[SQUEAKING]

[RUSTLING]

[CLICKING]

JUSTIN SOLOMON: Well, welcome to problem session 5 of 6.006. It's a pleasure to see all your smiling faces here today. This week, we're going to cover some problems in graph theory related to depth-first search and breadth-first search, which were roughly the topics that I guess we've covered in the last couple lectures and what's going to be on your homework.

And I believe this is basically the homework from last year with a few revisions, based on some typos we caught. Oh, and I caught a spelling mistake that I'll bother our instructor about later. OK. So without further ado, let's get started. I guess we'll just do them in order for lack of creativity here.

The very first problem has to do with some measurements on a graph, which is actually a really interesting one to me. So it turns out that in a lot of research in-- for some reason, in computer science, there's graph theory research, and then there's networks research. And these are two different communities for weird historical reasons that I don't totally understand.

But people in the network science literature often measure things like the radius of the graph and some other kind of measures that are trying to tell you something about, like, is a graph a long, spread-out thing, like a line graph, or something super compact, like a star? And so on. And so this problem is kind of digging into the algorithmic aspects of how we might compute one of the measurements that I believe is fairly common in that community.

So let's kind of go through these problems. As usual, in 6.006, we like to take actually relatively straightforward computational problems and then dress them up with a lot of language to make it annoying for you guys to parse. And indeed, this problem is no exception to that. So in this problem, we're given an undirected graph. And as usual, we will call him G.

And we define a particular number that we're trying to measure in this problem, right? So in particular, if we're given a vertex v, then we can define something called the eccentricity of v, which is the distance to the farthest-away thing. So in particular, we can define-- it's going to be given by the following, which is the max over all the possible-- I'll try to make sure my notation-- oops, I've already-- well, that's OK-- over all the other vertices of the distance from v to w.

So I'm standing at a point on a graph. And now I like make a loud noise. And the last person to hear me, the distance to him would be the eccentricity of that vertex. And so this is some kind of notion of radius or diameter, but sort of planted at a point. And then if we want to learn a property not of a vertex, but of the entire graph, one thing we can do is define the radius.

And that is given by R of G is the min over all of the different vertices, u, of the eccentricity of u. OK, so I think this is one of these definitions that's really annoying to parse and think about. So we should draw a little bit of a schematic and see what's going on here, because especially as a geometry professor, this one's kind of nice, because it translates directly to what you might do in metric geometry.

So let's say that I have a circle here. And I want the world's most complicated way of defining its radius. So for any given point-- there's a point in a circle. That's a circle, in case you were wondering. You know those internet contests where they have people that just walk up to the board, and draw perfect circles, and then leave? I unfortunately am not an expert at this matter.

But anyway, so if we think of a point-- like a circle as some analog of our graph, and then I draw a point, which might be the analog of a vertex, then what is the eccentricity? Well, it's the distance to the farthest-away point, right? So for this guy, it might be the length of this line, roughly, because that's the distance to the farthest-away thing.

So for every different point that I draw, each point has its own farthest-away point in the circle. So there's some positive number that's assigned to every single point in this domain. And if I take the minimum of that positive number, where do you think I end up?

AUDIENCE: Center of the circle.

JUSTIN SOLOMON: That's right, Jason. I end up in the center of the circle, because if I think about it, the distance to the farthest-away point in this domain, one thing you can convince yourself is that that's sort of as small as possible. So this is what we might call a min-max problem in optimization, because we are minimizing the maximum distance, yeah? This also shows up in game theory, all kinds of different places that solve this stuff. But thankfully, in this particular problem, we're not going to need all that.

OK, so right. So this homework problem has two parts. The first is to give an algorithm for computing the radius of a graph. And then the second one is to give an algorithm for approximating the radius of the graph really quickly, or more quickly than the first part. I actually don't know if there's a lower bound there. But come back to that later.

OK, so in part a, we're given G. And moreover, we're given one additional piece of information, which we actually do need in this problem. I think it's one of those words that kind of slips past us when we read a graph theory problem. But it's important to pay attention, of course. And that is, we're given G. And it's connected. I suppose, really, it should be given connected G. But that's OK.

Now what we want is to compute the radius of G in time that looks like the product of the vertices, the number of vertices-- oops-- times the number of edges, or the number of times the number of vertices. Your instructor struggles to speak and write at the same time. But it's a skill that I'm working on. And frankly, handwriting is much easier with this little chalk.

OK. So essentially-- I used to have a math professor in college that used this phrase all the time that was just like, it's important not to think here. The problem asks you to compute the radius of a graph. And in some sense, there's an algorithm that just writes itself for computing the radius, right? Because the radius is the min over all the vertices, of the eccentricity. The eccentricity is the max distance.

So what would be the simplest thing to do here? Well, in some sense, it would be to loop over all the vertices, compute their distance to all the other vertices, and take the max for each one of those and then the min over all of the guys in the outer loop. Since I just said a sentence that I'm realizing doesn't parse particularly well, let's sort of write down what I mean, which is to say, we're going to think of there being an outer-- that's why we don't use this chalk-- an outer for loop, which is computing this min.

So-- right? Well, what are we going to do? We can compute the shortest path distance to all of the other w in my graph, take the max of w-- of distance from v to all the other w's. Obviously, we can kind of do these two at the same time.

And then, if this number is bigger than my current max, keep it. Oh, yikes. If it's smaller than the current estimate I have of the radius, then I keep it. And if it's not, then I throw it away, right? So maybe I initialize my radius at infinity. And now let's call this number, I don't know, little r. If little r is less than big R, then just keep it around, right?

And so if we think about it, I don't think it's terribly hard to prove that this algorithm is correct, because it's sort of just taking our definition of what the radius of a graph is and translating it into a braindead algorithm. So I think really, the challenge here is proving the runtime in this particular algorithm.

So what does our runtime look like? So we have a loop over vertices. So I kind of incur a factor of mod v here. And then, well, our graph is unweighted. So one strategy for computing the shortest path distance would be breadth-first search. I think that's what's in my notes. Yep.

And in general, breadth-first search, if you recall from lecture, takes mod v plus mod E time. So the question is, OK, so if I multiply these things together, what do I get? I get O of v times v plus E, like that, time.

But like, uh-oh. That's not the time that my homework problem wanted, right? Because the homework problem asks you to solve this in just mod v times E time. And somehow we've incurred an extra factor. And now we have to figure out why this is actually OK, or we have to fix our algorithm. But in this case, it turns out that this runtime is just inaccurate, OK?

What's our intuition here? Well, I kind of underlined it for you here. Our graph is connected. And in particular, there's going to be a nice property of connected graphs, which is that the number of edges dwarfs the number of vertices here. So really, if we have v plus E, in some sense, this is going to look like a constant factor times E plus another E here. So this whole thing is going to be v times E time, yeah?

So let's make that argument a tiny bit more formal here. So in particular, we know that G is connected. And every vertex-- so in particular, what can happen here is-- OK, unless my graph consists of one vertex, which is a case you could dispose of pretty quickly, what I can't have is a graph that looks like this, like one vertex and then an edge floating around there. Everything has to be connected together-- connected together.

