6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 16

-- valuable experience. OK, today we're going to start talking about a particular class of algorithms called greedy algorithms. But we're going to do it in the context of graphs. So, I want to review a little bit about graphs, which mostly you can find in the textbook in appendix B. And so, if you haven't reviewed in appendix B recently, please sit down and review appendix B. It will pay off especially during our take-home quiz. So, just reminder, a digraph, what's a digraph? What's that short for? Directed graph, OK? Directed graph, G equals (V,E), OK, has a set, V, of vertices.

And, I always get people telling me that I have one vertice. The singular is not vertice; it is vertex, OK? The plural is vertices. The singular is vertex. It's one of those weird English words. It's probably originally like French or something, right? I don't know. OK, anyway, and we have a set, E, which is a subset of V cross V of edges. So that's a digraph. And undirected graph, E contains unordered pairs.

OK, and, sorry? It's Latin, OK, so it's probably pretty old, then, in English. I guess the vertex would be a little bit of a giveaway that maybe it wasn't French. It started to be used in 1570, OK. OK, good, OK, so the number of edges is, whether it's directed or undirected, is O of what? V^2, good. OK, and one of the conventions that will have when we're dealing, once we get into graphs, we deal a lot with sets. We generally drop the vertical bar notation within O's just because it's applied. It just makes it messier. So, once again, another abuse of notation. It really should be order the size of V^2, but it just messes up, I mean, it's just more stuff to write down. And, you're multiplying these things, and all those vertical bars.

Since they don't even have a sense to the vertical bar, it gets messy. So, we just drop the vertical bars there when it's in asymptotic notation. So, E is order V^2 when it's a set of pairs, because if it's a set of pairs, it's at most n choose two, which is where it's at most n^2 over 2, here it could be, at most, sorry, V^2 over 2, here it's at most V^2. And then, another property that sometimes comes up is if the G is connected, we have another bound, implies that the size of E is at least the size of V minus one. OK, so if it's connected, meaning, what does it mean to have a graph that's connected?

Yeah, there's a path from any vertex to any other vertex in the graph. That's what it means to be connected. So if that's the case, that a number of edges is at least the number of vertices minus one, OK? And so, what that says, so one of the things we'll get into, a fact that I just wanted to remind you, is that in that case, if I look at log E, OK, log of the number of edges, that is O of log V.

And by this, is omega of log V. So, it's equal to theta of log V. OK, so basically the number of, in the case of a connected graph, the number of edges, and the number of vertices are polynomially related. So, their logs are comparable. OK, so that's helpful just to know because sometimes I just get questions later on where people will say, oh, you showed it was log E but you didn't show it was log V. And I could

point out that it's the same thing. OK, so there's various ways of representing graphs in computers, and I'm just going to cover a couple of the important ones.

There's actually more. We'll see some more. So, the simplest one is what's called an adjacency matrix. An adjacency matrix of the graph, G, equals (V,E), where, for simplicity, I'll let V be the set of integers from one up to n, OK, is the n by n matrix A given by the ij-th at the entry is simply one if the edge, ij, is in the edge set and zero if ij is not in the edge set. OK, so it's simply the matrix where you say, the ij entry is one if it's in the matrix. So, this is, in some sense, giving you the predicate for, is there an edge from i to j?

OK, remember, predicate is Boolean formula that is either zero or one, and in this case, you're saying it's one if there is an edge from i to j and zero otherwise. OK, sometimes you have edge weighted graphs, and then sometimes what people will do is replace this by edge weights. OK, it will be the weight of the edge from i to j. So, let's just do an example of that just to make sure that our intuition corresponds to our mathematical definitions. So, here's an example graph. Let's say that's our graph.

So let's just draw the adjacency the matrix. OK, so what this says: is there's an edge from one to one? And the answer is no. Is there an edge from one to two? Yes. Is there an edge from one to three here? Yep. Is there an edge for one to four? No. Is there an edge from two until one? No. Two to two? No. Two to three? Yes. Two to four? No. No edges going out of three. Edge from four to three, and that's it. That's the adjacency matrix for this particular graph, OK? And so, I can represent a graph as this adjacency matrix.

OK, when I represent it in this way, how much storage do I need? OK, n^2 or V^2 because the size is the same thing for V^2 storage, OK, and that's what we call a dense representation. OK, it works well when the graph is dense. So, the graph is dense if the number of edges is close to all of the edges possible. OK, then this is a good representation. But for many types of graphs, the number of edges is much less than the possible number of edges, in which case we say the graph is sparse. Can somebody give me an example of a sparse graph?

