PROFESSOR: Hello. Welcome. Today we're going to talk about one last new topic which has to do with search. So as you remember, we're working on our last topic. The last topic was probability and planning.

Last lecture we talked about probability. Mostly we focused on Bayes' theorem, Bayes' rule. That was a way of updating our belief about some situation based on new information. And this week in Design Lab 12, you'll get a chance to use that in a robot application.

The idea to Design Lab 12 is going to be that a robot is pedaling along a corridor with some obstacles off to its left. And the idea will be, you don't know where the robot is, but the robot will be able to estimate where it is by the signals that it receives from its left-facing sonars. So this is a very realistic type of state-estimation problem.

The idea is going to be that at any given time, t, you have access to your previous belief at time t minus 1. And you have access to a new observation, which is the sonar to your left. And based on those two bits of information, you will update your belief. Which means that when you start out you'll have no idea where you are, but that situation should improve with time. Ok. So that's what we're going to do in Design Lab 12.

Today we're going to blast ahead and think about the other important topic, which is search. So we're going to think about planning. We're going to be planning ahead.

We're not going to just react to the situation that's given to us. We're going to try to figure out what's the right thing to do in the future. And to do that, we're going to think about all the things we could possibly do, search through that space, and figure out the one that's, quote, "best." And we'll have to define "best" somehow.

Just to get going, I want to show you a very simple kind of a search problem. This is called the eight puzzle. The idea is to make a plan to go from this configuration, which we'll call the state, to this configuration, which we'll call the goal state. And on each move, you get to move one of the tiles into the free space.

So I could move the 8 to the right or the 6 down. Those are the only two things I could do in the start state. So I have to make up my mind which of those I would like to do. And I would like to believe that ultimately, after a series of moves, I'm going to be able to control this state into that state.

And you can imagine guessing. And we'll estimate in a moment how big is the guess space. I mean, if the guess space only had like four elements in it, guessing's a fine strategy. If the guess space has a lot more elements than that, guessing's probably not a good idea.

So I previously ran our search algorithms, the ones that we'll develop during lecture, on this problem. And here's the solution that our search algorithm came up with. It's not exactly what you might do the first time you touched it. OK, I made it.

If you were counting, I made 22 moves. The question is, how difficult was that problem and how good was that solution? Was that a good solution or a bad solution? Is there a better solution? How much work did I have to do in order to calculate that solution?

To get a handle on that, let's start by asking a simple question. How many configurations are there? I got there in 22. What was the space of things I had to look through? How many different board configurations exist?

So think about that for 20 seconds. Talk to your neighbor. And figure out whether it is 8-squared, 9-squared, 8-factorial, 9-factorial, or none of those.

So how many bar configurations do you see? Raise your hand, show me a number of fingers, so I can figure out roughly how people were-- that's excellent. Very good participation, and nearly 100% correct.

The answer is number (4). You can think about this. What if you took all the tiles out and threw them on the floor, and then put them in one at a time?

Well, you would have 9 possibilities for where you wanted to put the first one. Then you would have 8 possibilities for where you wanted to put the second one. Then you'd have 7 possibilities for where you put the third one, et cetera, et cetera. Even though the space doesn't have a number on it, it still sort of counts. And so you end up with 9-factorial.

And the point is, 9-factorial is a big number. 9-factorial is 362880. So if you thought about simply guessing, that's probably not going to work all that well, right? Even if you guessed, you have on each-- there's a third of a million different configurations that you have to look at. And that's if you didn't lose track of things. If you lost track of--

Oh, my. It's coming up-- almost, anyway. It looks like it's chopped off at the top. So ignore that for now. Look over here.

So even if you didn't confuse yourself, there's a space of a third of a million things to look at. And if you confused yourself, there's even more. So it's not a huge problem by computer science standards. But it's certainly not a trivial problem. It's not something that you can just guess and get right.

So what we want to do today is figure out an algorithm for conducting a search like that. We'd like to figure out the algorithm, analyze how well it works, optimize it, and try to find out a way to find the best solution, where "best" for this particular problem would mean minimum path length. So figure out the best solution by considering as few cases as possible.

Obviously, if you enumerate all the cases, that should work. The problem is, it will be interesting to solve problems where that enumeration is quite large. Even here, the enumeration is quite large. So let's think about the algorithm. And I'll think about the algorithm by way of an even simpler, more finite problem.

