[SQUEAKING]

[RUSTLING]

[CLICKING]

JASON KU: OK, welcome back to 6.006. We're going to have a-- at this problem session, we're doing a quiz 2 review. We're in a much bigger room today. So I have a little bit more board space. So I just wanted to go through what's going to be covered on the exam.

First off, the scope, now, quiz 1 material will be fair game for this quiz. But it's not something that's going to be explicitly emphasized or anything like that. You should know that when we're storing graph data structures that they can achieve certain running time bounds and that kind of thing. But we're really going to be concentrating on graphs, the six lectures that we've had on graphs, the two on unweighted graph algorithms, and the four that we had on weighted graph algorithms that covered the material that was covered in the two problem sets, problem set 5 and problem set 6.

Now, usually, this material covers three problem sets worth of material. This term, it's covering two problem sets worth of material. So just keep that in mind when you're studying and you want to go look back on previous material.

In general, there's lots of graph problems that we've talked about how to solve. There's really a small number of graph algorithms. But they can solve a lot of different problems. And so we saw two algorithms to solve the graph reachability problem, single source, what is reachable from me. And I can only search a connected component of my graph from me. And the connected component from me is actually upper-bounded, asymptotically by the number of edges in my graph.

Because a spanning tree of my component has at least, or has v plus 1 edges. And so the number of vertices I can reach is upper bounded by the number of edges in my graph, asymptotically, anyway. Then we talked about exploring an entire graph, even if it's disconnected, not necessarily from a source, just touching every vertex in a graph.

Of course, we can just touch every vertex in the graph. i can look at the adjacency representation of my graph and just go through it. But this is, really, we're trying to explore the whole graph, maybe count how many things are reachable from each other in the graph. And this is what we had-- we talked about exploring a graph and counting the size of connected components in a graph in an unweighted graph.

And we could do this via full BFS or full DFS. It's basically putting a loop around one of these graph readability algorithms to explore an entire graph by exploring component by component. And when I'm done with the component, I find a vertex I haven't reached yet before and explore it again. And that still gets linear time in the graph V plus E.

Then we had special types of graphs, directed graphs, directed acyclic graphs, that we could use-- we proved this property of if we ran DFS, full DFS, on that graph, we could actually get topological sort of that graph, basically an ordering of the vertices in the graph, such that all edges go in one direction with respect to that ordering, all forward in that ordering. And we could actually use that to detect cycles in a directed graph by just looking at the topological sort order and seeing if-- look at the finishing time, the reverse finishing time order of DFS, and just checking to see whether it was a topological sorter.

Because any back edge there would correspond to a cycle in our graph. Because the proposition was that if our graph was acyclic, doing this procedure would give us a topological order. Then we had an algorithm, Bellman-Ford, that was able to detect and find negative weight cycles in our graph in a way that we've presented it in lecture.

But normally, we concentrated on these two problems, single source shortest paths, and some a little bit, all pair shortest paths, first in the unweighted context, and then in the weighted context for the majority of the lectures. So let's move on to what those single source shortest paths algorithms were.

We had, kind of, to me, increasing in generality here. The first restriction is DFS already solves unweighted shortest paths in linear time. In the unweighted context, that's all good. But for weighted graphs, regardless of the weights, if we had this very strong property on the graph that the property-- that the graph didn't have any directed cycles, then we could get this in linear time via DAG relaxation.

And then for general graphs, we had these increasing-- or decreasing restrictions on the weights. First, we had the restriction was that they were unweighted. And that's the BFS constraint. Or that they're non-negative, that's the Dijkstra constraint. And if we have no constraints, that gives us Bellman-Ford. And they increase in time.

In general, you want to choose an algorithm that's higher on this list. But sometimes, the algorithms higher on this list don't apply. If, on a quiz, you come to a graph for which-- so it's not a DAG, but you use DAG relaxation, that's no longer a correct algorithm. And so you're going to get fewer points than if you happen to use an inefficient algorithm that is correct.

So if I just-- whenever I saw shortest paths, I used Bellman-Ford, it's the slowest thing. That's probably going to be a correct algorithm. It's not necessarily going to be the most efficient algorithm. But you'll get more points, because it is a correct algorithm than if you apply a faster algorithm that doesn't apply to your problem, because it's not going to solve it correctly. Does that make sense?

So and then, in the last lecture we had, we talked about all pairs shortest paths. And really running a single source shortest paths from each vertex is pretty good in most circumstances. We don't know how to do a lot better for a lot of these. And then Johnson gives us, basically, in this last line of our graph restrictions and weight restrictions, where Bellman-Ford is right there we, can actually get a speed up over V times Bellman Ford by kind of two tricks.

Reweight the-- find if the graph has a negative weight cycles. And if it doesn't, then there exists a reweighting of this graph. So that all the weights are non-negative. But the shortest paths are preserved. And so we can use Dijkstra V times and get that running time instead.

So that's an overview of the content that we've covered so far. Just wanted to go-- just a brief overview of what these algorithms actually do. DAG relaxation finds a topological order of the thing using DFS looking at the reverse order of the finishing times. We proved that that's a reverse topological order. And then we relax edges forward in that order, because we know that we'll have found shortest paths distance to everything before us before. And we use that invariant to prove that this constructs it in linear time.

BFS explores things in levels. Increasing in the number of edges as we go out, and I just process all of the ones in the same level at the same time. And Dijkstra generalizes this notion by saying, well, I don't know all of the things that are in the same level per se, as I'm going. But I can, using a clever use of a data structure, find the next one I should process, in kind of a DAG topological relaxation order, to find shortest paths when the weights are non-negative.

Because in some sense, I know that once I've reached things from a short distance, I will never have to update their distance again. That's kind of the invariant that we're having with Dijkstra. And then Bellman-Ford essentially duplicates our graph so that each node corresponds to reaching a vertex using, at most, a certain number of edges. And then that duplicated graph is a DAG. And we can run DAG relaxation.