A class of graphs: so, I want a class of graphs that as n grows, the number of edges in the graph doesn't grow as the square, but grows rather as something much smaller. A linked list, so, a chain, OK, if you look at it from a graph theoretically, is a perfectly good example: only n edges in the chain for a chain of length n. So therefore, the number of edges would be order V. And in particular, you'd only have one edge per row here. What other graphs are sparse? Yeah? Good, a planar graph, a graph that can be drawn in a plane turns out that if it has V vertices has, and V is at least three, then it has, at most, three V minus six edges.

So, it turns out that's order V edges again. What's another example of a common graph? Yeah, binary tree, or even actually any tree, you know, what's called a free tree if you read the appendix, OK, a tree that just is a connected graph that has no cycles, OK, is another example. What's an example of a graph that's dense? A complete graph, OK: it's all ones, OK, or if you have edge weights, it would be a completely filled in matrix. OK, good. So, this is good for dense representation. But sometimes you want to have a sparse representation so we don't have to spend V^2 space to deal with all of the, where most of it's going to be zeroes. OK, it's sort of like, if we know why it's zero, why bother representing it as zero?

So, one such representation is an adjacency list representation. Actually, adjacency list of a given vertex is the list, which we denote by Adj of V, of vertices adjacent to V. OK, just in terms by their terminology, vertices are adjacent, but edges are incident on vertices. OK, so the incidence is a relation between a vertex and an edge. An adjacency is a relation between two vertices. OK, that's just the language.

Why they use to different terms, I don't know, but that's what they do. So, in the graph, for example, the adjacency list for vertex one is just the list or the set of two three because one has going out of one are edges to two and three. The adjacency list for two is just three, four, three. It's the empty set, and for four, it is three. OK, so that's the representation. Now, if we want to figure out how much storage is required for this representation, OK, we need to understand how long the adjacency list is.

So, what is the length of an adjacency list of a vertex, V? What name do we give to that? It's the degree. So, in an undirected graph, we call it the degree of the vertex. This is undirected. OK, about here, OK. So that's an undirected case. In the directed case, OK, actually I guess the way we should do this is say this. If the degree, we call it the out degree for a digraph. OK, so in a digraph, we have an out degree and an in degree for each vertex.

So here, the in degree is three. Here, the out degree is two, OK? So, one of the important lemma that comes up is what's called the handshaking lemma. OK, it's one of these mathematical lemmas. And so, it comes from a story. Go to a dinner party, and everybody at the dinner party shakes other people's hands. Some people may not shake anybody's hand. Some people may shake several people's hands. Nobody shakes hands with themselves.

And at some point during the dinner party, the host goes around and counts up how many, the sum, of the number of hands that each person has shaken. OK, so he says, how many did you shake? How many did you shake? How many did you shake? He adds them up, OK, and that number is guaranteed to be even. OK, that's the handshaking lemma. Or, stated a little bit more precisely, if I take for any graph the degree of the vertex, and sum them all up, that's how many hands everybody shook, OK, that's actually equal to always twice the number of edges. So, why is that going to be true? Why is that going to be twice the number of edges?

Yeah? Yeah. Every time you put in an edge, you add one to the degree of each person on each end. So, it's just two different ways of counting up the same number of edges. OK, I can go around, and if you imagine that, that every time I count the degree of the node, I put a mark on every edge. Then, when I'm done, every edge has two marks on it, one for each end. OK: a pretty simple theorem. So, what that says is that for undirected graphs, that implies that the adjacency list representation, uses how much storage?

OK, at most, 2E, so order E because that's not all. Yeah, so you have to have the number of vertices plus order the number of edges, OK, whether it's directed or undirected because I may have a graph, say it has a whole bunch of vertices and no edges, that's still going to cost me order V, OK? So, it uses theta of V plus E storage. And, it's basically the same thing asymptotically. In fact, it's easier to see in some sense for digraphs because for digraphs, what I do is I just add up the out degrees, and that equal to E, OK, if I add up the out degrees as equally.

In fact, this is kind of like it amortized analysis, if you will, a book keeping analysis, that if I'm adding up the total number of edges, one way of doing it is accounting for a vertex by vertex. OK, so for each vertex, I basically can take each degree, and basically each vertex, look at the degree, and that allocating of account per edge, and then ending up with twice the number of edges, that's exactly accounting type of analysis that we might do for amortized analysis.

