PROFESSOR: Today we're going to introduce graph search in general and talk about one algorithm, which is breadth-first search, and understand how in principle you can solve a puzzle like the Rubik's Cube. So before I get to Rubik's Cubes let me remind you of some basic stuff about graphs. Or I can tell you to start out with, graph search is about exploring a graph. And there's many different notions of exploring a graph.

Maybe I give you some node in a graph, s, and some other node in a graph, t, and I'd like to find a path that's going to represent a problem like I give you a particular state of a Rubik's Cube and I want to know is there some path that gets me into a solved state? Do I really want to solve this on stage? What the hell? We started. So this is a particularly easy state to solve, which is why I set up this way. All right, so there you go. Seven by seven by seven Rubik's Cube solved in 10 seconds. Amazing. New world record.

So you're given some initial state of the Rubik's Cube. You're given the targets that you know what solved looks like. You want to find this path. Maybe you want to find all paths from s. Maybe you just want to explore all the nodes in a graph you can reach from s. Maybe you want to explore all the nodes in a graph or maybe all the edges in a graph. These are all exploration problems. They're all going to be solved by algorithms from this class and next class.

So before we go further though, I should remind you what a graph is and sort of basic features of graphs that we're going to be using. This is also 6042 material so you should know it very well. If you don't, there's an appendix in the textbook about it. We have a set of vertices. We have a set of edges. Edges are either unordered pairs-- some sets of two items-- or ordered pairs. In this case, we call the graph

undirected. In this case, we call the graph directed. Usually, there's only one type. Either all the edges are directed or all the edges are undirected. There is a study of graphs that have both, but we are not doing that here.

Some simple examples. Here is a graph. This is an undirected graph. This is a directed graph. The set of vertices here is a, b, c, d. The set of vertices here is a, b, c. The set of edges here is-- E is going to be things like a, b; b, c; c, d-- I think you get the idea. Just for completeness, V is a, b, c, d. Just so you remember notations and so on.

One of the issues we're going to talk about in this class is how do you represent a graph like this for an algorithm? So it's all fine to say, oh, this is a set of things. This is a set of things. An obvious representation is, you have a list or an array of vertices. You have an array of edges. Each edge knows its two end points. That would be a horrible representation for a graph because if you're, I don't know, at vertex, a, and you want to know, well what are the neighbors of a? b and c. You'd have to go through the entire edge list to figure out the neighbors of a. So it's been linear time just to know where you can go from a. So we're not going to use that representation. We're going to use some better representations. Something called an adjacency list.

Over here, you've got things like a, c; b, c; and c, b. So you can have edges in both directions. What am I missing? b, a. So that's E, in that case. There are a whole lot of applications of graph search. I'll make you a little list to talk about few of them.

So we've got web crawling. You're Google. You want to find all the pages on the web. Most people don't just tell you, hey, I've got a new page, please index it. You have to just keep following links-- in the early days of the web, this was a big deal-- following links finding everything that's out there. It's a little bit of an issue because if you define it wrong, the internet is infinite because of all those dynamically generated pages. But to deal with that, Google goes sort of breadth-first for the most part. It's prioritized You want to see all the things you can reach from pages you already have and keep going. At some point, you give up when you run out of

time.

Social networking. You're on Facebook. You use Friend Finder. It tries to find the friends that are nearest to you. Or friends of friends is sort of a level to search. That's essentially a graph search problem. You want to know what's two levels or three levels of separation from you. And then you loop over those and look for other signs that you might be good friends.

You are on a network like the internet or some intranet. You want to broadcast a message. So here's you. You want to send data out. That's essentially a graph exploration problem. That message, that packet, is going to explore the graph.

Garbage collection. I hope you all know that modern languages have garbage collection. This is why you don't have to worry about freeing things. Even in Python-- even in CPython, I learned-- there is a garbage collector as of version two. But also in PyPy, and JPython and in Java-- pretty much every fairly modern language you have garbage collection. Meaning, if there's some data that's unreachable from-- So you have your variables. Variables that can be accessed by the program. Everything that's reachable from there you have to keep. But if some data structure becomes no longer reachable, you can throw it away and regain memory. So that's happening behind the scenes all the time, and the way it's being done is with their breadth-first search, which is what we're going to talk about today.