So that's the basic idea of all these algorithms. When I approach problems on a quiz, there's a couple of things to keep in mind. There's kind of two things that we have to worry about when you're looking at a graph problem in this class. The first thing is I might not see a graph in my problem.

I mean, on quiz 2, you know that there's going to be a graph in your problem. Because we covered graph algorithms on this quiz. But in general, some of the word problems you've been seeing on your problem sets, there's no graph defined for you. They give you an array of things, or a set of things, or some connections between some things. And that might be a graph that you want to make.

But kind of defining the graph is an important aspect of that problem solving that is not necessarily something that we've covered in lecture. We've not emphasized that in lecture so much. But it's something that you've had to do on your problem sets and something that will appear on the quiz.

So part of this is a modeling context. Can you look at a real-world situation, or maybe not so real world, but non-mathematical context, and you're trying to abstractify, put it in the language of this class, the mathematics of this class, make a graph. So that solving one of the problems that you know how to solve can adequately solve the word problem that we gave you. This is a modeling part.

So I always suggest, when you see a word problem on quiz 2 or on your problem set, it's that-- see if you can state, cleanly, an abstract problem. Relate it that if you knew the answer to that abstract problem, you could easily solve your word problem, can make it a little easier to decouple the complexity of the word problem.

Then you don't have to think about, I don't know, various strange characters we come up with in weird contexts and their weird conditions. If you can map that to just a graph with a certain-- with certain properties, and solving an abstract problem on that graph, that might be easier for you to think about and apply the material in this class. So you don't have to worry about, oh, do I have to remember that roads are connected to five other things, or do I have to-- maybe you're given as the input a sparse graph or something like that. That's a little easier to think about when applying this material.

And so converting your problem into a finding a shortest path problem, or finding a cycle, or finding a topological sort, or connected components, or a negative weight cycle, or any of these kinds of things can make it easier for you to think about. It's not fundamental material in this class that's super-- that we need to lecture on, but it is really important for you when you're out in the real world looking at problems to be able to make that transformation from a non-mathematical context to a mathematical. I like to think of this as a modeling part of the problem.

But in general, once you've got that nice abstract problem, then in general, you might have a graph. But it might not be the graph that you-- the only graph you want when you're solving this problem. That might be the input graph that you have. But in general, a lot of the kind of tricks of this thing is not modifying the algorithms that we gave you.

If you find yourself trying to modify the algorithms we gave you, that we spent entire lectures on proving their correctness, and things like that, that's maybe not something you want to be doing on an exam. Because then, you're going to be writing pages of derivation and proof that these algorithms work.

This unit in particular is much more on the let's reduce to some very powerful black box that we showed you how it works. And so because that's the framework here, the way in which we introduce complexity into problems is to make the graph non-obvious on the thing that you're supposed to apply. And so the graph that we give you is the input may be different than the graph that you'll want to use to solve the problem.

And here are some strategies that you can use to modify a graph. If you want to store state as you're traversing this graph, you can expand the number of vertices in your graph to keep track of what state I'm in. I can have a different vertex for every possible state I could be at that vertex.

In your problem session, you had this guy who's drinking when he got to bars, or every third time, and you need to remember how many times it's been since I've been to a bar, I had a drunk in a bar. And so you can duplicate the vertices to be able to store that information.

Another thing, if you need to search from multiple locations at the same time or search two multiple locations at the same time, you can simulate that without having to run an algorithm many times. You can simulate that by adding an auxiliary node, an extra node, in your graph, with edges to those sources or to those things.

And run a single source shortest path algorithm from that super node, sometimes we call it, to get better performance. It's kind of an efficiency. We're adding efficiency by changing our graph to fit the algorithms that we know how to solve efficiently.

And then the last thing, maybe it helps to pre-process the graph in some way. Some edges in the graph that we gave you might be forbidden or may need to be traversed in one direction rather than the other, even though the problem statement seems-- kind of seems like they should be traversable in either direction. And doing this pre-processing of the graph could mean that you break up your graph into-- your connected graph into a set of disconnected components that you need to find. Or makes a cyclic graph acyclic.

Or it pruned part of the graph that you don't want to explore. I never want to touch on my money my way to get to a location. So these are all really common strategies that we have, duplicating graph, adding auxiliary vertices or edges to the graph. I don't know the context in which we add edges. That's an interesting question. And then pre-processing, kind of filtering out the graph or transforming it in some way to give it properties that will allow us to solve the problem better.

So any questions about the problem-solving strategies that we have or the content-- the kind of baseline content of this class? This is kind of an overview of the lecture type material, where we're not necessarily applying this material in lecture. The rest of this quiz review session will be on applying this material to some-- a quiz from a previous term, some of those problems. Yeah, question?

AUDIENCE: What are some common ways that people lose points when they write down--

JASON KU: What are some common ways people lose points? That's a great thing. I'll add it to the notes when we post. So common things that people lose points on in this unit when they're solving problems, you're given a word problem, and you don't define a graph.

It's as easy as that. You start solving assuming that we know what graph you're talking about. When the implicit graph in the problem may or may not be correct, but we don't-- there's no graph defined in the problem. So you need to define a graph in the problem. So that's the first thing.

The second thing is, a lot of times, it's really useful, just as a strategy, when you construct that graph, tell us how many vertices and edges are in it, tell us if it's acyclic, tell us what the weights are on each edge. If you don't tell us these things, it's really hard for us to base-- to judge your application of algorithms based on that graph.

If there's redundancy, even if you define, for every vertex in my original graph I have 10 vertices, or blah, blah, blah, and maybe you're adding a super node, or all these things, it can be difficult for us to follow how many things are. So you do that bookkeeping for us, your graders are going to be a lot happier.