OK, so we'll see that. So, this is a sparse representation, and it's often better than an adjacency matrix. For example, you can imagine if the World Wide Web were done with an adjacency matrix as opposed to, essentially, with an adjacency list type of representation. Every link on the World Wide Web, I had to say, here are the ones that I'm connected to, and here are all the ones I'm not connected to. OK, that list of things you're not connected to for a given page would be pretty dramatically, show you that there is an advantage to sparse representation.

On the other hand, one of the nice things about an adjacency matrix representation is that each edge can be represented with a single bit, whereas typical when I'm representing things with an adjacency list representation, how many bits am I going to need to represent each adjacency? You'll need order log of V to be able to name each different vertex. OK, the log of the number is the number of bits that I need. So, there are places where this is actually a far more efficient representation. In particular, if you have a very dense graph, OK, this may be a better way of representing it. OK, the other thing I want you to get, and we're going to see more of this in particular next week, is that a matrix and a graph, there are two ways of looking at the same thing.

OK, and in fact, there's a lot of graph theory that when you do things like multiply the adjacency matrix, OK, and so forth. So, there's a lot of commonality between graphs and matrices, a lot of mathematics that if it applies for one, it applies to the other. Do you have a question, or just holding your finger in the air? OK, good. OK, so that's all just review. Now I want to get onto today's lecture. OK, so any questions about graphs? So, this is a good time to review appendix B. there are a lot of great properties in there, and in particular, there is a theorem that we're going to cover today that we're going to talk about today, which is properties of trees. Trees are very special kinds of graphs, so I really want you to go and look to see what the properties are.

There is, I think, something like six different definitions of trees that are all equivalent, OK, and so, I think a very good idea to go through and read through that theorem. We're not going to prove it in class, but really, provides a very good basis for the thinking that we're going to be doing today. And we'll see more of that in the future. OK, so today, we're going to talk about minimum spanning trees.

OK, this is one of the world's most important algorithms. OK, it is important in distributed systems. It's one of the first things that almost any distributed system tries to find is a minimum spanning tree of the nodes that happened to be alive at any point, OK? And one of the people who developed an algorithm for this, we'll talk about this a little bit later, OK, it was the basis of the billing system for AT&T for many years while it was a monopoly.

OK, so very important kind of thing. It's got a huge number of applications. So the problem is the following. You have a connected undirected graph, G equals (V,E),

with an edge weight function, w, which maps the edges into weights that are real numbers. And for today's lecture, we're going to make an important assumption, OK, for simplicity. The book does not make this assumption. And so, I encourage you to look at the alternative presentation or, because what they do in the book is much more general, but for simplicity and intuition, I'm going to make this a little bit easier. We're going to assume that all edge weights are distinct.

OK, all edge weights are distinct. So what does that mean? What does that mean that this function, w, what property does the function, w, have if all edge weights are distinct? Who remembers their discreet math? It's injective. OK, it's one to one. OK, it's not one to one and onto necessarily. In fact, it would be kind of hard to do that because that's a pretty big set. OK, but it's one to one. It's injective. OK, so that's what we're going to assume for simplicity. OK, and the book, they don't assume that. It just means that the way you have to state things is just a little more precise. It has to be more technically precise. So, that's the input. The output is--

The output is a spanning tree, T, and by spanning tree, we mean it connects all the vertices. OK, and it's got to have minimum weight. OK, so we can write the weight of the tree is going to be, by that, we meet the sum over all edges that are in the tree of the weight of the individual edges. OK, so here I'(V,E) done a little bit of abusive notation, which is that what I should be writing is w of the edge (u,v) because this is a mapping from edges, which would give me a double parentheses.

And, you know, as you know, I love to abuse notation. So, I'm going to drop that extra parentheses, because we understand that it's really the weight of the edge, OK, not the weight of the ordered pair. So, that's just a little notational convenience. OK, so one of the things, when we do the take-home exam, notational convenience can make the difference between having a horrible time writing up a problem, and an easy time. So, it's worth thinking about what kinds of notation you'll use in writing up solutions to problems, and so forth. OK, and just in general, a technical communication, you adopt good notation people understand you. You adopt a poor notation: nobody pays attention to what you're doing because they don't understand what you're saying.

OK, so let's do an example. OK, so here's a graph. I think for this, somebody asked once if I was inspired by biochemistry or something, OK, but I wasn't. I was just writing these things down, OK? So, here's a graph. And let's give us some edge weights. OK, so there are some edge weights. And now, what we want is we want to find a tree. So a connected acyclic graph such that every vertex is part of the tree. But it's got to have the minimum weight possible. OK, so can somebody suggest to me some edges that have to be in this minimum spanning tree?