What if I thought about a grid of possible locations where I could be. Maybe this is the intersections of streets in Manhattan. I want to go from point A to point I. What's the minimum-distance path?

I hope you probably can all figure that out. What I want to do is write an algorithm that can figure that out. And then if we write the algorithm well, we'll be able to use it for the tile problem, which is not quite so easy to do.

The way we're going to think about doing that is to organize all of our possible paths through that maze, in a tree. So if I started at A, I have a decision to make. I could either go to B or D.

Then if I went to, say, B, I could either then go to A or C or E. I've organized them alphabetically for no particularly good reason. Just-- I needed some order.

Then if I went from A to B to A, say, then I could either go from A to B or D. That's illustrated here. So the idea then-- ah, it works. So now it looks like they're all three working.

So the idea is, think about the original problem. The original problem is find the shortest path through some grid. I want to go from A to I. And I'll think about all the possible paths on a tree. Then the problem is that for the kinds of problems we're going to look at, that tree could be infinite in length.

Oh, that's a bummer. That means that the strategy of building the tree and then searching it is probably not a good strategy. So what we'll do instead is, we'll try to write the algorithm in such a way that we construct the tree and search for the best solution all in one pass. Then hopefully, if we find a solution in some finite number of steps, we'll only have built part of the tree. But we'll have built the part that has the answer.

The idea, then, is going to be, think about what is the path we want to take, by thinking about the tree of all possible paths. But what we want to do is write code that will construct the tree on the fly, while it's considering how good were all the different nodes. Ok. So how are we going to do that?

We'll be working in Python, not surprisingly. We'll represent all the possible locations. We'll call those states. So the problem will have states A, B, C, D, and we'll just represent those by strings. That makes it flexible. That makes it arbitrary.

Then we'll think about transitions, not by enumerating them. Remember, we don't want to enumerate them, because there could be infinitely many of them. So how's the other way we could do it?

Well, we'll embody that information in a program. We'll write a procedure called "successor" that will, given the current state and action, figure out the next state. So that's a way that we can incrementally build the tree.

So imagine here, if I started in A and I executed action 0 or 1, I would end up in B or D, respectively. So I tell you the current state and the current action, and the successor program then will return to you the new state. That's all we need to construct the tree on the fly.

Then, to specify the particular problem with interest, I have to tell you where you start. So I have to define initial state. And I have to tell you where to end.

I could just tell you the final state. But in some of the problems of the type that we will want to do, there could be multiple acceptable answers. So I don't want to just give you the final state.

I'll give you a test. I'll give you another procedure, called "goalTest." And that goal test, when passed an input which is a state, will tell you whether or not you reached the goal. That way, for example, all the even-numbered squares could satisfy the goal, if that were the problem of interest.

Or all the states on the right could satisfy the goal. It's just a little bit more flexible. The idea, then, is that in order to represent that tree, we'll do it by specifying a procedure called successor and specifying the start state and the goal test.

So here's how I might set that up for the simple Manhattan problem that I showed-- that I started with. So I want ultimately to have something called successor that eats a state and an action. I've built the structure of Manhattan into a dictionary.

The dictionary lists for every state-- A, B, C, D, E, F, G, H, I-- for every state, it associates that state with a list of possible next states. So if I'm in A, I could next be in B or D. I could next be in B or D.

I've organized these arbitrarily in alphabetical order so I can remember what's going on. So the next states are all in alphabetical order. The number of next states depends on the state. I'm not going to worry about that too much. I'm just going to specify the action as an integer-- 0,1,2,3 -- however many I need.

So the possible actions are taken from that list. The possible action might be do action 0, do action 1, do action 2. So if I did action 2 starting on state E, I would go to-- so action started at 0 -- so 0, 1, 2 -- I would go to state F. OK? Is that all clear?

The initial state is A, and the goal state is, return S equal to I. So if S is equal to I, it returns True. If S is not equal to I, it returns False. I'm not quite done. That's enough to completely specify the tree, but now I have to build the tree in Python.

Not surprisingly, from our object-oriented past, we will use an object-oriented representation for that tree. So we'll specify every node in the tree as an instance of the class SearchNode. SearchNode is trivial.