And so common mistakes, not defining a graph. Not specifying your graph completely. And then not-- I would also suggest that instead of just applying an algorithm to a graph, that you clearly state the problem you're solving on the graph first. I want to solve this problem, because we've given you a number of ways to solve that problem on the graph.

And if you happen to choose the wrong algorithm, then maybe that's-- separating off the problem from your implementation of how you solve that problem can maybe help you get some points for stating the problem you're solving, even if you choose the wrong or an inefficient way to solve it. So that it can really help decouple some of the things that we're going to give points on in this class.

So usually, what we're breaking up a graph rubric on grading in did you describe a graph. Did you modify it in a way that's going to help you solve the problem? Did you identify a problem that you need to solve on this thing? Did you use a correct algorithm to solve it? Did you analyze the runtime, usually involves is the size of my graph not too large, and what is the running time based on that graph? And then, argument of correctness in this unit is basically I constructed a graph that has properties so that shortest paths in this new graph correspond to whatever it is that I want in the original problem.

Some statement that links the problem you're solving in your problem statement to the problem you're solving on your graph. That's a really good statement to have to bring together correctness. But aside from that statement, you're mostly relying on the correctness of the algorithm. So you don't need to do much on the correctness side.

But forgetting to analyze runtime is a big thing. So those are a bunch of tips. I'm going to add them to the end of this slide after the lecture. Great question. Any other questions?

All right, let's get to solving problems. All right, so these problems that we're going to solve are from spring '18 Quiz 2, slightly modified. But we're just going to go through them one at a time. So the first problem we have, we have an image of black and white squares. So it's like a pixel grid. You think of as a bitmap on your computer.

And what we say is each white pixel is contained in a blob. But what is a blob? I don't know. I kind of am giving you an implicit definition of what a blob is. Two white pixels are in the same blob if they share an edge of the grid.

So this kind of tells me this graph has an edge. If these pictures are adjacent, they're both white. That's what it means. But an interesting part about that definition is that it kind of is transitive.

If I have a white pixel that shares an edge with-- a white pixel A that shares a-- let's start writing things on the board, shall we? Instead of me just talking at you. Right, we have kind of a pixel grid here. And I don't know how to do this with a chalkboard because it's white versus black. I guess I have to color in the white things. These are all white.

All right, so these guys are in the same blob, because they share an edge. These guys are in the same blob. But because they share an edge in the pixel grid, these guys are also in the same blob. Because if these are in the same blob, and these are in the same blob, there's a transitivity argument here. This guy needs to be in the same blob as that guy.

And then it says that black pixels are not in any blob. And so I'm given an n by m array. I never remember which one comes first, but we have dimensions of this thing is n by m. So we have n times m pixels.

And so we're describing, essentially, a linear time algorithm to compute the number of blobs in the image. Why do I say linear time? It's because for every pixel in my grid, I needed to give you a specification of whether that was white or black. And so if I naively gave you the input of this algorithm with a word per one of these pixels, that would be the input size of my-- and so even though this looks quadratic, the actual input size has-- is what we define as linear.

And so we're looking for a linear time algorithm to count the number of blobs in the image. OK, so what is-- this a little underspecified as a problem, I admit. I hate to admit that I was involved in this class at that time. But the idea here is if these are-- share an edge, then everything-- the observation here is if I just draw this picture, I notice that anything, kind of, that's reachable through white-white connections is going to be in the same blob.

So this is a blob, and this is a blob, and this is a blob, and this is a blob. But there's no path here. This black part is not part of the blob. Now, actually, there's nothing in this specification that doesn't say-- that says that we couldn't have these things be in the same blob. So that's a little confusing, may be a source of error. This is a source of error that I had when reading this problem after a couple of years.

But when you are looking at a problem, if everything could just be in the same blob, then you just return 1, and this problem is not so interesting. So the right way to interpret this problem, I mean, I would not need n times m time. I could just say 1. So in some sense, I'd like there to be something interesting in this problem.

And having these things that are not reachable from each other be different blobs is kind of the more algorithmically interesting thing to have. And so what is this then? This is just a pixel grid. There's adjacencies. There's connections between pixels. But in particular, I really only care about the connections between white pixels.

Hard to draw on here, but this component has a graph that looks like this. This component is a single vertex. This one's an edge here. And here's a singleton there. And if we were to construct this graph, we would have an unweighted graph, such that the number of blobs in my image would be the number of connected components in this graph.

See how I'm relating the thing that they're asking for in the problem to a property of a graph that I'm constructing? So that's really the key part of argument of correctness that we're looking for is for you to make some kind of statement connecting the two. Otherwise, you're just constructing a graph. And I have no idea what you're doing to that graph. You have to tell me.

It's about communication to us. So how do I construct this graph? Well, I can just loop through all of the pixels, look at its four neighbors, at most four, and if those things share-- are both white, then I add an edge. We're going to essentially have a graph. We're going to construct a graph. I told you to do this.

So what is V here? Then V is a vertex for each white pixel. And I can just-- I mean, from the beginning, I can just walk through all the things, find all the white vertices, maybe I identify them uniquely by their xy coordinates in this grid. That's fine.

So now I have all the vertices. And now I want to see what the edges are. I can loop through the pixels again and just look at it for possible adjacencies, see if any of them are white, stick that edge in this set. So edge is any two white pixels that share an edge.

So I can construct both of these things in order n times m, because there's, at most, that many vertices. I just loop through them. And the edges, for each pixel, I'm only checking a constant number of things. And I'm adding them to a set. So the number of edges-- the size of the number of vertices in my graph is, at most, n times m, and the number of edges is, at most, n times m times 4. It's upper bounded by that, because that's the number of adjacencies I have in the graph.

You can probably get a better bound in terms of the number of vertices. It can be, at most, V times 4. But that's a little stronger. It doesn't really matter, we're trying to get within the order n times m time bound. So anything's fine here.