Yeah, so nine, good. Nine has to be in there because, why? It's the only one connecting it to this vertex, OK? And likewise, 15 has to be in there. So those both have to be in. What other edges have to be in? Which one? 14 has to be it. Why does 14 have to be in? Well, one of 14 and three has to be in there. I want the minimum weight. The one that has the overall smallest weight. So, can somebody argue to me that three has to be in there?

Yeah? That's the minimum of two, which means that if I had a, if you add something you said was a minimum spanning tree that didn't include three, right, and so therefore it had to include 14, then I could just delete this edge, 14, and put in edge three. And, I have something of lower weight, right? So, three has to be in there.

What other edges have to be in there? Do a little puzzle logic. Six and five have to be in there. Why do they have to be in there?

Yeah, well, I mean, it could be connected through this or something. It doesn't necessarily have to go this way. Six definitely has to be in there for the same reason that three had to be, right? Because we got two choices to connect up this guy. And so, if everything were connected but it weren't, 12, I mean, and 12 was in there. I could always, then, say, well, let's connect them up this way instead.

OK, so definitely that's in there. I still don't have everything connected up. What else has to be in there for minimum spanning tree? Seven, five, and eight, why seven, five, and eight? OK, so can we argue those one at a time? Why does five have to be in there? Yeah? OK, so we have four connected components because we have this one, this one, we actually have, yeah, this one here, and this one, good. We need at least three edges to connect them because each edge is going to reduce the connected components by one. OK, so we need three edges, and those are the three cheapest ones. And they work.

That works, right? Any other edges are going to be bigger, so that works. Good. OK, and so, now do we have a spanning tree? Everything is, we have one big connected graph here, right? Is that what I got? Hey, that's the same as what I got. Life is predictable. OK, so, so everybody had the idea of what a minimum spanning tree is, then, out of this, OK, what's going on there? So, let's first of all make some observations about this puzzle. And what I want to do is remind you about the optimal substructure property because it turns out minimum spanning tree has a great optimal substructure property.

OK, so the setup is going to be, we're going to have some minimum spanning tree. Let's call it T. And, I'm going to show that with the other edges in the graph, are not going to be shown. OK, so here's a graph. OK, so here's a graph. It looks like the one I have my piece of paper here. OK, so the idea is, this is some minimum spanning tree. Now, we want to look at a property of optimal substructure. And the way I'm going to get that, is, I'm going to remove some edge, (u,v), move an arbitrary edge, (u,v), in the minimum spanning tree. So, let's call this u and this V. And so, we're removing this edge.

OK, so when I remove an edge in a tree, what happens to the tree? What's left? I have two trees left, OK? I have two trees left. Now, proving that, that's basically one of the properties in that appendix, and the properties of trees that I want you to read, OK, because you can actually prove that kind of thing rather than it just being obvious, which is, OK? OK, so we remove that. Then, T is partitioned into two subtrees. And, we'll call them T_1 and T_2. So, here's one subtree, and here's another subtree. We'(V,E) partitioned it. No matter what edge I picked, there would be two subtrees that it's partitioned into.

Even if the sub tree is a trivial subtree, for example, it just has a single node in it and no edges. So, the theorem that we'll prove demonstrates a property of optimal substructure. T_1 is a minimum spanning tree for the graph, G_1, E_1, a subgraph of G induced by the vertices in T_1. OK, that is, V_1 is just the vertices in T_1 is what it means to be induced. OK, so V_1 is the vertices in T_1. So, in this picture, I didn't label it. This is T_1. This is T_2. In this picture, these are the vertices of T_1. So, that's V_1, OK? And, E_1 is the set of pairs of vertices, x and y, that are the edges that are in E_1 such that both x and y belong to V_1.

OK, so I haven't shown the edges of G here. But basically, if an edge went from here to here, that would be in the E_1. If it went from here to here, it would not. And if it went from here to here, it would not. OK, so the vertices, the subgraph induced by the vertices of T_1 are just those that connect up things in T_1, and similarly for T_2. So, the theorem says that if I look at just the edges within the graph here, G_1, those that are induced by these vertices, T_1 is, in fact, a minimum spanning tree for that subgraph. That's what the theorem says. OK, if I look over here conversely, or correspondingly, if I look at the set of edges that are induced by this set of vertices, the vertices in T_2, in fact, T_2 is a minimum spanning tree on that subgraph.