SearchNode simply knows, what was the action that got me here? Who's my parent? And what's my current state?

So when you make a new node, you have to tell the constructor those three things. What was the action that got me here? What's my current state? And who is my parent?

Knowing the node, you're supposed to know the entire path that got you here. So we'll also add a method which reports the path. So if I happen to be in node E, my path ought to be I started in A, I took action 0 and got to B, and then I took action 2 and got to E. This subroutine is intended to do that.

If my parent is "none," which will happen for the initial state, simply report that the path to me is none, A. However, if I'm anybody other than the initial node, if I'm other than the initial node, then figure out the description of the path to my parent, and add the action that got me here, and my state. So that's what this is.

OK, so what are we doing? We specify a problem by telling you the successor function, the start state, and the goal test. Then we provide a class by which you can build nodes to construct, on the fly, the search tree.

Now we're ready to write the algorithm. Here's the pseudocode for the algorithm. What do we do? We initialize-- so we're going to be doing a s-- oh, this is very confusing. I'm trying to solve a search problem. To solve the search problem, I'm going to search through the tree.

So I'm going to think about the state of my search through the tree by way of something we'll call the agenda. Very jargon-y word. I completely apologize for it. I didn't invent it. It's what everybody calls it. Sorry.

The agenda is the set of nodes that I'm currently thinking about. So I'll initialize that to contain the starting node. Then I'll just systematically keep repeating the same thing over and over again. Take one of the nodes out of the agenda, think about it, replace that node by its children.

While I'm doing that, two things are supposed to happen. I'm supposed to construct the search tree. But I'm also going to be looking over my shoulder to see if I just constructed a child who is the answer. Because if I just constructed the answer, I'm done. Ok.

So initialize the agenda to contain just the starting node. Then repeat the following steps. Remove one node from the agenda. Add that node's children to the agenda.

And keep going until one of two things happens. Either you found it-- goal test returned True-- or the agenda got empty, in which case there must not be a solution. If I've removed all of my options and still haven't found anything, then there's no solution.

Ok. So what's the program look like? It's actually remarkably simple, especially when you think about just how hard the problem is. Imagine if you wanted to do that tiles problem with a very simple-minded "if this then this, if this then this." We're talking about a third of a million ifs, right?

That's probably not the right way to do it. This program is going to end up being about this long. It'll fit on this page. And it's going to be able to handle that case, or even harder cases.

Define the search procedure. The search procedure is something that's going to take the initial state, the goal test, the possible actions, and the successor sub routine, the successor procedure. That's everything you need to specify the problem. And it's going to return to me the optimal path.

First, step (1). Initialize the agenda to contain the start node. I want to put the start node in. Well, there's a chance-- I want this procedure to be general purpose-- there's a chance that that start node is the answer.

So take care of that first. If you're already there, return the answer. The path to the answer is me.

I'm trying to create the agenda. I'm trying to put the first node into the agenda. There's a chance that first node is the answer. If that first node's the answer, return the path to me. Which is, take no action. You're here.

But that's not likely to be the case for the kinds of questions we ask, in which case we will create a list that contains one node, which is the node that represents the start node. Then, repeat 'remove a node' -- which we'll call the parent-- from the agenda. And substitute -- replace that node that we pulled out of the agenda -- replace that with the children.

While not empty of agenda-- empty is some kind of a pseudo-routine that I'm going to fill in, in a minute-- while the agenda is not empty, get an element out of the agenda, which we'll call the parent. Then I want to think about all the children.

Well, there's a list of possible actions. So for a in actions, do the following things. Each parent can have multiple children, one for every possible action.

So for a in action, figure out what would be the new state. The new state is just the successor of the parent state. Remember, the parent is a node, right? The parent is a node. We're constructing nodes in the search tree.

But nodes know their state. So figure out the new state, which is the successor of my parent, the guy that I pulled out of the agenda. Make a new node, which corresponds to this child. Then ask the question, did the new state satisfy the goal test?

If it did, the answer is the path to the new node. Ok. So create a new state, which is the successor of the parent under the given action A. Create a new node. See if it's the end. If it is, just return, I'm done. Return from search.

Otherwise, add it-- again, one of these pseudo-procedures. We'll fill that in, in a minute-- add the new node into the agenda.