So that's the graph we construct. And then we can run full BFS or full DFS. We've identified a graph. We've identified that we want to count the number of connected components of my graph. So idea, count connected components. And then, for example, using full BFS or full DFS.

I wouldn't want you to write both of these algorithms there. But when we write up our solutions, we want them to cover the space of student solutions. And so we will usually mention it. You only have to mention one of them. And because these run in linear time, this also runs in n times m. So all of these things are n times m. And we're gold. Any questions on this question? Yeah.

AUDIENCE: If I were writing down my proof of correctness and my proof of efficiency for this problem, what sort of things would you be looking for?

JASON KU: Right, so when I'm writing down-- I've described to you the algorithm. And so the question is what kinds of things do I need to write down when I'm proving-- when I'm arguing running time of my algorithm and I'm arguing correctness. For running time, mostly just check out the size of your graph. State to me what the size of your graph is here. In this case, it's order n times m.

And then I state what the running time is of the algorithm that I have is applied to that. And so because full BFS runs in O of V plus E time, it's useful to actually write this down. Even though right it's not in the terms of our original problem variables, it's useful to write this down. So that if I mess up when plugging these variables in, that you're showing your steps.

And so if you mess up arithmetically, then we can still give you points. But because the number of vertices in the graph is n times m, the number of edges is n times m, I add them together, it's still order n times m, and that would be a sufficient for an argument of running time.

And then as I was saying for correctness, most of this, the correctness of this algorithm, is relying on the fact that this thing counts connected components correctly in my graph. The key observation on a word problem that I-- or even a graph transformation problem, is that the property that you're wanting of the original graph or the original problem corresponds to the thing you're solving in a new graph that you've made.

And so here, an argument of correctness that I would be looking for, that we might allow some weaker statements, is that the number of blobs in the image corresponds to the number of connected components in this graph that I made. That's really all it needs. But I would like a connection between those values.

Now, why why would you be constructing this graph and finding connected components if that wasn't what your thought was? I don't know. But it's good, when you're communicating, to make sure that that's abundantly clear that that's why this is-- I mean, you should be able to argue why these things are-- that is connected component. You could say something like because anything reachable is in the same blob or something like that.

So that's problem 1. You've got these nice mechanical boards. So that's problem 1. Problem 2 is a little funky. It's been reworded a little bit from Spring '18. So that I could point out some other features of this graph.

We're given a connected-- so connected is in bold, so that might be an important property of our graph that we're trying to communicate to you, a connected undirected graph with strictly positive edge weights. So they're mapping to the positive integers where E is the same size as V. So the size of E is the same as the size of V. So I have the same number of edges as I have vertices.

We're trying to find a order V time algorithm to determine a path from some vertex s to some vertex t with minimum weight. So what's the first thing I notice? I notice that on this thing, I've got a graph, problem 2, we've got a graph. It's undirected. It's connected. It has this weird property that V equals E, or E equals V. And weights are positive.

And we're asking for a single pair shortest paths. We want a path, the shortest path, a shortest path between two vertices. Now, if we just were given this graph and we wanted to solve this problem, a very easy way to do that would be to just say, let's run Dijkstra on the graph. This is a graph. It has only positive edge weights, run Dijkstra on this graph.

How long does Dijkstra take on this graph? Idea 1, run Dijkstra. What's the problem with this? It applies-- we're in the context of non-negative edge weigths. We can find single source paths from s to everything else in the graph. And using Dijkstra, it applies, it's a correct algorithm. What's the difficulty with this algorithm?

AUDIENCE: Too slow.

JASON KU: Too slow, right? That algorithm would run in O of V log V plus E. In this case, these are the same. So this is asymptotically smaller than this one. It runs in V log V. So we're a little off. We're off by a logarithmic factor in our running time.

But this would at least be a correct algorithm. Whenever you approach a problem on an exam and you see a really stupid polynomial algorithm that still solves your problem correctly, you might as well write that write that down in a line. It doesn't hurt you that much to just write that down, because it's possible we give you points for that right.

But on your exam, notice why it is not sufficient. Notice that this, this is V, notice that this is not the running time bound we're looking for. We've got to exploit something different. Now, this doesn't seem-- this is a weighted context. We have weighted paths. It doesn't seem to be in one of the conditions that we can get a linear time weighted single source shortest path algorithm.

In particular, using BFS, we saw a transformation where, as long as the sum of your weights was linear in the combinatorial size of your graph we could use BFS by making each edge a bunch of undirected edges. We don't have that in this context. And this graph is undirected. I mean, so it definitely contains cycles. So we can't use DAG shortest paths. So how the heck can we do this?

Well, what does this graph look like? Here, I'm going to take a look at this condition, V equals E. So what does this graph look like? It's connected. And it's V plus E. Well, how many edges does a tree have? V minus 1.

So in a sense, a tree is the smallest number of edges you can have in a connected graph. So this has one more edge than a tree. So really, what this looks like, what our graph tree looks like is some kind of tree. And somewhere, we've got an extra edge in this graph.

It's a tree plus an extra edge. That's what our graph is. So well, let's take a step back. If I just had a tree, and I had a weighted graph here, undirected, and the weights are all positive, if any of the weights were negative, how could I solve this problem? Well, every edge is reachable from every vertex. I can just go to that edge and traverse a negative weight back and forth.

And my shortest pathway would be infinite for all our vertices. That's not the case we have here. We have positive edge weights only. Which means shortest paths are simple. And actually, there's only one simple path between any pair of vertices in a tree.

I basically-- there's one thing I can do. And in fact, if I took-- if this was s and this was t, that's an x. What am I doing? OK t, if I just ran any unweighted short-- I mean, reachability algorithm, I would get a tree, a BFS tree or a DFS tree. It would visit vertices in some order.