OK, OK, we can even do it over here. If I took a look, for example, at these, let's see, let's say we cut out five, and if I cut out edge five, that T_1 would be these four vertices here. And, the point is that if I look at the subgraph induced on that, that these edges here. In fact, the six, eight, and three are all edges in a minimum spanning tree for that subgraph. OK, so that's what the theorem says. So let's prove it.

OK, and so what technique are we going to use to prove it? OK, we learned this technique last time: hint, hint. It's something you do it in your text editor all the time: cut and paste, good, cut and paste. OK, so the weight of T I can express as the weight of the edge I removed, plus the weight of T_1, plus the weight of T_2. OK, so that's the total weight. So, the argument is pretty simple. Suppose that there were some T_1 prime that was better than T_1 for G_1. Suppose I had some better way of forming a spanning tree.

OK, then I would make up a T prime, which just contained the edges, (u,v), and T_1 prime, union T_2. So, I would take, if I had a better spanning tree, a spanning tree of lower weight for T_1. And I call that T_1 prime. I just substitute that and make up a spanning tree that consisted of my edge, (u,v), whatever works well for T_1 prime and whatever works well for T. And, that would be a spanning tree.

And it would be better than T itself was for G, OK, because the weight of these is just as the weight for this, I now just get to use the weight of T_1 prime, and that's less. And so, therefore, the assumption that T was a minimum spanning tree would be violated if I could find a better one for the subpiece. So, we have this nice property of optimal substructure. OK, I have subproblems that exhibit optimal, if I have a globally optimal solution to the whole problem within it, I can find optimal solutions to subproblems.

So, now the question is, that's one hallmark. That's one hallmark of dynamic programming. What about overlapping subproblems? Do I have that property? Do I have overlapping subproblems over here for this type of problem? So, imagine, for example, that I'm removing different edges. I look at the space of taking a given edge, and removing it. It partitions it into two pieces, and now I have another piece. And I remove it, etc.

Am I going to end up getting a bunch of subproblems that are similar in there? Yeah, I am. OK, if I take out this one, then I take out, say, this one here, and then I'll have another tree here and here. OK, that would be the same as if I had originally taken this out, and then taken that one out. If I look at simple ordering of taking out the edges, I'm going to end up with a whole bunch of overlapping subproblems.

Yeah, OK. So then, what does that suggest we use as an approach? Dynamic programming, good. What a surprise! Yes, OK, you could use dynamic programming. But it turns out that minimum spanning tree exhibits an even more powerful property. OK, so we'(V,E) got all the clues for dynamic programming, but it turns out that there's an even bigger clue that's going to help us to use an even more powerful technique.

And that, we call, the hallmark for greedy algorithms. And that is, we have a thing called the greedy choice property, which says that a locally optimal choice is globally optimal. And, of course, as all these hallmarks is the kind of thing you want to box, OK, because these are the clues that you're going to be able to do that. So, we have this property that we call the greedy choice property. I'm going to show you how it works in this case. And when you have a greedy choice property, it turns out you can do even better that dynamic programming.

OK, so when you see the two dynamic programming properties, there is a clue that says dynamic programming, yes, but also it says, let me see whether it also has this greedy property because if it does, you're going to come up with something that's even better than dynamic programming, OK? So, if you just have the two, you can usually do dynamic programming, but if you have this third one, it's like, whoa! Jackpot!

OK, so here's the theorem we'll prove to illustrate this idea. Once again, these are not, all these hallmarks are not things. They are heuristics. I can't give you an algorithm to say, here's where dynamic programming works, or here's where greedy algorithms work. But I can sort of indicate when they work, the kind of structure they have. OK, so here's the theorem. So let's let T be the MST of our graph. And, let's let A be any subset of V, so, some subset of vertices.

And now, let's suppose that edge, (u,v), is the least weight edge connecting our set A to A complement, that is, V minus A. Then the theorem says that (u,v) is in the minimum spanning tree. So let's just take a look at our graph over here and see if that's, in fact, the case. OK, so let's take, so one thing I could do for A is just take a singleton node. So, I take a singleton node, let's say this guy here, that can be my A, and everything else is V minus A.

And I look at the least weight edge connecting this to everything else. Well, there are only two edges that connect it to everything else. And the theorem says that the lighter one is in the minimum spanning tree. Hey, I win. OK, if you take a look, every vertex that I pick, the latest edge coming out of that vertex is in the minimum spanning tree. OK, the lightest weight edge coming out, but that's not all the edges that are in here.

