, T′ is also a minimum weight and in fact, the weight of T′ must equal the weight of T.

So the weight of e* must equal the weight of e′. Otherwise, we retain the new tree T′ which is

strictly smaller weight than T, which would contradict the fact that T is an MST. So, all these edges of T which cross this cut, must be of exactly the same weight as e*; otherwise, we can obtain a new tree which is of smaller weight.

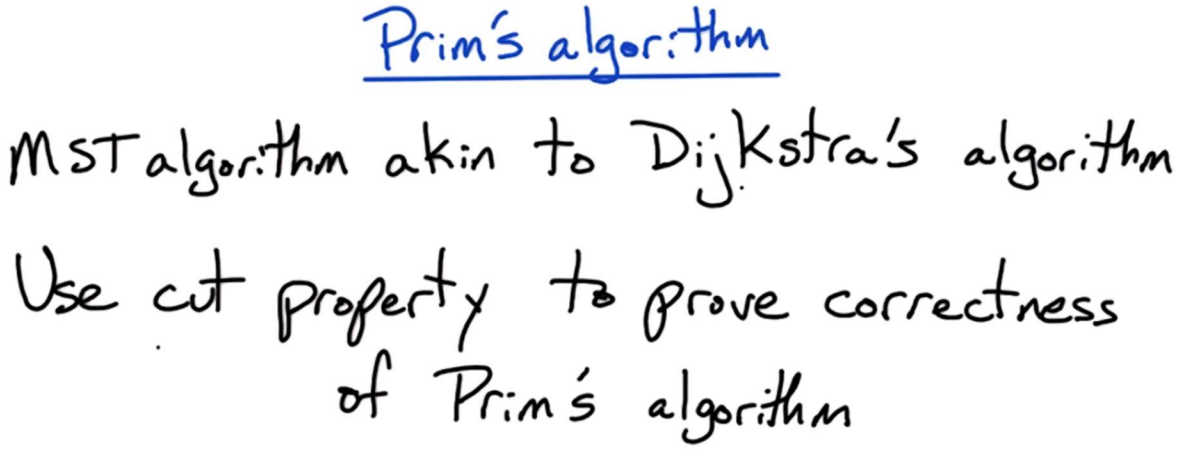
That completes the proof of the cut property.

Now, the key idea, that I want to stress, is that I can take a tree T and add an edge into that tree - so I take T U {e*} - that creates a cycle. Now I can remove any edge of that cycle and I get a new tree T'. Now that's the idea that I wanted to get from the proof of the cut property.

The other idea I want you to get, is the idea from the statement of the cut property. I want you to get the idea that a minimum weight edge across the cut is part of a MST. Those are the two key ideas I want you to get from the statement of the cut property and the proof idea of the cut property.

***

(Slide 15) Prim's Algorithm



Now, one last point about the cut property. There's another algorithm, Prim's algorithm. Prim's algorithm is for the Minimum Spanning Tree problem. And is very similar to Dijkstra's algorithm for the shortest path.

To make sure you understand the cut property, you should use it to prove correctness of Prim's algorithm. Make sure you understand why Prim's algorithm is correct, by using the cut property.