OK, so in particular, what this means is that every vertex is adjacent to at least one edge, again, except, I guess, technically, the one vertex case. But I think we can convince ourselves that for any graph of constant size, we're not terribly worried about it, right? It's just the asymptotics that matter in this problem.

OK, so if every vertex is adjacent to one edge, well-- and remember that every edge, kind of by definition of an edge, is adjacent to two vertices. Then what we can conclude is that the number of vertices is less than or equal to the number of edges divided by 2. This is a conservative estimate.

And so in particular, what does that mean? It means that v is big O of E. This is a case where we have to be quite careful about big O being an upper bound, right? In this case, typically, v is much less than E-- well, depends how many edges, like if you have a really dense graph or not.

But in this case, what does that mean? That means that mod v plus mod E is really just big O of mod E, right? Because this is big O of mod E plus big O of mod E. And that means that our problem really runs in v times E time, which is what we wanted in our problem. Are there any questions from our audience on part a here? Cool.

AUDIENCE: I don't quite understand why-- where you went from the first statement to the second statement there. At least one edge implies v is less than or equal to E over 2.

JUSTIN SOLOMON: Oh, yeah. So I guess there's sort of two things that matter here, right? Every vertex is adjacent to one edge at most. And every edge-- yikes. Every edge is adjacent to two vertices. I guess, actually, it's the second one that matters.

So you can never have a vertex just floating by itself. So one way that I can count my number of vertices is by looking at the number of edges and saying that, well, every edge can touch exactly two vertices. Every vertex has to touch exactly-- well, at least one edge.

So if you put those together, you can convince yourself that this bound has to be 2. If you want to be conservative about it, you can just get rid of the divided by 2 here, I guess. It doesn't really matter. Any other questions from our audience? Cool.

All right, so now, let's take a look at part b. So in part b here, they ask us to basically do some version of the same thing, right? They want us to now approximate the radius. But we're given a smaller budget of time. So now what we want in number b here is, compute an R star such that-- I got yelled at in my textbook that st should always be "subject to." I got an angry review of the textbook I wrote because of that, which was puzzling to me. But amazon.com is not a great source of useful data.

But in any event, we want R star, which is sandwiched between the radius of G and 2 times the radius of G, like that. Now, notice-- so in other words, we want to-- the first thing to notice is we want to upper bound the radius of our graph. And already, this should suggest to us how we might solve this problem, because if we take a look back at our definition of radius over here, notice that the radius is a min, right?

So what's going to happen if I returned epsilon of some other vertex? Well, it's lower bounded by the radius, because the radius is the smallest possible epsilon over any vertex. That make sense? Now, when I was doing this problem, because, you know, I'm the dumb instructor of the three, I said, well, OK, but maybe I need to be somehow judicious about what vertex I choose.

Like, well, in some sense, what this suggests is that maybe I choose some other vertex, and compute its radius, and return that as our approximation. But of course, the problem wants me to sandwich it between two values here. So in addition to upper bounding R, I want to be less than 2 times R. In other words, my approximation is within a constant factor.

I tried some weird stuff, like farthest point sampling and so on. Then I realized that you actually don't really need to do any of that. One thing you can do is literally choose any vertex, return its eccentricity. And that's actually good enough. So here's our algorithm. Let me go back to my notes here. I don't know why I'm following my notes, actually. I could do this off the top of my head. But they make me feel better if I'm looking at them at the same time.

So in particular, what I'm going to do is choose u and v. Let me be clear here-- any u and v. So if I'm using some data structure to store all my vertices, I just take the first one, whatever. And two, I'm going to return R star is equal to epsilon of u.

Now, of course, this isn't really an algorithm. If you do this on your homework, you'll lose points. And the reason is that I haven't told you how to compute this value here. So if you were to write out your answer for this problem, of course, you should tell us that, like, really, to compute epsilon, what do I do? I use breadth-first search to compute the shortest path from u to all the other vertices. And then I guess I take the max value here.

OK, so I think you guys can fill in the details of the algorithm. The bigger challenge is going to be to prove that this is actually a good bound, right? And so, in other words, what we need to prove here-- I don't know. Like, there's a claim. There's a proposition. There's a theorem, somewhere on that axis.

I'm going to call this one a claim. I'm going to downgrade it. And that is that the radius of my graph is less than or equal to R star, which is less than or equal to 2 times the radius of my graph. OK, so let's prove this thing. I'm managing to use all of my boards on one problem here.

OK, so in particular, to prove this claim, I need to prove two inequalities. This is like two homework problems in one. So let's number those off. There's 1. There's 2. OK, so let's do inequality 1. I think we can squeeze him into a relatively small space.

So remember, what is the radius of my graph? Well, just by definition, we know that it's the min over all possible u of epsilon of u. So in particular, what's-- the nice property about the minimum of something is that it's less than everything else or equal.

So this-- maybe let's call this u0, just to distinguish between that and the notation I have on the left-hand side. This is less than or equal to epsilon of u, because, I don't know, because min, yeah? So that actually already-- and of course, this is exactly what we chose to be our R star. So our first part of our proof is done here. So this is the easy part.

And sometimes, like, this is sort of what inspired our algorithm. So we expect this bound to be kind of straightforward. OK, but the other half of the problem is a little more tricky. And actually, there's a solution in the notes. And then I decided, just to make it a little more inaccurate, to write up my own. But actually, I have I have an ulterior motive, which is I notice in this class we don't tend to use a tiny piece of notation that I like. So for my convenience in future problems sessions, I thought I'd introduce it now.

So we're solving a minimization problem. The nice thing is that in this class, everything we do is finite. If you take my graduate course, that's not going to be the case. In fact, actually, in lecture 2, we're going to do like variational calculus. But in this course, what does that mean? That means if I minimize a function, there is actually a vertex in my graph that achieves that minimum, right?

This is different than like, for example, if I-- then I'll shut up. But if I wanted to minimize-- here's f of x equals 1 over x, and I ask you for the minimum value. Well, it's over all x greater than or equal to 0. Well, the minimum value is 0 if I take x off to infinity. But it never quite crosses 0. So you're kind of in this weird universe.

If you remember Jason's lecture, he talked about infs and sups as opposed to mins and maxes. But this can't happen in our problem, because when we compute a min, there's actually a vertex that achieves it. And that vertex, we call arg min. And so this arg here stands for argument.

So one thing that I can do is say, OK. So remember that my radius is the min, the min over all u of epsilon of u. Then I'm going to define a vertex, u0, to be the arg min over u of epsilon of u. And this is just fancy notation for saying, give me the actual vertex that makes this value as small as possible, yeah?

The nice thing about this problem is that we're not worried yet about how we make runtime. So I can construct this kind of thing and not worry about how I actually found it, right? OK, so let's say that we did that. So this is, find me the vertex that actually gives me the radius, right? So in other words, I find that vertex. And then I find his or her farthest-away vertex and measure the distance. And that distance is the radius of my graph, OK?

So let's actually do that. So in particular, then I can define a second vertex, v0. Well, how does the radius algorithm work? I find the central guy, and then I find the one that's farthest away. So we're going to make him the arg max over all v in my graph of the distance starting at u to any v.

So if I think about my circle-- that's a circle. Then u0 is like that center of my circle. And then v0 is like that far-away point. This is a schematic, right? My circle is really a graph in this problem. But I think the analogy actually works.