OK, or let's just imagine, let's take a look at these three vertices connected to this set of vertices. I have three edges is going across. The least weight one is five. That's the minimum spanning tree. Or, I can cut it this way. OK, the ones above one, the edges going down are seven, eight, and 14. Seven is the least weight. It's in the minimum spanning tree. So, no matter how I choose, I could make this one in, this one out, this one in, this one out, this one in, this one out, this one in, this one out, take a look at what all the edges are. Which ever one to the least weight: it's in the minimum spanning tree. So, in some sense, that's a local property because I don't have to look at what the rest of the tree is.

I'm just looking at some small set of vertices if I wish, and I say, well, if I wanted to connect that set of vertices to the rest of the world, what would I pick? I'd pick the cheapest one. That's the greedy approach. It turns out, that wins, OK, that picking that thing that's locally good for that subset, A, OK, is also globally good. OK, it optimizes the overall function. That's what the theorem says, OK? So, let's prove this theorem. Any questions about this? OK, let's prove this theorem. So, we have (u,v) is the least weight edge connecting A to D minus A. So, let's suppose that this edge, (u,v), is not in the minimum spanning tree.

OK, let's suppose that somehow there is a minimum spanning tree that doesn't include this least weight edge. OK, so what technique you think will use to prove to get a contradiction here? Cut and paste, good. Yeah, we're going to cut paste. OK, we're going to cut and paste. So here, I did an example. OK, so -- OK, and so I'm going to use the notation. I'm going to color some of these in.

OK, and so my notation here is this is an element of A, and color it in. It's an element of V minus A. OK, so if it's not colored it, that's an A. This is my minimum spanning tree. Once again, I'm not showing the overall edges of all the graphs, but they're there, OK? So, my edge, (u,v), which is not my minimum spanning tree I say, let's say is this edge here. It's an edge from u, u as in A, v as in V minus A. OK, so everybody see the setup? So, I want to prove that this edge should have been in the minimum spanning tree, OK, that the contention that this is a minimum spanning tree, and does include (u,v) is wrong.

So, what I want to do, that, is I have a tree here, T, and I have two vertices, u and v, and in a tree, between any two vertices there is a unique, simple path: simple path meaning it doesn't go back and forth and repeat edges or vertices. OK, there's a unique, simple path from u to v. So, let's consider that path. OK, and the way that I know that that path exists is because I'(V,E) read appendix B of the textbook, section B.5.1, OK, which has this nice theorem about properties of trees. OK, so that's how I know that there exists a unique, simple path.

OK, so now we're going to do is take a look at that path. So in this case, it goes from here, to here, to here, to here. And along that path, there must be a point where I connect from a vertex in A to a vertex in V minus A. Why? Well, because this is in A. This is in V minus A. So, along the path somewhere, there must be a transition. OK, they are not all in A, OK, because in particular, V isn't. OK, so we're going to do is swap (u,v) with the first edge on this path that connects a vertex in A to a vertex in V minus A.

So in this case, it's this edge here. I go from A to V minus A. In general, I might be alternating many times, OK, and I just picked the first one that I encounter. OK, that this guy here. And what I do is I put this edge in. OK, so then, what happens? Well, the edge, (u,v), is the lightest thing connecting something in A to something in V minus A. So that means, in particular, it's lighter than this edge, has lower weight. So, by swapping this, I'(V,E) created a tree with lower overall weight, contradicting the assumption that this other thing was a minimum spanning tree.

OK: so, a lower weight spanning tree than T results, and that's a contradiction -- -- than T results. And that's a contradiction, OK? How are we doing? Everybody with me? OK, now we get to do some algorithms. Yea! So, we are going to do an algorithm called Prim's algorithm. Prim eventually became a very high-up at AT&T

because he invented this algorithm for minimum spanning trees, and it was used in all of the billing code for AT&T for many years. He was very high up at Bell Labs back in the heyday of Bell Laboratories. OK, so it just shows, all you have to do is invent an algorithm.

You too can be a president of a corporate monopoly. Of course, the government can do things to monopolies, but anyway, if that's your mission in life, invent an algorithm. OK, so here's the idea. What we're going to do is we're going to maintain V minus A as a priority queue. We'll call it Q. And each vertex, we're going to key each vertex in Q with the weight of the least weight edge, connecting it to a vertex in A.

So here's the code. So, we're going to start out with Q being all vertices. So, we start out with A being, if you will, the empty set. OK, and what we're going to do it is the least weight edge, therefore, for everything in the priority queue is basically going to be infinity because none of them have any edges. The least weight edge to the empty set is going to be empty. And then, we're going to start out with one guy.