There's several things that could happen when I run this loop. If the node has no children, I will take out the parent and not put anything back in. If the node has multiple children, I could take out one node and put in more nodes than I took out, so the agenda could get longer. So the agenda could either increase in length or decrease in length as a result of spinning around this loop.

Also, we could either identify a goal or fail to identify a goal. So as it's increasing and decreasing, we either will or won't find an answer. Ok. Now the trick-- the only thing that makes this complicated-- is that order matters. So those pseudo-operations, whatever they were-- get element and add-- exactly how I get element and exactly how I add it to the agenda affects the way I conduct the search.

Let's think of something very simple. Let's always remove the first node from the agenda and replace it by its children. So pull out the first node. And put back into the beginning of the agenda the children of the first node.

How? I would start out in step (0). I would put the start node into the agenda. So now there's one element in the agenda, the start node.

Then, on the first pass through the loop, I would pull out that. That's the first node in the agenda. There's only one node in the agenda. That is the first one.

Pull out the first one and replace it by its children. Its children are AB and AD. I'm representing the nodes in this notation by the path, because the same state can appear multiple times in the tree.

Notice that I could walk ABAB -- which would correspond to the same state being repeated in the tree. So I can't, when I'm writing it down here, represent the node by a state. But I can represent a node by a path. So on the first pass through the loop, pull out the first item in the agenda, which is A, and push back that A's children. Well A's children are AB and AD.

Ok. So now on the second pass, the rule is pull out the first guy and replace it by the children. Now the first guy is AB. So I'm here.

So pull that guy out and replace him by his children. His children are ABA, ABC, and ABE. AD is left over. The number of elements in the agenda got bigger.

Next step, pull out the first item in the agenda, replace it by its children. The first item in the agenda is ABA. The children of ABA are ABAB and ABAD.

Ok. Notice the structure of what's going on. Ignore the stuff on the bottom, and just watch the picture on the top. So I start by putting A in the agenda, then its children, then its children, then its children.

So when I implemented the algorithm-- take out the first and replace it by its children-- I'm searching along the depth first. I'm going deeper and deeper into the tree without fully exploring all the horizontal spaces. So I'm tracing a line down that way. If you imagine this tree-- I've only represented the first three layers of nodes here-- this tree goes on forever. It's an infinite tree, because you can walk around in that Manhattan grid forever.

There's no limit to how long you can walk around. So although I'm only listing the first three, the tree, in principle, goes on forever. And this algorithm will have the feature that it walks along the left edge.

We call that depth-first search because we're exploring depth first, as opposed to breadth. That results because of our rule. The rule was, replace the first node by its children.

Let's think about a different rule. Let's replace the last node by its children. We start by initializing the agenda to the node that represents the start state. So that's the path A. Then pull out the last node in the agenda-- that's A-- and replace it by its children. Its children are still AB and AD, just like before.

Now the answer differs from the previous answer. Because when I pull out the last node, I'm pulling out AD now, instead. And now I replace AD by its children, which are ADA, ADE, ADG. Repeat, and what I've got is a different, but still depth-first search.

So I've looked at two different orderings-- pull out the first node from the agenda and replace it by children, pull out the last node and replace it by its children. Both of those algorithms give an exploration of the decision tree searching out depth first. So it's going to try to exhaustively go through the entire depth before it tries to explore the width.

As an alternative, think about a slightly more complicated rule. Remove the first element from the agenda and add its children to the end of the agenda. So initialize it with the start state, A. Pull out the first element from the agenda-- that's A-- and replace it by its children, which is AB, AD.

Now pull out the first guy-- the first guy is AB-- and put its children at the end. It's children are ABA, ABC, ABE-- ABA, ABC, ABE-- and they are now put at the end. So that on the next step, I'll pick up AD-- the guy at the beginning-- and put AD's children at the end. et cetera, et cetera, et cetera, et cetera, et cetera, et cetera. The idea being-- and now, pay no attention to the bottom for a moment and just think about the pattern that you see at the top.

In this order, where we remove the first node and put its children at the end of the agenda, has the effect of exploring breadth first. So we call that a breadth-first. so the idea is, we got this generic set of tools that let us construct search trees. But the order by which we manipulate the agenda plays a critical role in how the search is conducted.