Another one. Model checking. Model checking is-- you have some finite model of either a piece of code, or a circuit, or chip, whatever, and you want to prove that it actually does what you think it does. And so you've drawn a graph. The graph is all the possible states that your circuit or your computer program could reach, or that it could possibly have. You start in some initial state, and you want to know among all the states that you can reach, does it have some property. And so you need to visit all the vertices that are reachable from a particular place. And usually people do that using breadth-first search.

I use breadth-first search a lot, myself, to check mathematical conjectures. So if you're a mathematician, and you think something is true. Like maybe-- It's hard to

give an example of that. But you can imagine some graph of all the possible inputs to that theorem, and you need to check them for every possible input-- If this is true-- the typical way to do that is breadth-first searching through that entire graph of states.

Usually, we're testing finite, special cases of a general conjecture, but if we find a counter-example, we're done. Don't have to work on it anymore. If we don't find a counter-example, usually then we have to do the mathematics. It doesn't solve everything, but it's helpful.

And then, the fun thing we're going to talk about a little bit today, is if you want to solve something like a two by two by two Rubik's Cube optimally, you can do that using breadth-first search. And you're going to do that on your problem set. To do it solving this one optimally using breadth-first search would probably-- would definitely-- take more than the lifetime of the universe. So don't try seven by seven by seven. Leave that to the cubing experts, I guess. I think no one will ever solve a seven by seven by seven Rubik's Cube optimally. There are ways to find a solution just not the best one.

So let me tell you just for fun, as an example. This Pocket Cube, which is a two by two by two Rubik's Cube. What we have in mind is called the configuration graph or sometimes configuration space. But it's a graph, so we'll call it a graph. This graph has a vertex for each possible state of the cube. So this is a state. This is a state. This is a state. This is a state. Now I'm hopelessly lost. Anyone want to work on this? Bored? No one? Alright, I'll leave it unsolved then. So all those are vertices. There's actually a lot of vertices. There are 264 million vertices or so. If you want. To the side here. Number of vertices is something like 8 factorial times 3 to the 8. And one way to see that is to draw a two by two by two Rubik's Cube.

So these are what you might call cubelets, or cubies I think is the standard term in Rubik's Cube land. There's eight of them in a two by two by two. Two cubed. You can essentially permute those cubies within the cube however you like. That's 8 factorial. And then each of them has three possible twists. It could be like this. It

could be like this. Or it could be like this. So you've got three for each. And this is actually an accurate count. You're not over-counting the number of configurations. All of those are, at least in principle, conceivable. If you take apart the cube, you can reassemble it in each of those states. And that number is about 264 million. Which is not so bad for computers. You could search that.

Life is a little bit easier. You get to divide by 24 because there's 24 symmetries of the cube. Eight times three. You can divide by three, also, because only a third of the configuration space is actually reachable. If you're not allowed to take the parts apart, if you have to get there by a motion, you can only get to 1/3 of the two by two by two. So it's a little bit smaller than that, if you're actually doing a breadth-first search, which is what you're going to be doing on your problem set. But in any case, it's feasible.

That was vertices. We should talk about edges. For every move-- every move takes you from one configuration to another. You could traverse it in one direction and make that move. You could also undo that move. Because every move is undoable in a Rubik's Cube, this graph is undirected. Or you can think of it as every edge works in both directions. So this is a move. It's called a quarter twist. This is a controversy if you will. Some people allow a whole half twist as a single move. Whether you define that as a single move or a double move is not that big a deal. It just changes some of the answers. But you're still exploring essentially the same graph.