OK. But in reality, my algorithm was braindead. I didn't actually compute u0. I just randomly drew-- sorry, I shouldn't use that word. I arbitrarily drew a vertex, u, and then computed the farthest-away distance from that guy. And of course, what we have to check is that that thing is within a factor of 2 of what I wanted.

So OK. If I have u, then I'm additionally going to define one more thing called v. And that-- oh boy. OK, so I'm noticing I'm saying one thing, and I'm writing another. u0 is the center of my graph. I think I said it. I just forgot to write it. And then this v is the farthest-away guy from him.

So basically, the subscript 0 here means-- is the platonic ideal of what I wanted in my problem. And no subscript is going to mean the other one. So now I compute the farthest-away thing from the u that I actually chose in my algorithm. That's some v bar.

So again, remember, my algorithm just says, OK, I'm going to choose some other point, v, and then return v's distance to some farthest-away point-- oh, sorry, choose another-- oh boy. Choose a point, u, and return his distance to some far-away point, v. I think I've managed to lose everybody, knotting together u's and v's here.

OK, so why'd I introduce all of this notation? Because this is what's going on in this problem, right? To actually compute the radius, I want to find the most central point, u0, and its distance its farthest-away thing, v0. In reality, I arbitrarily chose the point u. And I returned u's distance to some point, v. And I want to show that those two things are within a factor of 2 of each other. OK, that summary makes sense even, if I talked in circles for a little while.

OK, so let's actually do that. So remember that the thing that I'm going to actually return is R star. And that is equal to the distance from u to v now, because I just made all these definitions. And now I get to use my favorite inequality. In fact, this is sort of the only inequality we know in this class so far, I think, which is the triangle inequality, which says that, of course, this is less than or equal to the distance from u to u0 plus the distance from u0 to v.

So in other words, this is saying, the shortest path from u to v is always upper bounded from the length of the shortest path from u to u0 and then u0 to v, right? This is drawing a triangle. Aha! But take a look. What is the actual radius of my graph?

Well, in my notation, the radius of my graph is exactly the distance from u0 to v0. And this thing is bigger than the distance from u0 to anything else, by definition, for all v, right? So if I flip this inequality backward, well, take a look. This is the distance from u0 to something. This is the distance from u0 to something. So I incur two factors of the radius. And I get the bound that I wanted, yeah? And so this is a slightly more formal little proof of exactly the same thing that's in the homework notes.

OK, so the one thing that's remaining is to actually show that our algorithm runs in a reasonable amount of time. So I think they give us a budget of order E time. But notice, that argument is precisely the argument that we just made right here, just minus the v factor. And the v factor just came from looping over all the vertices in part a. So now I think we're done with problem 1.

As usual, I've wasted too much time on the easy problem. All right, any questions about this one? Excellent. Well, now that I've written too much, let's do the rest of it. I spent time on this problem because I like it. It looks like a geometry problem. OK.

So now, let's see. In problem 2, which I noticed that this homework is kind of full of prototypical 6.006 slash graph theory problems in general. Like, they just go down the list of things that people typically do in graph theory that are useful tricks to know.

So I would suggest to the students in this class, even if it's pass-fail, look very closely at this homework before doing the current one. I think the ordering works out that they can do that, because I think you'll get some good hints for how to solve all the current homework. So you heard it here first, guys.

OK. So in problem 2, we're talking about internet investigation. So in particular, at MIT has a bunch of different routers that are connected by cables to one another. And essentially, what are we given? We're given a bunch of different routers. And we're given the length of the cable in between them. And the latency, unsurprisingly, is proportional to the length of the cable.

That, in my abstract understanding of how computers work, kind of makes sense to me. I'm not sure that's actually true. But that's sort of immaterial for 6.006. I assume our department has a networks class if you're interested in that kind of thing.

And essentially, what we're trying to do is sum up the latency over all of the routers. So let's break down a little bit of notation here while I continue to dance all over the room here. I keep losing my chalk. I need like a holster. I feel like that would be useful for the chalk bucket.

OK. So now we're going to do problem 2 here. So we're given r routers. And some of them are marked as entry points. And now we have a bunch of bidirectional wires, wi, each of which has length li. And that's a positive integer value here.

And actually, because of this-- so technically, I think a lot of students in this class have encountered weighted graphs before. But if you think about the narrative of this course, I think, for the version of this homework, we haven't really encountered weighted graphs yet. But a better way of putting it, rather than psychologically diagnosing your instructors, is that what we're going to find is that there are often problems that look like they're weighted graph problems, but they really aren't. And this is a nice example where that's the case.

OK, so we define latency as follows, that it's at least proportional to the shortest path to an entry point. And now we have two additional assumptions that we need, right? One is that the total latency, or at least the latency of every vertex, which is the same thing-- latency-- is less than infinity.

What is this really saying, by the way? Like, when would the latency be infinity? It would only be infinity if I like took a pair of scissors, and cut a wire, and just connected for the rest of the network. Yes?

AUDIENCE: Every router is connected to some entry point.

JUSTIN SOLOMON: Yes, exactly. Like, there's some path from every router to some entry point. Doesn't necessarily mean the entire graph is connected, I guess. But at least you can always get to an entry point. And then, moreover, and this one's the real kicker here, that there's at most 100r feet of wire.

Incidentally, r stands for routers. I had the previous problem in my head and was thinking radius a long time. So don't be like your instructor. And actually read the entire problem before getting hung up on it. But in any event, the thing that you're trying to do is to compute the sum over all of the routers-- I don't know, r, whatever-- of the latency of that router. OK, so that's our problem here.

Incidentally, this little goofy exercise I just did of taking this paragraph problem and kind of writing it in bullet points, I find helps me a lot when I'm trying to solve these algorithms problems, because I think it's really easy to just get like thrown off by a wall of text here. OK, so this problem is screaming out graph theory. Like, we're practically using the terms here. We are using the terms, right?

Like, we've got nodes that are kind of like routers. And maybe edges are kind of like wires. But there's a bit of a catch, which is that your runtime-- at the end of the day, I think you want order r runtime. That's where things get a little funky initially. And so we have to think a little bit carefully about how to do it.

And here's going to be the trick. So this is starting to look like a shortest path problem. But what would you maybe not want to do? Would be to iterate over every single router, or every single vertex and every single router, and compute the shortest path between every single pair, because if you did that-- oh boy, I'm confusing my terminology. There are entry points, which is the thing that I need to compute the distance to. And I need to iterate over every single router and compute its distance, maybe, to all the entry points, and then take the min, or something like that.

But if I had a double for loop, then I'm probably not going to get order r time, right? Because somehow, you expect it to look like something squared, or like the product of two terms. So we have to be a little more sneaky than that. And we're going to use sort of a canonical trick in graph theory.

OK, so let's follow the Toucan Sam approach here. We're going to follow our nose and say that, OK, there's basically a graph that's staring us in the face in this problem. But then we're going to have to make a little bit of an edit, because we'd like to use the kind of linear-looking time search that BFS affords us.

But it looks like we have edge weights in our graph, because the wires are associated to lengths, right? Different wires have different sizes. But we have this nice fun fact, which is that the total amount of wire in our whole universe is less than 100r. I guess the units of this 100 are kind of weird, right? It's like feet per router or something, but whatever.