Now actually, in a tree, I have to output a tree that connects all the vertices. And that would be this tree. And so, in a sense, the paths that I got from BFS or DFS in this graph would be exactly shortest paths. I would just have to then go and add up all the path edge weights along the edges. That makes sense?

OK, so BFS or DFS in the unweighted context can give me the shortest path in the weighted context. Because there's only one simple path in this graph. But we have a complication here. That's not the question that we're asking. We have an extra edge.

And now, we have a property where there's not just one simple path to t. There could be two simple paths. I could go this way around the cycle. Or I could go this way around the cycle.

So that's a complication. But there's only one cycle. If t is over here, there's only one path. So if there is only one path, I'll be golden. But if, basically, the cycle can be reached between these two things, I could have two simple paths. That's the property.

We have the closest vertex on-- so this is the cycle. There's a cycle here. If this is the closest vertex to s, and this is the closest vertex to t on the cycle, then I could take either path around the cycle to get from one to the other. And that gives me my two paths.

But this path and this path, these are completely edge disjoint. In other words, any simple path from s to t, if I find this vertex going through here, it can only is one of these edges. Because I can't come back to this vertex. Once I go into it here, I got to go out one direction, and I can't come back.

So it's only one of these two edges. So the idea behind this algorithm is I'm going to find the cycle. Or in particular, I'm going to find this thing, s prime, on the cycle, find the outgoing two edges here, remove one, and then do my tree search. Basically find the shortest path by running an undirected-- I mean, an unweighted reachability algorithm, which will give me a path back to s, the only simple path in that tree. I get rid of this edge.

And I do that once. And I do it again without this edge. So that's the idea of my algorithm. So how can I do-- I first have to find s prime. How can I do that? Well, I don't know what this edge is.

But if I ran an unweighted shortest path algorithm, like BFS or DFS on here, I would get back a tree. Some edge of my graph will not be in my tree. Something like here, a shortest path to this-- I look through, I run-- algorithm, idea 2, first, find s prime. OK, and I can find s prime by run, I don't know, a single source shortest path unweighted-- I guess, run single source reachability unweighted from s using BFS or DFS to explore tree of my graph.

Then some edge is not in my tree of the graph, that will exist on the cycle, kind of by definition. It's connecting two parts of my tree. Now, I can look at those two paths from here. And the last one that they're in common from s is going to be my split point, s prime.

It's the closest one to my source that is on the cycle. Because I constructed this cycle here. So I can find edge u, v, not in the parent tree. So maybe this is u, v. Not in the parent tree. And then, find last common vertex in paths from s to u and s to v. That's going to give me my s prime.

And I can do that by-- I mean, these are each of linear size. And I can just look at their prefix. I can start from s. I can walk forward until they diverge. And the one before they diverge is s prime. That's s prime right here. Once I have s prime, I know what the edges are when they diverge.

I remove one of those from the graph. I do the same algorithm again to find a path to t. And I do the same algorithm again to find a path to t. And I see which one is shorter. That's it. There's only two of them. And so I check.

Or they could be the same path, in which case my t is actually before s prime on my cycle. Does that makes sense? So that's the idea. The last thing is remove an edge from s prime.

I don't even have to be picky about this. It has degree 3. I can just run single source shortest paths on all of them and take the min. Remove each edge from s prime, remove-- that's rigth, for each edge from s, remove and run SSR from s. And one of the paths there to t will be shortest, my shortest path in the original graph. Because it can't use more than two of those edges. That's the claim.

And this runs in linear time, because what I'm doing is I'm running single source reachability once, and maybe two more times, or three more times, a constant number of times on a graph that has size v. And this prefix finding also only takes order v. And so we're done. OK, any questions about this problem? No questions, all right, we will move on to-- what's up?

AUDIENCE: There's a hint in the title.

JASON KU: Yeah, there's a hint in the title. Actually, the original version of this problem said, instead of this E equals V specification, it said there's only one cycle in the graph. But it's in the context of undirected cycles, as opposed to directed cycles, which is usually what we talk about in this class.

We say that there's a negative edge weight cycle in the graph if we can-- usually, we're talking about-- we're allowing non-simple cycles in this class. So to remember this property about trees and to enforce this property without talking about cyclicity, I changed the condition for this problem session, this review. Yeah?

AUDIENCE: Could I also just run depth-first search on this graph?

JASON KU: Could you just run depth-first search on this graph to do what?

AUDIENCE: To find the shortest path.

JASON KU: To find the shortest path, right? So depth-first search on this path, if I ran it from s, when I got to s prime, I would have a choice on what the next outgoing age to do. So if I ran depth-first search for one of those choices, I would find a path to t. And then I would-- then I could run-- and that would find a path to t. There's only two of them, or at most, two of them. But then there's the possibility I missed this other path that could be shorter.

AUDIENCE: How would I miss it if depth-first search has to go through the other edge too?

JASON KU: It doesn't go through the other. That's the point.

AUDIENCE: [INAUDIBLE].

JASON KU: It won't go through. So depth-first search will actually go through this thing, traverse an edge, go all the way around this cycle, because everything here is reachable from here, because it's an undirected graph. It will reach back to here and then backtrack all the way. So it will actually never traverse this last edge here of the cycle. That's something you can actually prove with DFS.

Now, you could actually, while you're running DFS, try every possibility. Because MY branching factor's at most 3 at some of these things. So what I could do is-- or it could be at most four. I could connect two things with the same branching. But in general, it's a constant.

And with every choice DFS could make, I could try all possibilities. How many possibilities would that be? You get a blow up of the degree of every vertex in my graph. So the degree multiplied by each other. That's the number of times I would have to run DFS, which is exponential. A constant degree-- so a constant multiplied like 2 or 3, right, multiplied the times is 3 to the V, which is exponential in the size of my graph.

AUDIENCE: Can I have that if the size of E equals the size of V?