And the two that epitomize the two extreme cases are, what would happen if I replace the last node by its children? Or what would happen if I remove the first node and put its children's at the end? Those two structures have names because they happen so often.

We'll call the first one a stack and the second one a queue. The stack-based is going to give us depth first. The queue-based is going to give us breadth first.

So, stack. How do you think about a stack? You think about a stack by saying, OK, I've got a stack. A stack is like a stack of dishes. So here's my table, and I'm going to rack my dishes up.

I'm going to put them on a stack. So I make a stack. OK, I made the stack-- push a 1, push a 9, push a 3. Push a 1, push a 9, push a 3. That's how I do a stack.

Then pop. When I pop, the 3 comes out. Then pop, then the 9 comes out. Then push a minus 2. Then pop. Now the minus 2 comes out. OK?

It's stack-based. So the last in becomes the first out. That was the rule that we wanted to have for the depth-first search.

It's very easy to implement this. We can implement it as a list. All we need to do is be careful about how we implement the push and pop operators.

So if we set up the push operator to simply append to the end, and then pop ordinarily pops from the end, we'll get the behavior of a stack. That gives me, then, the rules that I would want to use for those procedures, the get element an add. I will use these stack-based operators.

The other alternative is a queue. A queue is different. A queue is like when you're waiting in line at the Stop & Shop. The queue is, I've got this queue here and I've got the server over here.

The first person who comes into the queue-- so say I push one. So now 1 goes into the queue. Then another person walks up while he's-- the second person lines up behind the first person. Then I push a 3.

But the way the queue works is that when I pop the next person off the queue, I take the head of the line. So the 1 comes out. If I pop again, the 9 comes out. If I then push a minus 2 and pop, then the next person in the queue comes out.

And it's like that. It's queue based versus stack based. And the queue based is the one that we want to do for a breadth-first organization.

And the implementation of a queue is very trivially different from the implementation for a stack. The only difference is that I'll manipulate the list by popping off from the head of the queue. So pop takes an optional argument, which when 0, tells you the-- the argument tells you which element to pop. So when you specify the zero-th one, it takes it from the head of the queue.

That makes it very easy now to replace the pseudo-procedures with real procedures. If I wanted to implement a depth-first search, I would replace the "create a list that contains the agenda" with "create a stack that will contain the agenda." So create a new stack, the agenda is a stack. And then rather than simply sticking the node-- the start node-- into a list, I will push it into the stack.

So agenda is a stack. Agenda.push, the initial node. And then every time I want to get a new element out, I'll agenda.pop it.

And every time I want to put something into it, I'll agenda.push it. Other than that, it looks just the same as the pseudocode that I showed earlier. So there is an implementation, then, for a depth-first search.

If I wanted instead to do breadth, it's trivial. Change the word "stack" to the word "queue." Now create an agenda that is a queue, but queues have the same operations-- push and pop and empty-- that stacks have.

So nothing else in the program changed. All I needed to do is change the structure of the thing that's holding the agenda. Everything else just follows.

Ok. So that's everything we need, right? Now what I want to do is think through examples and think about the advantages and disadvantages of different kinds of searches. And I want to go on to the second step that I raised in the first slide. I want to think about, how do I optimize the search?

As I said, even that simple little tile problem, even the eight puzzle-- eight sounds easy, right? Even the eight puzzle had a third of a million different states. I don't necessarily want to look through all of them. I want to think now about these different search strategies, and how optimal are they relative to each other, and are there ways to improve that?

Now some of you may have noticed that all of these paths don't seem equally good. So take a minute. Think about it. Remember the problem. The problem was this walk around Manhattan problem.

I wanted to go from A to I. This was the tree of all possible paths from A to I. What I'd like you to do is think about whether all of those paths are important. Could we get rid of some of them?

So the question is, can I throw away some of the terminal nodes? Notice that I'm using the word "terminal" kind of funny here. The tree keeps going. The tree is actually infinite in length. So by "terminal," I just mean this row three.

So could I throw away some of the nodes in row three? And in particular, how many of them could I throw away? 0, 2, 4, 6, or 8? or Raise your hand with the funny coding.

And the answer is-- come on, come on. Raise your hands, raise your hands. Blame it on your neighbor. That's the whole point. OK, it's 2/3 (5) and 1/3 (4)

How'd you get (5) and (4)? Yes?