OK, so in particular, I'm going to make a graph with the node per router. So like, maybe here's a router. There's another router. There's router 1 and router 2. But since I want to use the sort of linear time advantages of breadth-first search when I'm computing distances, I can be a little bit sneaky about this, which is to say, instead of having like 10 feet of wires, I'm going to have 10 1-foot wires, yeah?

Except now I'm additionally going to have little chains. So here, maybe the length l 1, 2 is equal to 3, right? So I'm going to put three edges in between. So in other words-- and I'm going to connect them with chains of li edges for each wire.

Does that make sense? So essentially, I'm going to take my weighted graph problem and make it unweighted by just like repeating a bunch-- well, not really repeating, but chaining together a bunch of edges so that the total length of this thing is equal to the distance from one router to another.

OK. One thing we might as well do is bound the number of vertices and edges in our graph when we do that. So first of all, let's think about the number of vertices. And we can be totally lazy and upper bound this stuff. It doesn't matter.

Well, for one thing, I have one node per router. So we incur one factor of r there. And now, notice that we're kind of laying down cable one little piece at a time here in our chains. And now I always tend to have a fencepost-style headache about exactly what the constant factor is here.

But if we're conservative about it, we incur at most a factor of 100r kind of additional edges, because those are all the different pieces that we could lay together. I think it's actually less than that because of the endpoints, but whatever, because r plus 100 r is big O of r. So the number of vertices in my graph here is big O of r.

Similarly, what's the number of edges? Well, this is exactly the amount of cable that's inside of my network, I believe. Yep. So this is exactly 100r. Well, I guess the way the problem is written, it's upper bounded by 100r, but whatever. So this, again, is big O of r. This is kind of convenient. So now we have one number that rules them all, which is r, which tells you both the number of vertices and the number of edges, up to a constant factor, right?

So one thing I can convince myself is if I do BFS on my graph, that's sort of OK. Remember, that's vertices plus edges time. But in this case, those are the same. OK, so right. So remember, at the end of the day, I'm trying to compute the latency. This is like the length of the shortest path to the entry point nodes.

So here would be a braindead algorithm, which is to say, for all routers, for all entry points, compute-- I don't know. Let's call the router on i, the entry point j. I compute distance ij like using breadth-first search or something. And then I take the min of these values and add them all together, right?

So I compute-- for every router, I look at every possible entry point. I compute its distance to the entry point. I take the min over all these things. And now I add that to my running sum. There's a problem here, which is, I haven't told you the relative number of entry points to the total number of routers. So at least the way that I've written this algorithm here, how much time would this take?

Well, there's two different for loops. And in the worst possible case, at least in my braindead algorithm, I don't notice that if I am an entry point, then I don't need to compute distances. Well, this would take order r squared-- whoa, r squared time, right? At least, right?

Actually, I shouldn't even write big O. I should write-- what's lower bound? Oh god, I'm a terrible algorithms professor. Omega of r squared time, because I haven't even accounted for the amount of time that it takes to compute the distance, right? And this is a problem, because I've only given you a budget of linear time for your algorithm, right? So this is frowny face. I tried drawing the turd emoji on my notes. And it really-- it didn't work.

OK, so we need a better trick. And this is actually one of these prototypical tricks, which is to do the following. So let's construct a graph. I'm going to draw my graph in a particular way. But notice that there's nothing about my algorithm that cares about the way that I drew it. This is just to make my life easier, which is, I'm going to put all the entry points on the left and all the remaining non-entry-point routers on the right, because I can.

And so this is what my graph looks like. So these are like my entry points. Here are my other routers. My graph doesn't have to be bipartite. Like, it could be that my routers are connected to each other, whatever. And then there are some edges that go from my entry points to the routers in the graph. I'm trying to make sure that my graph is connected.

OK. And so essentially, what this problem is asking you to do is to say, OK, for every single node in my graph, I need to compute the distance to the closest entry point and then sum all those things together, right? That's just the schematic we could have in mind.

So in some sense, what we want to do is think about the set of entry points as like one giant node, because it doesn't matter which of these guys I choose for my shortest path to an entry point. I just need to find one, yeah? And so here's the basic trick. And this is one that appears all over graph theory, which is I'm going to introduce one additional node to my graph. And I'm going to put him on the left-hand side.

He's really big, because he is a supernode, which is a term of art. This term shows up a lot. And I'm going to connect it to every entry point in my network of routers. Does that makes sense, class? OK. So here's the kind of cool thing. So first of all, for every entry point, what's the shortest path from the entry point to the supernode? Well, obviously, it has length 1, right? I drew it for you here.

Now, here's the thing. Let's take the shortest path from the supernode to any of the routers on the right-hand side. What do I know? Well, clearly-- like maybe I choose this guy here. Well, what is my shortest path? It goes here and then there.

There's one property that matters here, which is that it has to pass through one of these entry nodes. Which one does it have to pass through? Shrug. For shame. Well, remember, Justin's favorite inequality is the triangle inequality.

And what does it say? It says that if I compute the shortest path from the supernode to any node in my graph, then every sort of sub-piece of that shortest path is also a shortest path. That sentence was hard to parse. Let's try that again.

So in particular, if I have a graph from the supernode to some router over here, well, we've convinced ourselves it has to pass through one of the entry nodes. Which one does that have to pass through? Is it ever something that is farther than the closest entry node? Well, no, because I could compute a shorter path in that case by choosing the closest entry node and then going to the supernode.

So this is a complicated way of saying that essentially, what we really want is for every router, the distance from that router, let's call it i, to the supernode, s. Is that quite right? Is that the distance to the closest entry point?

AUDIENCE: You went one more inch too far.

JUSTIN SOLOMON: I went one inch too far, right? Because I went to the closest entry point. And then I took an additional edge. So we want to do minus 1. OK. So what does this mean? Well, that means that I don't actually have to have this inner for loop over all the possible entry points. I just need to construct this new special graph with one additional node-- notice that's not going to affect my runtime-- and compute the shortest distance from the supernode to every other node in my graph, and then use that as my output, yeah?

So in other words, what is my algorithm going to look like? Well, first, I'm going to construct my graph, right? So what do I need to do? If I were to write this out in my homework, I would have to talk about how I've got these chains of edges between different pairs of routers. In addition to that, I'm going to make one additional supernode and insert an edge from that to every entry point.

Notice that adding the entry point here just adds a 1 to the number of vertices, and at most, I guess, an r to the number of edges, which doesn't affect asymptotically the size of either of these two sets. So that's a good thing. Now I'm going to do-- I'm going to use BFS to do a single-source shortest path from my supernode to all other vertices.

And how much time does this take? Well, remember that in general, BFS takes v plus E time. In this case, v plus E are both-- look like r. So this is order r time. OK. And then finally, I'm going to sum over routers i the value of the distance from the supernode to the router i, minus 1 to account for that additional edge that I added. OK, and that's the solution to our problem.

OK, any questions about number 2 here? Excellent. Go team. OK. So now let's move on to problem 3. Am I-- yeah, we're about halfway. OK, so in problem 3-- right. So we're doing Potry Harter and three wizard friends. The number three here, I believe, is actually irrelevant, although like any time you see a specific number in a problem, you should cache that in your bag of things to remember. And in this case, that was a red herring.