JASON KU: Sure, because I could still have large branching for a large number of vertices. OK, great question. All right, cool, so that's that problem. Problem 3, I have half an hour for the last two problems. I think that should be fine.

This one's-- OK, this is Doh!-nut is the problem name. Momer has just finished work at the FingSprield power plant at a particular location p and needs to drive home to a known location h. But along the way, if his driving route ever comes within driving distance k of a doughnut shop, he won't be able to resist himself and will have to go there and eat doughnuts. And his wife Harge will be angry. OK, maybe you can get the reference here.

Momer knows the layout of FingSprield, which can be modeled as a set of n locations with two-way roads of known driving distance connecting some pairs of them. And you may assume that no location is incident to more than five roads. So we've got a degree bound here.

As well as the location-- and he knows the locations that-- all the locations that contain doughnut shops. There's, at most, d of them. Describe an n log n time algorithm to find the shortest driving route from the power plant back to home that avoids driving within distance k of a doughnut shop.

OK, so we got a couple variables in here. We've got k. We've got d. But a running time bound only relies on n. OK, I see shortest paths. I see that-- I don't see an explicit mention of positive distances of the-- I see lengths. They say he knows the known driving distance connecting some pairs of locations.

So usually, I think if I were writing this problem now, I would probably be a little bit more explicit the distance is positive. But that's something that you might come into contact with. Distances are positive. And so we can have negative distances here.

So I look at n log n. I'm like, hey, what has a log in it in this unit? Dijkstra, maybe I can use Dijkstra. So let's see if we ran Dijkstra from p to h. So we've got-- we've got a graph here. We've got our graph.

So this is a word problem. So there's no graph there. So I have to define a graph. So I'm going to define a graph V, E. And we've got V. That's going to be my set of locations, locations. So this says there are order n of them. There's actually n of them.

And then E, what are we going to have? We're going to have-- it's a known pair of things, road, roads with weight equal to driving, driving distance, which, by my assumption, is going to be greater than 0. Now, that's not stated explicitly. But that would be a reasonable assumption for you to make on your exam. Because distances are positive. We would probably be more explicit about that these days.

All right, so this is a graph I could make. And I have a vertex s or a vertex p and a vertex h. And I'm trying to find the shortest path between them, shortest driving distance, driving route. So I could run Dijkstra-- wait, so what do I know about this? How many edges do I have in my graph?

I have, at most, five per vertex. So this is upper bounded by 5 times V, which is order V. So I have an order V sized graph. That's a good thing, order n, because n is the number of vertices. And so if I were to just run Dijkstra on here from p, doing Dijkstra on G from any s takes order n log n. n log n plus n, n log n is bigger than n.

So that's a nice observation. Maybe-- we have we can at least afford to use Dijkstra in this problem to find shortest distances. But what's the problem with a shortest distance found by Dijkstra in this graph. Doughnuts, like the entire point of the problem, I need to avoid being too close to doughnut shops.

So we might have a doughnut shop here. And we need to stay outside of that distance k. Or we might have another doughnut shop here. And so we got to find a path that kind of goes around these doughnut shops. So in other words, if I have a vertex in my graph where I can reach a doughnut shop within-- shop within distance k, I can never visit that vertex.

Because then, Momer will not be able to resist himself and will have to go eat a doughnut. So that's the thing we're trying to avoid. So how can we do this? Well, here's a silly thing. I could run Dijkstra from each of these vertices, these doughnut shops, find all the things reachable in k driving distance from them. And then remove those vertices from the graph. That's an idea.

But how long would that take? That would need me to run Dijkstra d times, because there's d doughnut shops. I could run Dijkstra d times. So that gives me a running time bound of I run d times to filter out the graph, to modify this graph, and then I do one more to find the shortest path if there is one.

But in general, that's going to take d times n log n, not n log n. I have no bound on d, except that it's under n. So it could be n. And that would give me a bad running time. So we're going to use a very similar trick here to one of your-- I think a previous review session-- stop, there we go. Is to, when you want to find things, if we want to prune a graph from multiple locations, one of the things we can do is, any tricks?

Supernode, right? I can have a vertex. Well, maybe I don't want to put it up yet. OK, if I have all of these doughnut shops, what I can do is provide a-- I guess, I guess these are unweighted, undirected edges. Here, we can model all of those directed things by two undirected edges. It doesn't really matter.

But here, I don't want to be able to go back to my supernode. But what I'm going to do is I'm going to add a supernode with edge weight, say, 0 to everything else. And then, if I ran Dijkstra from the supernode, and found all vertices reachable within distance k, well, I didn't-- I didn't spend any of that distance going through this first edge.

And I didn't come back to s, because these things are directed into the things. And so anything I write as going to be within distance k of this doughnut shop, but for all doughnut shops. In some sense, I'm doing this search in parallel. So this is the same trick that we had in the you're looking through the sewer network, or something, and they're trying to avoid monitors, or sensors, or something like that. We actually did this transformation and then binary searched on a distance. It was kind of involved.

But this is an easier example. Now, you can actually generalize this further. What if each doughnut shop had an amount that Momer liked it? So if Momer is within a larger distance of a doughnut shop he really likes, he still won't be able to resist. But a doughnut shop that doesn't make very good donuts, he'll be able to resist a shorter distance without having to go to that doughnut shop.

So in other words, each one of these doughnut shops has a different k, a different radius that Momer will allow. Is there any way to generalize this technique to be able to prune all of those vertices instead?

AUDIENCE: Put weights on it.

JASON KU: All of these had weight 0 before, the same weight. I mean, the algorithm would have worked for any weight I put on all these edges, as long as I searched a distance that weight plus k. Here, I can just make the distance of this frontier for each one of the doughnut shops the same by modifying the distance of the incoming edge.