So that's the graph and you'd like to know some properties about it. So let me draw a picture of the graph. I'm not going to draw all 264 million vertices. But in particular, there's the solved state-- we kind of care about that one, where all the colors are aligned-- then there's all of the configurations you could reach by one move. So these are the possible moves from the solved state. And then from those configurations, there's more places you can go. Maybe there's multiple ways to get to the same node. But these would be all the configurations you can reach in two moves. And so on.

And at some point, you run out of graph. So there might be a few nodes out here. The way I'm drawing this, this is everything you can reach in one move, in two movies, in three moves. At the end, this would be 11 moves, if you allow half twists. And as puzzlers, we're particularly interested in this number, which you would call, as a graph theorist, the diameter of the graph. Puzzlers call it God's number. If you were God or some omni-- something being. You have the optimal algorithm for solving the Rubik's Cube. How many moves do you need If you always follow the best path? And the answer is, in the worst case, 11. So we're interested in the worst case of the best algorithm.

For two by two by two, the answer is 11. For three by three by three, the answer is 20. That was just proved last summer with a couple years of computer time. For four by four by four-- I don't have one here-- I think we'll never know the answer. For five by five by five, we'll never know the answer. For six, for seven, same deal. But for two by two by two, you can compute it. You will compute it on your problem set. And it's kind of nice to know because it says whatever configuration I'm in, I can solve it in 11 moves.

But the best known way to compute it, is basically to construct this graph one layer at a time until you're done. And then you know what the diameter is. The trouble is, in between here this grows exponentially. At some point, it decreases a little bit. But getting over that exponential hump is really hard. And for three by three by three, they used a lot of tricks to speed up the algorithm, but in the end it's essentially a breadth-first search. What's a breadth-first search? This going layer by layer. So we're going to formalize that in a moment. But that is the problem.

So just for fun, any guesses what the right answer is for an n by n by n Rubik's cube? What's the diameter? Not an exact answer, because I think we'll never know the exact answer. But if I want theta something, what do you think the something is? How many people here have solved the Rubik's Cube? Ever? So you know what we're talking about here. Most people have worked on it. To think about an n by n by n Rubik's Cube, each side has area n squared. So total surface area is 6 n squared. So there's, roughly, stata n squared little cubies here. So what do you

think the right [INAUDIBLE] is for n by n by n? No guesses?

AUDIENCE: n cubed?

PROFESSOR: n cubed? Reasonable guess. But wrong. It's an upper bounds. Why n cubed?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Oh, you're guessing based on the numbers. Yeah. The numbers are misleading, unfortunately. It's the law of small numbers I guess. It doesn't really look right. I know the answer. I know the answer because we just wrote a paper with the answer. This is a new result. From this summer. But I'm curious. To me the obvious answer is n squared because there's about n squared cubies. And it's not so hard to show in a constant number moves you can solve a constant number of cubies. If you think about the general algorithms, like if you've ever looked up professor's cube and how to solve it, you're doing like 10 moves, and then maybe you swap two cubies which you can use to solve a couple of cubies in a constant number of moves.

So n squared would be the standard answer if you're following standard algorithms. But it turns out, you can do a little bit better. And the right answer is n squared divided by log n. I think it's cool. Hopefully, you guys can appreciate that. Not a lot of people can appreciate n squared divided by log n, but here in algorithms, we're all about n squared over log n. If you're interested, the paper's on my website. I think its called, Algorithms For Solving Rubik's Cubes.

There's a constant there. Current constant is not so good. Let's say it's in the millions.

[LAUGHTER]

You've got to start somewhere. The next open problem will be to improve that constant to something reasonable that maybe is close to 20. But we're far from that.

Let's talk about graph representation. Before we can talk about exporting a graph, we need to know what we're given as input. And there's basically one standard

representation and a bunch of variations of it. And they're called adjacency lists. So the idea with an adjacency list, is you have an array called Adj, for adjacency of size V. Each element in the array is a pointer to a linked list. And the idea is that this array is indexed by a vertex.