Potry Harter and her three wizard friends are tasked with searching around a labyrinth, yeah? And in particular, there's some nice things to know about the labyrinth and Potry Harter world-- this is really throwing off my dyslexia here-- which is the following. Right, so what do we know? We know that there are n rooms in my labyrinth and that each of my rooms has at most four doors.

So in other words, if I think of building a graph out of my rooms, which is like, I don't think I'm giving much away about this problem by jumping to the solution a little bit, what do we know about the degree of any vertex, assuming my vertices are rooms in the labyrinth? It's at most four. So that's kind of nice.

OK, right. And all the doors start closed. So that seems like a useful piece of information to remember. But we have this kind of weird thing, which is that some doors are enchanted. And apparently, Potry Harter can open up certain doors for free, which are not the intended doors. And then other ones, they have to do the blessing, and the holy water, and whatever it is that happens in this universe, and then opens up that door. But that costs them materials and heartache, right? And so we want to minimize that.

And so what they're given is basically a map. And this includes all of the different rooms, how they're connected to one another, and which of the doors are enchanted. And what I want is the minimum number of doors that they have to disenchant.

Now, this problem is like kind of sneaky. And the reason why is that there's like the network that's obvious to build. And that turns out to be not quite the right one. And then you can start thinking about adding weights on your graph and going crazy with that. But that turns out not to be the right direction.

And in fact, in Potry Harter world, apparently, we're not worried about their physical fitness. In other words, shortest paths are actually irrelevant in this problem. Do you see that? Because let's say that I have a really complicated, annoying problem. So like, maybe I have-- here's my labyrinth.

And we don't even talk about the entry point, like where they actually go in. But just for fiction purposes, let's say that they enter my labyrinth here and that, just to be annoying, the two doors that are enchanted-- remember, we could make a graph where all the vertices are rooms, and the edges are doors-- are like at these two endpoints of the T. So I have a giant T. And I enter right in the middle.

Now, what is Potry Harter to do here? Well, obviously, there's-- since this graph is a tree, there's only so much they can do, right? Maybe they enter here. They walk over all the way to the end to disenchant the door over here. And then they turn around and walk to the other end. They disenchant that guy, yeah? And now they can reach other rooms. Yeah, because that's their goal, is to visit every room. Sorry, I think I skipped that step.

Now, there's a few things to notice about this example, which make it a little bit different from the typical graph theory thing, which is, once they disenchant this door, like they walk over here, and they open it, well, now they walk over to this other room, just to-- you know those gym exercises where you run to the other side of the room, you touch the floor, and then you run back?

That's kind of what they did here, right? They ran to this room. They tapped that vertex. And now they want to turn around and walk to the other side. They don't pay money again on their way out, right? So once they open that door, it stays open. And that's actually quite important, because what it does is it makes this problem not look like a traveling salesman problem, which wouldn't be so great.

OK, so right. And moreover, does the fact that they're-- like, maybe I subdivide these edges. I have a bunch of edges here that are all not enchanted. Does that matter, like if I had like five billion edges here? No, right? Because they only ask in this problem for the minimum number of doors that you have to disenchant, yeah?

So it might be the Harry Harter walks really far along my graph. But as long as they don't walk through an enchanted door, it costs them nothing. So what does that mean? Well, that means that in some sense, the second that I enter a room, I might as well walk to every other room that it's connected to through unenchanted doors. And that doesn't cost me anything.

So sort of as a policy, I should do that, right? I enter a room. And then I kind of just search around and enter every possible door that I can that doesn't cost me an enchantment, because those are free. And my goal is to visit every room, yeah?

OK, so here's going to be the sneaky trick. Like, what is that starting to smell like? I open a door, and now I want to explore all the other rooms that are connected to that one.

AUDIENCE: Maybe a connected component.

JUSTIN SOLOMON: Yeah, maybe a connected component. There's a problem. Is it connected component in this graph? Well, no. Like, this whole graph is one giant connected component. So the sneaky trick is we're actually going to remove the enchanted doors. That was supposed to erase, and it didn't happen.

But the point is that if we remove the connected doors, these are like the chunks of my map that I can visit without incurring any cost. So if I think of my graph, maybe there's a bunch of vertices over here. And then there's an enchanted door. And there's a bunch of vertices over here, and then like two more enchanted doors like that. And like, what goes on in here, like if this is like a giant triangle or something, is actually irrelevant, because once I touch any one of these, I can now touch all the rest of them.

So let's suggest an algorithm. So our first step is that we construct a graph, G, where the nodes are the rooms-- are the rooms. And what should the edges be? Well, if I'm just trying to find these little clumps of rooms that I can visit for free if I get to any one of them, then the edges are the non-enchanted doors.

OK? And so now, in step two, I'm going to compute my connected components, which we covered in lecture-- the connected components of my graph, G. How much time does that take? Well, remember that there are two different algorithms we mentioned that can do this. This is full BFS or DFS. And both of them are going to take the same amount of time. What is that time?

AUDIENCE: Linear in the size of the graph.

JUSTIN SOLOMON: Linear in the size of the graph. So initially, that could be problematic, because I want order n. Remember, there are n rooms here. But thanks to our degree bound, thanks to knowing that every room has at most four doors, you can convince yourself that both the number of vertices and the number of edges are order n, which I should probably rush through, because as usual, I'm going slowly.

OK. So now, what do I have? I have a list of all the connected components in my graph. And each one is potentially connected to some other ones by enchanted doors. So in some sense, I could think about-- it's not to say this is the solution to the problem. But I could think about modeling my problem as making some new graph, where I put a giant vertex in every connected component. And maybe I connect them by enchanted doors. And I want a path that touches every one of these rooms.

But that's not quite the right way to go. And this is what catches you by surprise, because this starts something scary, right? If you've heard of the traveling salesman problem, it kind of smells like that. But that's not actually correct here for two reasons.

One is that once I open an enchanted door, I can go back through it. Like, I can like hopscotch back and forth through that door as many times as I want, and it doesn't cost me anything. It only costs me something the first time I open it, yeah? And moreover, I didn't ask you to actually compute me that path. If you read the problem closely, it just asks for the minimum number of doors you have to open.

So this is a really sneaky problem, because it turns out there's an additional one line of code that solves this problem. That's step three, which I'm going to write before steps one and two, just to keep you confused. And that is to return this number of connected components minus 1.

That seems sneaky. Why is that? Well, what's going on here is the following, which is that let's say that I walk along-- remember that my graph is connected. So what I know is that I can always get from any one connected component to any other.

And so let's just take whatever order-- notice that the problem hasn't actually asked me how to return an efficient path. It just asked me for the minimum number of doors I have to open. So all I have to do is convince myself there exists a path with this many doors I have to open. I don't have to actually return it. If I did, it would be mildly more annoying to think about.

OK. So my graph is connected. So one thing that I could do is make the world's-- well, how do I want to do this? Well, let's see here. I guess I could come up with an ordering that looks like depth-first search of my graph. That should do it.