So I can set the length-- the weight to the doughnut shop with the largest radius to 0. And then put the difference between the largest radius to all the other ones, I put that is the weight on the other edges. And then we still have a graph with positive edge weights.

And I can run Dijkstra from this. And that would generalize this problem and something that we've done in some practice exams, or in exams and problem sets in the past. So that's another common way. So we filter forbidden-- there's two d's in forbidden, or two d's. There's a three d's, vertices, by using supernode plus one run of Dijkstra.

Those are the extra letters. So on your exam, you would probably want to be a little bit more explicit. This is a summary of the things that we just talked about. I just talked 10 minutes about the algorithm.

But it doesn't hurt to add a summary at the top of what you're going to write. This is the approach that we're going to have. We're going to filter out the vertices from G, essentially, by running Dijkstra from each of these doughnut shops. But we're going to do it in parallel by adding this supernode. Actually, on another recommendation I have for you on an exam is that almost any problem that we give you in this class, can get 80% to 90% of the points by writing maybe three lines.

Almost, maybe not some of the data structures problems, but almost any question in this class can be solved with not all of the points, but most of the points, by just writing a couple of lines. That we know that you know how to solve the problem.

And this would be one of those situations. Now, I would want, to give you full points, I would want all the details here. Did I construct a new graph? I add this vertex here. I have to add edges to each of the d things. But I've only added d more edges and one more vertex.

So it still has this linear size in my input. And then I want to say that I'm putting the weights on here based on what the distance is. Now, they're all the same weight, because I don't have that generalization. And then I run this thing.

And I remove all those graphs. And I construct a new graph from G. That's a third graph that I'm constructing now. But notice that the graph that was implicit in my problem was very different than the graph that I'm ultimately running a shortest paths algorithm, like Dijkstra, from p to see if a path exists. Does that make sense? Any questions on this problem?

All right, we've got 20 minutes for my last problem. Man, I'm not using-- I write much less than some of your other instructors. So I like to talk more than I like to write, apparently. So problem 4, let's take a look at this. This one is one I made up last night, kind of fun.

Long shortest paths. Given a directed graph having arbitrary edge weights, basically, these could be positive, or negative, or 0, and 2 vertices from the graph, describe a V cubed time algorithm to find the minimum weight of any path from s to t. OK, that sounds like Bellman-Ford right there. But I have this last condition.

And the last condition is a little weird, containing at least V edges. So I want to short path in terms of weight. But I want a long path, in some sense, in terms of the number of edges I traverse. Does that make sense?

So of all paths having at least V edges, I want a shortest one among them in terms of weight. This is a weird frickin' problem. Usually, we're not trying to do this max-min kind of thing. You've got two different quantities here we're trying to optimize. Anyone have any ideas on how I could approach this problem?

What does this sound-- what does the at least V edges sound kind of similar to that we might have talked about in lecture?

AUDIENCE: [INAUDIBLE].

JASON KU: So when we were talking about Bellman-Ford, we defined this thing called a k edge weight. It's the weight of any path using, at most, k edges. This kind of edge constraint seems similar, except it's kind of the reverse. It's not at least, it's most-- it's not at most, it's at least.

Well, here's an observation I have for you. If I want a path that goes through at least V edges, some prefix of that path uses exactly V edges. That makes sense, right? So maybe it makes sense for me to-- maybe it might make this problem easier if it's not at least V edges, but if it's exactly V edges. Maybe I think about it that way.

That seem a reasonable other way to think about this problem? I knew how to do up to a certain set of edges. Here, we're asking for at most. Maybe the thing in between is a little easier to think about.

So what we're doing-- we're given a graph G. It has any weights. It's possible that this graph has E lower bounded by a quadratic in the vertices. I have no restrictions on how many edges this thing could be. And so the worst thing I could have is that this thing-- I mean, my graphs are simple.

The worst thing I could do is have this be quadratic in the number of edges, say, if it's the complete graph, it's the maximum number of edges that I could have. And I'm trying to find, in my graph, a path that uses a lot of vertices, but has small weight. Now, what's another thing to notice here is if I use at least V edges, can my path be simple? No.

Because I need to use at least V plus 1 vertices. And there are-- that's more than the vertices I have in the graph, obviously. Now, it could go through vertices more than once. But it's definitely not going to be a simple path.

So what's one thing-- what if there's a negative weight cycle in my graph? What's the minimum weight of any path from s to tee if the negative weight cycle is reachable from s-- reachable on a path from s to t. What is the answer to my problem then?

Negative infinity, right? Because certainly an infinite length path is going to use an infinite number of edges. If it's going arbitrarily long, then I can just run Bellman-Ford. So that's one thing I can do. I can just run Bellman-Ford on this graph. I have enough time to do that, because E is upper bounded by V squared. And I V cubed time.

And if there's a negative weight cycle on my graph, I can know that the minimum weight of any path is minus infinity. I detect that that's the case if, basically, t is reachable from s with a minimum of shortest path distance minus infinity. That the path that achieves that is going to have more than V edges. So I'm done. And no path actually achieves that. But that's the infimum, supremum, or infimum, sorry, we're going lower bound. I'm thinking of long paths. So the number of edges is approaching infinity.

But in the context where I don't have negative weight cycles, actually, one of the things we showed was that if you're reachable not through a negative weight cycle, or if no negative weight cycles traversable from s to t, then my shortest path is going to be simple. But that doesn't seem to apply here either, because we need to have a non-simple path. So what do we do?

So let's go back to this idea of trying to figure out the minimum weight of any path using exactly V edges. Can we use some of the tricks that we had in Bellman-Ford, when we're keeping track of the number of edges we're going through at a given time. That's the idea.

If we have a vertex-- if we have a new vertex for each vertex, different versions of it that talk about exactly how many edges I went through, then maybe I can keep track of this while I'm working on this graph. So let's say I have multiple layers of the graph. This is the idea.