So we're imagining a world where we can index arrays by vertices. So maybe, you just label your vertices zero through v minus 1. Then that's a regular array. Or, if you want to get fancy, you can think of a vertex as an arbitrary hashable thing, and Adj is actually a hash table. And that's how you probably do it in Python. Maybe your vertices are objects, and this is just hashing based on the address of the object. But we're not going to worry about that. We're just going to write Adj of u. Assume that somehow you can get to the linked list corresponding to that vertex.

And the idea is, for every vertex we just store its neighbors, namely the vertices you can reach by one step from u. So I'm going to define that a little more formally. Adj of u is going to be the set of all vertices, V, such that u, v is an edge. So if I have a vertex like b, Adj of b is going to be both a and c because in one step there are outgoing edges from b to a and b to c. So Adj of b is a, c. In that graph. I should have labeled the vertices something different. Adj of a is going to be just c because you can't get with one step from a to b. The edge is in the wrong direction. And Adj of c is b. I think that definition's pretty clear.

For undirected graphs, you just put braces here. Which means you store-- I mean, it's the same thing. Here Adj of c is going to be a, b, and d, as you can get in one step from c to a, from c to b, from c to d. For pretty much every-- At least for graph exploration problems, this is the representation you want. Because you're at some vertex, and you want to know, where can I go next. And Adj of that vertex tells you exactly where you can go next. So this is what you want.

There's a lot of different ways to actually implement adjacency lists. I've talked about two of them. You could have the vertices labeled zero to v minus 1, and then this is, literally, an array. And you have-- I guess I should draw. In this picture, Adj is an array. So you've got a, b, and c. Each one of them is a pointer to a linked list.

This one's actually going to be a, c, and we're done. Sorry, that was b. Who said it had to be alphabetical order? A is a pointer to c, c is a pointer to b. That's explicitly how you might represent it. This might be a hash table instead of an array, if you have weirder vertices.

You can also do it in a more object-oriented fashion. For every vertex, v, you can make the vertices objects, and v dot neighbors could store what we're defining over there to be Adj of v. This would be the more object-oriented way to do it I've thought a lot about this, and I like this, and usually when I implement graphs this is what I do. But it is actually convenient to have this representation. There's a reason the textbook uses this representation. Because, if you've already got some vertices lying around and you want to have multiple graphs on those vertices, this lets you do that. You can define multiple Adj arrays, one for graph one, one for graph two, one for graph three but they can all talk about the same vertices. Whereas here, vertex can only belong to one graph. It can only have one neighbor structure that says what happens. If you're only dealing with one graph, this is probably cleaner. But with multiple graphs, which will happen even in this class, adjacency lists are kind of the way to go.

You can also do implicitly-represented graphs. Which would be to say, Adj of u is a function. Or v dot neighbors is a method of the vertex class. Meaning, it's not just stored there explicitly. Whenever you need it, you call this function and it computes what you want. This is useful because it uses less space. You could say this uses zero space or maybe v space. One for each vertex. It depends. Maybe you don't even need to explicitly represent all the vertices. You start with some vertex, and given a vertex, somehow you know how to compute, let's say in constant time or linear time or something, the neighbors of that vertex. And then from there, you can keep searching, keep computing neighbors, until you find what you want. Maybe you don't have to build the whole graph, you just need to build enough of it until you find your answer. Whatever answer you're searching for.

Can you think of a situation where that might be the case? Where implicit representation would be a good idea? Yes. Rubik's Cubes. They're really good. I

never want to build this space. It has a bajillion states. A bajillion vertices. It would take forever. There's more configurations of this cube than there are particles in the known universe. I just computed that in my head.

[LAUGHTER]

I have done this computation recently, and for five by five by five it's like 10 to the 40 states. Or 10 to the 40, 10 to the 60. There's about 10 to the 80 particles in the known universe. 10 to the 83 or something. So this is probably 10 to the 200 or so. It's a lot. You never want to build that.