OK. So maybe I start at this guy. I just start at some arbitrary vertex. And then I'm going to do depth-first search, but rather than on the full graph, on this kind of meta graph, where I've clumped together rooms that I can get to with no cost.

So what am I going to do? I'm going to start walking outward toward this guy and then a depth-first search, backtracking, and then going back down. And if you think about it, remember that in depth-first search, I have this property, I never need to revisit a clump which I've got to it once.

Well, the total number of doors that I'm going to open is exactly the number of connected components minus 1, because as soon as I've done that, my depth-first search is done here, yeah? In other words, that's the number of nodes in my graph. So if I took-- what would be a better way to-- I'm noticing that in my head, this was easier to articulate than in words. Here would be a way to do it, would be--

AUDIENCE: Maybe add some more enchanted doors to the graph?

JUSTIN SOLOMON: Maybe add some more enchanted doors. Ah, that's true. Actually, my problem's a little too easy. So as long as my depth-first search backtracks along the paths it's already found, then I'm sort of reaching out into this tentacle, and then reaching back, and then reaching to a new place. I'll never traverse an enchanted door that I don't need to, because I've already seen the location.

AUDIENCE: So you're traversing a tree, basically?

JUSTIN SOLOMON: Yeah. So I've got a shortest path tree that's going on here. Actually, I guess a breadth-first search would be a better example. In fact, here's-- OK, let's be concrete about it. I'm sorry. I should have thought about this more carefully than I did at home yesterday.

One thing I could do would be to compute a shortest path tree from one vertex in this graph to all the other ones. In particular, that gives me the shortest path. And I could traverse that tree to one node, and then traverse it all the way back, and then reverse it to a new node, and then traverse it all the way back, and so on.

This is not an efficient path from a walking perspective. But from a door opening perspective, it's extremely efficient, because it's a tree, right? And remember that the number of edges in a spanning tree of my graph is exactly the number of vertices in my graph minus 1, which is exactly the property we have here. Whoo, sweating for a second there.

OK. So now, in our remaining 30 minutes here, we've got two more problems, which is more than enough time, especially because the last problem is largely combinatorial and less algorithmic. So I think it's OK to focus-- maybe talk about that at a high level and show a fun plot.

OK, so for problem 4, we have an airline, Purity Atlantic. That's cute, Jason, really. And it's owned by Brichard Ranson. Did I get that right? And Purity Atlantic has a cute sale-- this is not like a cute angle, I suppose-- which is essentially the following, which is that you can book an itinerary where you have your home city. And then you choose, I believe, three other cities that you want to visit.

And then Purity Atlantic-- maybe you're on your honeymoon, and you're not concerned with price, but rather, just the efficiency, because you don't want to spend your whole time in an airplane. That's particularly true this month. Then what do you want to do? You want to minimize your total number of connections, right? Because as we all know, in spring 2020, we don't want to spend very much time in airports, yeah?

So, right. So how do we do that? Well, we make a website, where you tell Purity Atlantic the cities that you want to visit and your home city. And they give you back an efficient itinerary that minimizes the number of connections. OK. And the question is, how do you actually do that, right? How do you compute the best itinerary that minimizes the number of flights you have to take?

So what are our variables? And sometimes it feels like the variables are all the different permutations of the cities you could visit, right? I could go to, I don't know, Cambridge, Boston, and then Cambridge in the UK. Maybe you're doing like a University thing, and then, I don't know, Budapest and some other place. Or I could do those in any other order. And that feels like it should be factorial, which would be bad news.

But this is one of these problems which I suppose a computer science theorist might call fixed parameter tractable. But that's sort of an overkill term here. But essentially, as long as you ignore all the factors that make this problem hard, then it's easy.

A different way to put it is that, OK, if I'm only visiting three cities, what's the total number of possible orderings of my three cities? Class? So I have city A, B, C. I could do B, C, A. I could do B, A, C.

AUDIENCE: List them out.

JUSTIN SOLOMON: Yeah, fine, Jason. I'll do that. So this is what we call direct proof mathematically, which are other possible ways to visit three cities. And now, by my direct proof, I claim there are no other ways to visit three cities. And in particular, there are 1, 2, 3, 4, 5, 6 different orderings of the cities that I can visit.

Notice that this is a constant in my problem. I am not asking you to make a website that takes like the total set of cities that you want to visit as a couple and order them. It's specifically three. You might also notice that 6 is 3 factorial, which is perhaps a more efficient way to get to that same bound.

OK. So, right. So there's six different orderings of the cities. And in each case, what am I going to have to do? I'm going to have to compute the sum of going from my source city to the first one, from the first one to the second one, from the second one to the third one, from the third one back to the first one, OK?

So what do I need? Well, I need-- in some sense, I want to be conservative about it, just the cost of flying from every city to every other city. But that's not quite right. I only need the cost of flying from every city that you have specified as a city you're interested in to every other city that you've specified that you're interested in, yeah? OK.

So in particular, I go to my new one. So in this problem, we have c cities and f flights. OK. And initially, it might seem that we have to compute a ton of shortest paths. But like, if I want to go from Boston, to Budapest, to London, to-- I'm running out of cities-- Paris, and back to Boston, or whatever ordering I prefer, do I need to worry about the shortest path from Nebraska to California? Potentially not. Like, that could be irrelevant. The only ones that I care about are those four cities that I've identified, OK?

So there's 3 factorial possible permutations. And at the end of the day, well, there's 2 times 4 choose 2. If you're wondering, this is 12, or big O of 1 pairs of cities, meaning that, like-- for itinerary purposes-- itinerary-- meaning that if I always enter an airport in one of city A, B, C, or my hometown, and I always exit through another one, so then there's four possible cities. I choose them two at a time.

Notice that flights might not be ordered. Like, I might be able to get from one city to another. But then maybe the airplane has a connection or something. So going back is a different cost. But totally, there's 2 times 4 choose 2 different pairs of cities that I could enter or exit from.

OK. So now, what am I going to do? Well, so I can compute the 12 different shortest paths that matter in my graph. So when I say shortest path, what do I mean? Well, I'm going to construct a graph, G, with one vertex per city and one edge per flight.

And notice the number of connections that I need to make, the minimum number between any city than any other city, is equal to the shortest path-- the length of the shortest path minus 1, right? So like, maybe I have-- like, here's Boston. Here's London. Here's Paris, B, L, P for short. Then the length of my shortest path is 2. And the number of connections I have to make is 1, because I stop through London, yeah?

So what am I going to do? Well, for every pair of cities in this-- oops-- no, that's OK-- in the set of the source city and the three cities you want to visit, I'm going to compute the shortest-- the length of the shortest path. So this is the minimum number of connections I need to get from any one of these to any other one in my graph, G.

Well, how much time does this take? Well, there's 12 such pairs. We already argued that. And how much time does it take to actually do shortest path, so using breadth-first search?

AUDIENCE: Linear time the size of the graph.

JUSTIN SOLOMON: Linear time the size of the graph. I think Jason actually has a t-shirt that says that. Well, in this case, remember, that's big O of the number of edges plus the number of vertices. But just to make your life a little more annoying, the number of vertices is the number of cities. And the number of edges is the number of flights, right? So this takes 12 times O of c plus f time, which, of course, is just O of c plus f time.