Maybe we start at level-- level 0, down here to level-- how many edges do I want? I want V plus 1 vertices. So I'm going to have V plus 1 levels, which is V. So a level here, V, how many levels are there? There's V plus the 0. And so there-- I'm going to have, for every edge in my graph, I'm going to take it.

So this is a directed graph. So I direct it down into the next level. For each version of this graph that I have, I take that edge that was originally between u and v here in the graph. It was originally here in G. But here, I've pointed all of those edges downward.

Isn't that what we did in Bellman-Ford? We made one other addition in Bellman-Ford, to make it be the at most property. What was that transformation we did?

AUDIENCE: [INAUDIBLE].

JASON KU: We had 0 weight edges going from each vertex to another. It meant that we didn't have to traverse an edge. But here, if we don't add those edges, actually, this transformation gives us that any path that goes through V edges will be some path from a vertex in layer 0 to a vertex in layer V. Just because, to get down here, I had to traverse exactly V edges. And they're edges of my original graph.

Now notice this encodes non-simple paths as well. Because these things could go-- I could go here and back to u, and back to v, and back to u, if I had a cycle in my graph. But actually, what kind of graph is this? This is a DAG.

So this is DAG. Maybe, I call this G prime. How many vertices are in G prime? v times v plus 1, so I'm going to say order v squared. And how many edges are in my graph? V times E, I copied every edge, made it directed down between each level. There are V transitions between levels. And I copy each edge for each of those.

So the number of edges is order V times E. So this graph is blown up. There's a lot of things in this graph. But I notice that this graph has size order V cubed is what we're going for. So I can afford to construct this graph, since V is actually also upper bounded by V squared by simplicity.

OK, so we have this graph. We could find our vertex s here, s0 up here. And we could afford to compute the shortest path distance to all other vertices using exactly V edges in my graph, exactly. That's what we could do.

I can find everything reachable from s0 in this graph and calculate the shortest path down here at the bottom. So I can do that in V cubed times because DAG relaxation is linear in the size of the graph. But that's not what the problem is asking me, unfortunately. In particular, I could find the path to t, to t, v. And that would give me the shortest path using exactly V edges.

But that's not what I'm asking for. I am asking for at least. So it's possible that I get down here to some other vortex. And maybe there's a negative weight path going to t. And I want to be able to find that. So how can I do that? How can I allow paths to continue past this an arbitrary amount?

I could have more layers. Actually, simple paths from any-- I mean, shortest paths that are simple, that use fewer edges, here, I'm not restricted on the number of edges I use. So shortest paths in this graph are going to be simple, because there's no-- I can already throw away the case where I have negative cycles, because I ran Bellman-Ford at the beginning.

I can-- so I know that I'm going to want a short-- a simple path after I've reached V edges, because it's never going to be beneficial to me to come back to a vertex, because that will be a path of longer weight. This is the kind of surgery argument we had, both in unweighted and weighted context.

So these are going to be simple. So I know that I only have to go V more layers at most. So that's one way to look at it. I could add more layers of this thing, find the shortest path distance to all vertices using up to 2v edges, maybe even 2v minus 1, but order v. And then for all of the ones down below here, I just look at each vertex. And see which weight is the minimum.

Another way-- the way I like to look at it, which is a little bit more fun, I think, is once I'm down here, I'm just trying to find simple paths in the graph from this vertex v-- to this vertex. So one of the-- so actually, have these go up. So actually, on this bottom layer, I want to find short paths to t from, actually, every vertex.

And I actually know what the short-- just from what I did up here, DAG relaxation on this graph, I knew what the shortest path distance was from s0 to each of these vertices. Because I did that in V cubed time up here with DAG relaxation. So I could add a super node to this thing with a directed edge to each vertex with the shortest-- with weighted by the shortest path distance I found up above.

Now, I have a graph where any path from s0 to tv here will be a path that uses at least V edges in my original graph. because these represent the shortest path weights of anything using exactly V edges. And then the path can continue in the original graph.

So now I have a new graph here, such that every path from here to there corresponds to a path that I'm looking for. So I want to find a minimum weight path in this graph. How can I do that? Now, this graph might have negative weights.

AUDIENCE: Bellman-Ford.

JASON KU: I can run this with Bellman-Ford, Bellman-Ford. I can do that again. Sure, why not? Now, Jason, why couldn't we just add a bunch of edges here? Add our original edges here in the bottom layer of this graph and run b Bellman-Ford on this entire graph? Why couldn't I do that?

AUDIENCE: Too big.

JASON KU: It's too big, right? The number of vertices is v squared. The number of edges is potentially v cubed. Running Bellman-Ford on that huge, duplicated graph would give me a v to the fifth running time, which is awful. In a sense, we're separating out the complexity.

The upper part of the graph has very nice DAG structure. So let's do shortest paths in that DAG structure. And then reduce that complexity down to just being the thing that has the cycles that we are worried about. Reduce the complexity down here.

So how big is this graph? This graph has V plus 1 vertices, because I only added one supernode here. And it has E plus order V edges. I want to be careful here. But this is linear in the size of the original graph. So running Bellman-Ford here only takes V times E time, which is V cubed.

So that's two different ways how to solve this problem. One using a bunch of graph duplication and having the insights that going, at most, v more steps of this graph duplication could never get a better thing. So I can stop. Or recognizing that, well, I have this very powerful algorithm here that can find shortest paths, simple paths in a graph without negative weight cycles. And I can use this supernode to transfer a part of my graph with a lot of nice structure down to this other graph.

Any questions about this problem? So these are some-- we got two abstract problems for you, two word problems for you, with a lot of different transformations and a lot of different tricks of trade. Any of these would be something that has either appeared on an exam or is at a level of something that could appear on your exam. So go ahead and take a look at the practice material that we've posted and are accessible from previous years' websites. And wish you luck in working on graph problems on your exam.