But, it's very easy to represent this state. Just store where all the cubies are. And it's very easy to see what are all the configurations you can reach in one move. Just try this move, try this move, try this move. Put it back and try the next move. And so on. For an m by n by n cube in order n time, you can list all the order n next states. You can list all the order n neighbors. And so you can keep exploring, searching for your state. Now you don't want to explore too far for that cube, but at least you're not hosed just from the problem of representing the graph. So even for two by two by two, it's useful to do this mostly to save space. You're not really saving time. But you'd like to not have to store all 264 million states because it's going to be several gigabytes and it's annoying.

Speaking of space-- ignoring the implicit representation-- how much space does this representation require? V plus E. This Is going to be the bread and butter of our graph algorithms. Most of the things we're going to talk about achieve V plus E time. This is essentially optimal. It's linear in the size of your graph. You've got V vertices, E edges. Technically, in case you're curious, this is really the size of V plus the size of E. But in the textbook, and I guess in the world, we just omit those sizes of whenever they're in a theta notation or Big O notation. So number vertices plus number of edges. that sort of the bare minimum you need if you want an explicit representation of the graph. And we achieve that because we've got we've got v space just to store the vertices in an array. And then if you add up-- Each of these is an edge. You have to be a little careful. In undirected graphs, each of these is a half

edge. So there's actually two times e nodes over here. But it's theta E. So theta V plus E is the amount of space we need. And ideally, all our algorithms will run in this much time. Because that's what you need just to look at the graph.

So let's do an actual algorithm, which is breadth-first search. So to the simplest algorithm you can think of in graphs. I've already outlined it several times. You start at some node. You look at all the nodes you can get to from there. You look at all the nodes you can get to from there. Keep going until you're done. So this is going to explore all of the vertices that are reachable from a node.

The challenge-- The one annoying thing about breadth-first search and why this is not trivial is that there can be some edges that go sort of backwards, like that, to some previous layer. Actually, that's not true, is it? This can't happen. You see why? Because if that edge existed, then from this node you'd be able to get here. So in an undirected graph, that can't happen. In a directed graph, you could conceivably have a back edge like that. You'd have to realize, oh, that's a vertex I've already seen, I don't want to put it here, even though it's something I can reach from this node, because I've already been there. We've got to worry about things like that. That's, I guess, the main thing to worry about.

So our goal is to visit all the nodes-- the vertices-- reachable from given node, s. We want to achieve V plus E time. And the idea is to look at the nodes that are reachable first in zero moves. Zero moves. That's s. Then in one move. Well that's everything you can reach from s in one step. That's adjacency of s. And then two moves, and three moves, and so on until we run out of graph. But we need to be careful to avoid duplicates. We want to avoid revisiting vertices for a couple of reasons. One is if we didn't, we would spend infinite time. Because we'd just go there and come back, and go there and come back. As long as there's at least one cycle, you're going to keep going around the cycle forever and ever if you don't try to avoid duplicates.

So let me write down some code for this algorithm. It's pretty straightforward. So straightforward, we can be completely explicit and write [INAUDIBLE] code. There's

a few different ways to implement this algorithm. I'll show you my favorite. The textbook has a different favorite. I'm going to write in pure Python, I believe.

Almost done. I think I got that right. So this is at the end of the while-loop. And at that point we should be done. We can do an actual example, maybe.

I'm going to do it on an undirected graph, but this algorithm works just as well on directed and undirected graphs. There's an undirected graph. We're given some start vertex, s, and we're given the graph by being given the adjacency lists. So you could iterate over the vertices of that thing. Given a vertex, you can list all the edges you can reach in one step. And then the top of the algorithm's just some initialization. The basic structure-- We have this thing called the frontier, which is what we just reached on the previous level. I think that's going to be level i minus one. Just don't want to make an index error. These are going to be all the things you can reach using exactly i minus one moves. And then next is going to be all the things you can reach in i moves.