We'll call him S, which will set to zero for some arbitrary S in V. And then, the main part of the algorithm kicks in. So that's our initialization. OK, when we do the analysis, I'm going to write some stuff on the left hand side of the board. So if you're taking notes, you may want to also leave a little bit of space on the left hand side of your notes. So, while Q is not empty, we get the smallest element out of it.

And then we do some stuff. That's it. And the only thing I should mention here is, OK, so let's just see what's going on here. And then we'll run it on the example. OK, so what we do is we take out the smallest element out of the queue at each step. And then for each step in the adjacency list, in other words, everything for which I have an edge going from v to u, we take a look, and if v is still in our set V minus A, so things we'(V,E) taken out are going to be part of A. OK, every time we take something out, that's going to be a new A that we construct.

At every step, we want to find, what's the cheapest edge connecting that A to everything else? We basically are going to take whatever that cheapest thing is, OK, add that edge in, and now bring that into A and find the next cheapest one. And we just keep repeating the process. OK, we'll do it on the example. And what we do, is every time we bring it in, I keep track of, what was the vertex responsible for bringing me in.

And what I claim is that at the end, if I look at the set of these pairs that I'(V,E) made here, V and pi of V, that forms the minimum spanning tree. So let's just do this. And, what's that? We're all set up. So let's get rid of these guys here because we are going to recompute them from scratch. OK, so you may want to copy the graph over again in your notes. I was going to do it, but it turned out, this is exactly the board is going to erase this. OK, well let me just modify it. OK, so we start out. We make everything be infinity. OK, so that's where I'm going to keep the key value. OK, and then what I'm going to do is find one vertex.

And I'm going to call him S. And I'm going to do this vertex here. We'll call that S. So basically, I now make him be zero. And now, what I do, is I execute extract min. So basically, what I'll do is I'll just shade him like this, indicating that he has now joined the set A. So, this is going to be A. And this is element of V minus A. OK, so then what we do is we take a look. We extract him, and then for each edge in the

adjacency list, OK, so for each vertex in the adjacency lists, that these guys here, OK, we're going to look to see if it's still in Q, that is, in V minus A.

And if so, and its key value is less than what the value is at the edge, there, we're going to replace it by the edge value. So, in this case, we're going to replace this by seven. We're going to replace this by 15, and we're going to replace this by ten, OK, because what we're interested in is, what is the cheapest? Now, notice that everything in V minus A, that is, what's in the priority queue, everything in there, OK, now has its cheapest way of connecting it to the things that I'(V,E) already removed, the things that are in A. OK, and so now I just, OK, when I actually do that update, there's actually something implicit going on in this priority queue.

And that is that I have to do a decreased key. So, there's an implicit decrease of the key. So, decreased key is a priority queue operation that lowers the value of the key in the priority queue. And so, that's implicitly going on when I look at what data structure I'm going to use to implement that priority queue. OK, so common data structures for implementing a priority queue are a min heap.

OK, so I have to make sure that I'm actually doing this operation. I can't just change it and not affect my heap. So, there is an implicit operation going on there. OK, now I repeat. I find the cheapest thing, oh, and I also have to set, now, a pointer from each of these guys back to u. So here, this guy sets a pointer going this way. This guy sets a pointer going this way, and this guy sets a pointer going this way. That's my pi thing that's going to keep track of who caused me to set my value to what it is. So now, we go in and we find the cheapest thing, again. And we're going to do it fast, too. OK, this is a fast algorithm.

OK, so now we're going to go do this again. So now, what's the cheapest thing to extract? This guy here, right? So, we'll take him out, OK, and now we update all of his neighbors. So this guy gets five. This guy gets 12. This guy gets nine. This guy we don't update. We don't update him because he's no longer in the priority queue. And all of these guys now, we make point to where they're supposed to point to. And, we're done with that step. Now we find the cheapest one. What's the cheapest one now? The five over here. Good. So, we take him out. OK, we update the neighbors. Here, yep, that goes to six now. And, we have that pointer.

And, this guy we don't do, because he's not in there. This guy becomes 14, and this guy here becomes eight. So, we update that guy, make him be eight. Did I do this the right way? Yeah, because pi is a function of this guy. So basically, this thing, then, disappears. Yeah, did I have another one that I missed? 12, yes, good, it's removed, OK, because pi is just a function. And now I'm OK. OK, so now what do I do? OK, so now my set, A, consists of these three things, and now I want the cheapest edge. I know it's in the minimum spanning tree. So let me just greedily pick it.