Notice, this is one of these things where we're being sneaky. We told you that you specifically visit three places. And that's where this number 12 came from. If we'd said that you wanted to visit m cities, then this would be a very different homework problem. This is one of those things you got to remember, where we've given you a few constants, and you should use them.

OK. So now, what can I do? I can iterate over every permutation of A, B, C, right? So this is like saying I go from my source to city 1, to city 2, to city 3, back to my source. I add together and compute the cost, the cost of that trip.

And remember, cost in this case is equal to the minimum number of connections. And then I return the minimizer. So I say, like, is it cheaper for me to go Boston, Budapest, Paris, Boston, Paris, Budapest, and so on. So a for loop over permutations, which generally is frowned upon. But in this case, because we told you you're visiting precisely three places, how many steps are going to happen in that for loop?

Well, we actually wrote them all out over here on our board. It's exactly 3 factorial, or 6 steps, right? 3 times 2 times 1, which is 6. OK. So, right. So at the end of the day, this for loop is going to take, well, order 6 time, which, of course, is just order 1. So it doesn't really contribute to our runtime at all. And our entire algorithm runs in c plus f time.

OK, so right. So this is one of these problems where you're really taking advantage of the constants that we gave you. We said you're visiting three cities. So use it. Incidentally, as a computer science theorist, if I said you're visiting exactly 17 cities, well, what would be our numbers now? I mean, it would be 17 factorial and then like 17 choose 2. Those are big numbers. But they're still constants. So for purposes of this class, that would be OK. But the second that I give it a name, like m, then I got to think about those factorial things a little more carefully.

All right. So that's this problem. So the basic trick here was that, like, yeah, it looks like all pairs shortest path. But it's not quite. It's all pairs of things that you're actually going to travel between shortest path. And since that number of pairs is finite-- it's just 12-- that's an OK thing to do.

OK. How we doing? Ah, 15 minutes. Perfect. I didn't want to do the last problem. And I think I've managed to get myself in exactly that position. OK. So the very last problem on this homework, which, again, this homework really follows the prototypical 6.006 breadth-first search, depth-first search homework. I feel like they all fall into a similar pattern.

Again, all these resources are available to you guys. You should look at them. We're not trying to hide anything. This problem involves solving a pocket cube, which is like a little mini Rubik's cube, which is 2 by 2. And it looks like this. Ah, there's chalk. Actually, there we go.

So here's my Rubik's cube. Looks like a cube, which I'm having some trouble drawing. And in particular, it's 2 by 2, which makes it a little easier than your typical Rubik's cube. And in particular, we're going to mark some faces. Sneakily, they used a little geometry term here, which is cute.

So here's face f0. I'm sorry you can't quite see that. But the top face is f0, in case you were wondering. The left face is f1. And the front-facing face here, f2. Notice that we've identified these by like vectors that point 90 degrees out from the face. These are called normal vectors. If you want to define those rigorously, you can take my grad level class. But for a Rubik's cube, it's not terribly difficult.

But in any event, I can talk about flipping this Rubik's cube in a pretty easy way, which is that I like, I'm going to fix one corner of my cube. So this is like the corner that I'm holding onto with my hand. And now I can grab, what? The top, the side, or the front of my cube. And I can rotate it clockwise or counterclockwise. And you can convince yourself those are all the possible ways that I could sort of mess with the state of my cube after fixing one corner.

OK, right. And so this problem basically is involving sort of a very typical trick. In fact, a lot of the history of these different search algorithms-- breadth-first search, depth-first search, A*, which I guess we won't really cover here-- date back to, what, 20 or 30 years ago, we would have called artificial intelligence. These days, that has a very different meaning.

But back in the day, AI was all about solving board games, and Rubik's cubes, and all these kinds of things, using algorithms. And the way that they would do that is by searching the different spaces of configurations. And so now, if we think of every face of this cube as painted with a color, there are different configurations of my graph that I get by flipping the three sides.

So if we think of there being a vertex for each state of my cube, where state here means like the coloring of every face on my Rubik's cube, then there's an edge for every move. And in this problem, we encoded a move as a pair, j comma s, where it's saying that I'm going to rotate face fj, where j is between, I guess, 0, 1, 2, in direction s. And we can just index that as like plus or minus 1 to kind of say counterclockwise or clockwise.

So this is kind of a cute thing, where your graph has a bunch of vertices, which are all Rubik's cubes. That's a cube. And then there are edges if I can get from one to another by doing one of these moves. And this is a nice abstraction, because if I want to solve a Rubik's cube in the most efficient way possible, one way to do that is to compute the shortest path from my current configuration to the Platonic Rubik's cube, where all the colors are constant on the different faces of my cube.

And so that's like a sort of basic identification that happens all over the place in search strategies, where I'm going to think of every vertex of my graph as being the state of some system and every edge as being a transition from one to another. And then paths in this thing are kind of like different ways of solving my puzzle, right?

So like a different one would be, I don't know, every vertex is a chess board with the chess pieces scattered on the chess board. And every edge is one chess move by one player or the other. In that case, you'd have to be a little careful, because you want player 1 or player 2 to go back and forth from each other. But I'll let you think about the reduction there.

OK, so right. This problem, I think largely, is mostly just fun combinatorics rather than algorithms. But there's a little bit of algorithms hiding in here. So they want you to argue that the number of distinct configurations of this Rubik's cube, this 2-by-2 guy, is less than 12 million. This is nice, because 12 million is a number that computers can actually cope with.

And so there's a pretty straightforward argument there. Right. So in particular, here's a cube. How many quarters are in a cube?

AUDIENCE: Eight.

JUSTIN SOLOMON: Eight, thanks. I hid one back here, in case you were wondering. So let's say that I fix a corner of the cube, like we've done that. Then every time that I rotate one of the faces of my cube clockwise or counterclockwise, I'm essentially like taking one corner of my cube and like sticking it in another place, right?

So in all, seven corners of my cube can move. And if I'm not worried about, like-- it could be that some of these permutations are not actually achievable by a set of steps. Like, maybe I'd have to break my Rubik's cube and glue it back together. But if I'm being conservative about it, there's of course less than or equal to 7 factorial different configurations of the corners.

So in other words, every time I rotate my face, one of the corners ends up in a different place. So there's 7 factorial different ways that could have happened. OK, so that's part of my bound. Remember that I'm trying to bound the total number of configurations here.

And essentially, what I've done so far is I've said, OK, well, there's a bunch of cubes in my 2-by-2 Rubik's cube. So I'm going to like unglue this entire cube, take just this corner, and stick it up here. And there's like 7 factorial different ways that I could do that.

But I still have to account for the fact that I pull this piece off. I stick it in the top. But I have to figure out its orientation. I can still rotate it about this corner. And in fact, there are three different ways that I could rotate it, right? You can kind of see it, right? 1, 2, 3, yeah?

So in all, so each corner can rotate three ways. So that means that I have 3 times 7 factorial different configurations as an upper bound. And this number is, wait for it, 11,022,480. The problem asks you to argue that your upper bound is upper bounded by 12 million. And indeed, it is less than or equal to 12 million.