So to get started, what we know is s. s is what you can reach in zero moves. So we set the level of s to be zero. That's the first line of the code. There's this other thing called the parent. We'll worry about that later. It's optional. It gives us some other fun structure. We set i to be one because we just finished level zero. Frontier of what you can reach in level zero is just s itself. So we're going to put that on the list. That is level zero. i equals one So one minus one is zero. All good. And then we're going to iterate. And this is going to be looking at-- The end of the iteration is to increment i. So you could also call this a for-loop except we don't know when it's going to end. So it's easier to think of i incrementing each step not knowing when we're going to stop. We're going to stop whenever we run out of nodes. So whenever frontier is a non-empty list.

the bulk of the work here is computing what the next level is. That's called next. It's going to be level i. We do some computation. Eventually we have what's on the next level. Then we set frontier next. Because that's our new level. We increment i, and then invariant of frontier being level i minus 1 is preserved. Right after here. And

then we just keep going till we run out of nodes.

How do we compute next? Well, we look at every node in the frontier, and we look at all the nodes you can reach from those nodes. So every node, u, in the frontier and then we look at-- So this means there is an edge from u to v through the picture. We look at all the edges from all the frontier nodes where you can go.

And then the key thing is we check for duplicates. We see, have we seen this node before? If we have, we would have set it's level to be something. If we haven't seen it, it will not be in the level hash table or the level dictionary. And so if it's not in there, we'll put it in there and add it to the next layer. So that's how you avoid duplicates. You set its level to make sure you will never visit it again, you add it to the next frontier, you iterate, you're done. This is one version of what you might call a breadth-first search. And it achieves this goal, visiting all the nodes reachable from s, in linear time. Let's see how it works on a real example.

So first frontier is this thing. Frontier just has the node s, so we just look at s, and we look at all the edges from s. We get a and x. So those get added to the next frontier. Maybe before I go too far, let me switch colors. Multimedia here. So here's level one. All of these guys, we're going to set their level to one. They can be reached in one step. That's pretty clear. So now frontier is a and x. That's what next becomes. Then frontier becomes next. And so we look at all the edges from a. That's going to be s and z. s, we've already looked at, it already has a level set, so we ignore that. So we look at z. Z does not have a level indicated here, so we're going to set it to i which happens to be two at this point. And we look at x. It has neighbors s, d, and c. We look at s again. We say, oh, we've already seen that yet again.

So we're worried about this taking a lot of time because we look at s three times in total. Then we look at d. d hasn't been set, so we set it to two. c hasn't been set, so we set it to two. So the frontier at level two is that. Then we look at all the neighbors of z. There's a. a's already been set. Look at all the neighbors of d. There's x. There's c. Those have been set. There's f. This one gets added. Then we look at c. There's x. That's been done. d's been done. f's been done. v has not been done.

So this becomes a frontier at level three. Then we look at level three. There's f. D's been done, c's been done, b's been done. We look at v. c's been done. f's been done. Nothing to add to next. Next becomes empty. Frontier becomes empty. The while-loop finishes. TA DA! We've computed-- we've visited all the vertices.

Question.

AUDIENCE: [INAUDIBLE]. What notation?

PROFESSOR: This is Python notation. You may have heard of Python. This is a dictionary which has one key value, s, and has one value, zero. So you could-- That's shorthand in Python for-- Usually you have a comma separated list. The colon is specifying key value pairs.

I didn't talk about parent. We can do that for a little bit. So parent we're initializing to say, the parent of s is nobody, and then whenever we visit a new vertex, v, we set its parent to be the vertex that we came from. So we had this vertex, v. We had an edge to v from some vertex, u. We set the parent of v to be u.

So let me add in what that becomes. I'll change colors yet again. Although it gets hard to see any color but red. So we have s. When we visited a, then the parent of a would become s. When we visited z, the parent of z would be a. Parent of x is going to be s. Parent of d is going to be x. The parent of c is going to be x. The parent of f-- it could have been either way, but the way I did it, d went first, and so that became its parent. And I think for v, c was its parent.