OK, so what's the cheapest thing now? This guy appear? Yeah, six. So we take it. We go to update these things, and nothing matters here. OK, nothing changes because these guys are already in A. OK, so now the cheapest one is eight here. Good. So, we take eight out. OK, we update this. Nothing to be done. This: nothing to be done. This: oh, no, this one, instead of 14 we can make this be three. So, we get rid of that pointer and make it point that way. Now three is the cheapest thing. So, we take it out, and of course there's nothing to be done over there. And now, last, I take nine. And it's done. And 15: it's done. And the algorithm terminates.

OK, and as I look at, now, all the edges that I picked, those are exactly all the edges that we had at the beginning. OK, let's do an analysis here. OK, so let's see, this part here costs me order V, right? OK, and this part, let's see what we are doing here. Well, we're going to go through this loop how many times? V times. It's V elements we put into the queue. We are not inserting anything. We're just taking them out. This goes V times, OK, and we do a certain number of extract Mins. So, we're going to do order V extract Mins. And then we go to the adjacency list, and we have some constant things. But we have these implicit decreased keys for this stuff here.

That's this thing here. OK, and so how many implicit decreased keys do we have? That's going to be the expensive thing. OK, we have, in this case, the degree of u of those. OK, so overall, how many implicit decreased keys do we have? Well, we have V times through. How big could the degree of u be? OK, it could be as big as V, order V. So, that's V^2 decreased use. But we can do a better bound than that.

How many do we really have? Yeah, at most order E, OK, because what am I doing? I'm summing up the degrees of all the vertices. That's how many times I actually execute that. So, I have order E, implicit decreased keys. So the time overall is order V times time for whatever the extract Min is plus E times the time for decreased key. So now, let's look at data structures, and we can evaluate for different data structures what this formula gives us.

So, we have different ways of implementing a data structure. We have the cost of extract Min, and of decreased key, and total. So, the simplest way of implementing a data structure is an unsorted array. If I have an unsorted array, how much time does it take me to extract the minimum element? If I have an unsorted array? Right, order V in this case because it's an array of size V. And, to do a decreased key, OK, I can do it in order one.

So, the total is V^2, good, order V^2 algorithm. Or, as people suggested, how about a binary heap? OK, to do an extract Min in a binary heap will cost me what? O of log V. Decreased key will cost me, yeah, it turns out you can do that in order log V because basically you just have to shuffle the value, actually shuffle it up towards the root, OK? Or at log V. And, the total cost therefore is?

E log V, good. Which of these is better? It depends, good. When is one better, and when is the other better? Yeah, if it's a dense graph, E is close to V^2, the array is better. But if it's a sparse graph, and E is much smaller than V^2, then the binary heap is better. So that motivated the invention of a data structure, OK, called a Fibonacci Heap. So, Fibonacci Heap is covered in Chapter 20 of CLRS. We're not going to hold you responsible for the content, but it's an interesting data structure because it's an amortized data structure. And it turns out that it is data structure, you can do extract Min in order log V amortized time.

And remarkably, you can do decreased key in order one amortized. So, when I plug those in, what do I get over here? What's that going to be? Plug that it here. It's going to be V times log V plus E: E plus V log V. These are amortized, so what's this? Trick question. It's worst-case. It's not amortized over here. These are amortized, but that's the beauty of amortization. I can say it's going to be worst case: E plus V log V over here, because when I add up the amortized cost of my operations, it's an upper bound on the true costs. OK, so that's why I say, one of the beauties of this

amortized analysis, and in particular, being able to assign different costs to different operations is I can just add them up and I get my worst-case costs.

So this is already V log V. There are a couple other algorithms just before I let you go. Kruskal's Algorithm in the book uses another amortized data structure called a disjoint set data structure, which also runs in E log V, that is, this time: runs in this time, the same as using a binary heap. So, I'll refer you to the book. The best algorithm to date with this problem is done by our own David Karger on the faculty here with one of our former graduates, Phil Kline, who is now a professor at Brown, and Robert Tarjan, who is sort of like the master of all data structures who was a professor at Princeton in 1993. OK, it's a randomized algorithm, and it gives you order V plus E expected time. OK, so that's the best to date.

It's still open as to whether there is a deterministic, there is worst-case bound, whether there is a worst-case bound that is linear time. OK, but there is a randomized to linear time, and otherwise, this is essentially the best bound without additional assumptions. OK, very cool stuff. Next, we're going to see a lot of these ideas of greedy and dynamic programming in practice.