AUDIENCE: Is that 3 times 7 factorial or--

JUSTIN SOLOMON: Oh, I'm sorry. Right, because there are seven corners, each of which can rotate three different ways. It's actually 3 to the 7th power times 7 factorial. Thank you, student. OK, right. So let's see here. Really quickly moving here, the next problem says, state the maximum and minimum degree of any vertex in my graph.

First of all, do I expect vertices to have different degrees? This is kind of a goofy problem. Like, what would it mean to have a vertex that somehow has lower degree than another vertex? It would mean that there's some configuration of this cube for which there are fewer moves that I could do to change it than a different configuration of this cube.

And that's obviously not the case, right? Because when I flip one of the faces of my cube, all I'm doing is I'm moving the colors around. I haven't somehow changed the physics of how a Rubik's cube works, right? And so I think this was just intended to be annoying by your instructors. The min degree is equal to the max degree. And in fact, the degree of every node in my graph is constant here.

The one thing that's worth noting here-- what I haven't argued-- it turns out, I think, to be true. But what I have argued is that I couldn't rotate a face and actually end up in the same configuration. Like, maybe, for some reason, I had red all the way around the outside. And so when I rotate it, nothing changed. That obviously isn't true. But I haven't argued it carefully.

But as long as I don't worry about my graph being simple, like I'm OK with self-loops, then the degree is certainly constant, yeah? OK. And in fact, I don't think that can happen in a typical Rubik's cube.

AUDIENCE: Well, I think the point is to say what the degree was.

JUSTIN SOLOMON: Oh, yeah, indeed. So we haven't computed the degree. But we've argued that they're equal to one another. OK, so now we have to compute what that degree is. And here's how to do it. So of course-- well, this, I think, is actually even easier than the first part.

Essentially, remember, we have three different options for faces that I can rotate. I can rotate the top, the front, or the side here. So there's three faces that we could rotate. OK, and how many different ways can rotate them? I can rotate them counterclockwise or clockwise. So there's two directions. So in all, there's degree 6 for every vertex, right? There's sis different ways in or out of a vertex here.

OK, so the next part of the problem gives you a piece of code and then does breadth-first search on this graph. And it's super, super slow to give me the distance to all the other configurations. And Jason conveniently has run it on his laptop here. I don't-- I'm nervous to touch your laptop. I don't care so much, but I don't-- you know, I don't want to infect your-- yeah. Right.

So we have a piece of code that explores the graph of all the configurations of our cube by breadth-first search and then sort of gives me the shortest path, I think from the base cube, where all the faces are constant to all the other configurations that are reachable, and generates a plot. Right. And so what they ask is to figure out the total number of configurations that it explores.

One thing that you'll find out-- center down-- is that it explores pretty much a third-- in fact, exactly a third of all the possible configurations of my cube. I think we can see that here. So I guess it runs this whole thing. You saw them all together. Right. Oops.

That's OK. And so in fact, the kind of fun fact that you can learn about the 2-by-2-by-2 Rubik's cube is that there's actually three connected components in this graph. So in other words, there's sort of like three different Rubik's cubes you can make, modulo all the different flips that you can do to the faces. And those correspond to there corner rotations of one of the corners of the thing.

OK, so right. So then the next part of the problem asks you to state the maximum number of moves needed to solve any Rubik's cube. And you can see it in this plot. So what this plot is showing you is the size of the level set of this distance function for every distance.

So I think technically, it looks like 0. But there's actually, this is at a 1 here, which is to say there's one vertex at distance 0, which is the source. And as we move farther and farther out, our tree is expanding. And we're seeing more and more vertices. And apparently, most vertices are approximately distance-- is that 11 or 12? 11 away from the original.

Then eventually I explore the entire graph, and I'm done. And you can see that the farthest-away vertex is 14 away, meaning that the most annoying Rubik's cube to solve can be solved in 14 steps for the 2-by-2-by-2 pocket cube. I'm sure that Jason probably knows the equivalent of this number for the 3-by-3-by-3. But I have no idea what it is. I'm impressed if he can calculate it in his head, like he looked like he was about to try. But I digress.

Right. So in other words, this is actually a fancy term for-- we talked about the radius of your graph in the first problem. Now we've got the diameter, which is, well, not necessarily 2 times the radius, the way that we've defined it here, but actually, almost-- I think within some constant of that.

OK, so right. So notice that the vertical axis here is really big. And this is explaining why this BFS code is so slow, right? Because these are all the different configurations it has to hit. Or more accurately, if I take the y-position of each one of these vertices and sum up its height, those are all the configurations that are reachable. And those are all the steps that BFS needs before it's done. And so that number is in the-- it's certainly in the millions, yeah.

OK, so then the last part of this problem, which it conveniently looks like I'm low on time to solve, but I'll refer you to the solution anyway, is asking how we might do this faster. And so in particular, what it says is, let's say that I have a total of n configurations for my Rubik's cube. In this case, it turns out that that's like roughly three million, I think.

OK. And now I want an algorithm that gives me the shortest sequence of moves to solve any pocket cube. Man, I'm really ravaging the chalk today. And I want to solve any cube in a number of steps that looks like 2N to the ceiling of w over 2, where-- let's see here. The code provided-- sorry, this problem changed on me this afternoon. [LAUGHS] Right.

So where N sub i is equal to the number of configurations reachable within i moves. Oh, good. I see what we did here. So if this is my base cube, then we've got like maybe, I guess, six different-- 1, 2, 3, 4, 5, 6 different cubes that I can reach from those. And then there's 6 cubes I can reach from all of those.

But of course, some of those might be pointing backward or to each other. But this is the number of things that are reachable in i moves. And they ask for an algorithm that finds the shortest path in this amount of time. By the way, big N typically exponentiates in that subscript there. This looks innocent, but it's not.

The basic trick here is to do-- I'm not going to bother writing it down. We'll just talk about it for a second and call it for the day. Well, I'll draw a picture. So the breadth-first search algorithm that we've thought about so far chooses a vertex and then computes level sets outward from that vertex until it maybe reaches the destination that you want to hit.

That doesn't quite work here. And the-- well, I mean, it does work. But it's going to be quite slow, because like, let's say I had bad luck, and now we're in that 14 vertex, right? Then, somewhere in there, I'm going to hit this big height, which is sitting over the 11, before I can get to vertex 14.

So the trick here is it turns out that I can do it by only ever getting to 7. And the way that I'm going to do that is instead, I'm going to run BFS sort of in parallel for two different vertices, right? The source and the target. So in this case, my current cube and the cube I would like it to be, like the solution to the problem.

I'm first going to compute the level set 1 of that cube, then level set 1 of the next cube, then level set 2, level set 2, 3, 4. And notice that eventually, they're going to intersect, pretty much right at the midpoint. And so the size of the level set-- I never need to compute a level set that's bigger than a half of the shortest path length. I have to round up to be conservative about that. And that's where I get this factor here.

So that's just a nice little trick for reducing the search size. This is another kind of standard trick if you look at some of the code people use for solving board games algorithmically and so on. I think they typically sort of search from the beginning and end state outward and try and meet in the middle for exactly this reason, which is that exponential growth, as we all know, can be quite problematic.

All right, folks. So I think we're just about out of time. And I've certainly worn myself out. So with that, hopefully we'll see you next week. And yeah, I hope everybody is doing well.