So that's what the parent pointers will look like. They always follow edges. They actually follow edges backwards. If this was a directed graph, the graph might be directed that way but the parent pointers go back along the edges. So it's a way to return. It's a way to return to s. If you follow these pointers, all roads lead to s. Because we started at s, that's the property we have. In fact, these pointers always form a tree, and the root of the tree is s. In fact, these pointers form what are called shortest paths. Let me write down a little bit about this.

Shortest path properties. If you take a node, and you take its parent, and you take

the parent of the parent, and so on, eventually you get to s. And if you read it backwards, that will actually be a path in the graph. And it will be a shortest path, in the graph, from s to v. Meaning, if you look at all paths in the graph that go from s to v-- So say we're going from s to v, how about that, we compute this path out of BFS. Which is, follow a parent of v is c, parent of c is x, parent of x is s. Read it backwards. That gives us a path from s to v.

The claim is, that is the shortest way to get from s to v. It might not be the only one. Like if you're going from s to f, there's two short paths. There's this one of length three. There's this one of length three.. Uses three edges. Same length. And in the parent pointers, we can only afford to encode one of those paths because in general there might be exponentially many ways to get from one node to another. We find a shortest path, not necessarily the only one. And the length of that path-- So shortest here means that you use the fewest edges. And the length will be level of v. That's what we're keeping track of. If the level's zero, you can get there with zero steps. If the level's one, you get there with one steps. Because we're visiting everything you can possibly get in k steps, the level is telling you what that shortest path distance is. And the parent pointers are actually giving you the shortest path.

That's the cool thing about BFS. Yeah, BFS explores the vertices. Sometimes, that's all you care about. But in some sense, what really matters, is it finds the shortest way to get from anywhere to anywhere. For a Rubik's Cube, that's nice because you run BFS from the start state of the Rubik's Cube. Then you say, oh, I'm in this state. You look up this state. You look at its level. It says, oh, you can get there in nine steps. That's, I think, the average. So I'm guessing. I don't know how to do this in nine steps.

Great, so now you know how to solve it. You just look at the parent pointer. The parent pointer gives you another configuration. You say, oh, what move was that? And then you do that move. I'm not going to solve it. Then you look at the parent pointer of that. You do that move. You look at the parent pointer of that. You do that move. Eventually, you'll get to the solved state, and you will do it using the fewest possible moves. So if you can afford to put the whole graph in memory, which you

can't for a big Rubik's Cube but you can for a small one, then this will give you a strategy, the optimal strategy, God's algorithm if you will, for every configuration. It solves all of them. Which is great.

What is the running time of this algorithm? I claim it's order V plus E. But it looked a little wasteful because it was checking vertices over and over and over. But if you think about it carefully, you're only looking-- what's the right way to say this-- you only check every edge once. Or in undirected graphs, you check them twice, once from each side. A vertex enters the frontier only once. Because once it's in the frontier, it gets a level set. And once it has a level set, it'll never go in again. It'll never get added to next. So s gets added once then we check all the neighbors of s. a gets added once, then we check all the neighbors of a. Each of these guys gets added once. We check all the neighbors. So the total running time is going to be the sum over all vertices of the size of the adjacency list of v. So this is the number of neighbors that v has. And this is going to be? Answer?

AUDIENCE: Two times the number of edges.

PROFESSOR: Sorry

AUDIENCE: Double the number of edges.

PROFESSOR: Twice the number of edges for undirected graphs. It's going to be the number of edges for directed graphs. This is the Handshaking Lemma. If you don't remember the Handshaking Lemma, you should read the textbook. Six o four two stuff. Basically you visit every edge twice. For directed graphs, you visit every edge once. But it's order E. We also spend order V because we touch every vertex. So the total running time is order V plus E. In fact, the way this is going, you can be a little tighter and say it's order E. I just want to mention in reality-- Sometimes you don't care about just what you can reach from s, you really want to visit every vertex. Then you need another outer loop that's iterating over all the vertices as potential choices for s. And you then can visit all the vertices in the entire graph even if it's disconnected. We'll talk more about that next class. That's it for BFS.