AUDIENCE: When you have that [INAUDIBLE] nodes, you may know that there's [? a procedure ?] [INAUDIBLE] before.

PROFESSOR: Good. If you're walking around Manhattan, and you're trying to go from A to I, and if you spun around in this loop and came back to A, that would probably be a bad path, right? So revisiting a place you've been before is probably a bad idea, if what your goal was, was to get from A to I in the shortest possible distance.

So that's exactly right. So I would like to identify instances where I go back to where I started. So for example, that A. That A is bad. That means I went back to the start place. I'm just starting over.

So if I think about those, I can identify by red all the places where I'm repeating. So ABA, don't really care what happens after that. ABCB, well, that's B again.

So that's just brain dead, right? So I can actually remove a fair amount of the tree by simply getting rid of silliness. Don't start the path over again, where "over" means if you come to a place you've been before, stop looking there. That's not the right answer.

And so you can see there that I actually deleted half of the tree. The number of nodes on the third line was 16. And 8 of them had the property that they repeated. Yes?

AUDIENCE: [INAUDIBLE] [? after D. ?] Are you [INAUDIBLE]?

PROFESSOR: This B and D. So there's no reason to consider this D, even though the D didn't repeat.

AUDIENCE: That would be [? AD? ?]

PROFESSOR: ABC,

AUDIENCE: [? That would be E? ?]

PROFESSOR: ABED.

AUDIENCE: [INAUDIBLE]

PROFESSOR: So I didn't, in this path, ever hit D before.

AUDIENCE: When it did lead to the path there, you'll get the same one, AD [INAUDIBLE]

PROFESSOR: I guess I don't understand.

AUDIENCE: What I'm saying is that without getting the path AD, I still get (2)?

PROFESSOR: Yes. So this D seems clearly inferior to that D. Yes, that's absolutely true. So this is a very simple rule for removing things. You're thinking of a more advanced rule.

So if you saw-- if there's a shorter path to a particular place, don't look at the longer path. You're absolutely right. So in fact, there might be more severe pruning that you could do. There might have been an answer that was bigger than 8 -- right?

And so you're absolutely right. Ok. Let me ignore that for the moment and come back to it in about four slides. You're absolutely right.

So what we want to do now is take that idea of throwing away silly paths and formalize it so that we can put it into the algorithm. And we'll think about that as pruning rules. So the first pruning rule is the easy one. Don't consider any path that visits the same state twice. That doesn't pick up your case, but it does pick up 8 cases here.

So that's easy to implement. All we need to do is-- down here where we're thinking about whether this is a good state to add or not-- we just ask, is it in the path? So if the state that I'm about to put in the path is already in the path, don't put it there. If you don't shove it back into the agenda, it'll get forgotten.

So before you shove it into the agenda, ask yourself the question, is it already in the path? And so I do that here. Keep in mind, I popped out an element called the parent.

I'm looking at the children. The children's state is called "new state." So I ask, is new state in the parent's path? So parent.inpath of new state.

So that means I have to write inpath. Inpath is easy. It's especially easy if we use recursion. So inpath says, if my state is state, then return True. I'm in the path.

If that's not true, and I don't have a parent, then that means I'm the start state. That means it wasn't in the path. And if neither of those is true, ask the same question of my parent. So that makes it recursive.

So consider two cases that could either make it true or false. It would be True if I'm currently sitting on a node that happens to be the same state. It would be False if I recursed the whole way back to the start state and hadn't found it yet.

So there are two termination states-- I landed on a state in the path that was the same as new state, or I ran the whole way back to the start state and didn't find it. Those two terminate by doing returns -- return True or return False.

The other option is that I don't know the answer, ask my parent. So just recurse on inpath, and ask my parent to do the same thing. So that's the way I can figure out-- I can implement pruning rule (1).

Now pruning rule (2) -- if multiple actions lead to the same state, only think about one of them. That actually doesn't happen on the Manhattan grid problem. Because you can imagine search cases where there are three different things that you could do.

In fact, you saw some of those when you were coding the robot last week. There were multiple ways you could end up at the state at the end of the hall. You could get there by being there and moving left, which you hit the wall. Or you could get there by being here and moving left. Both of them left you in the same place.

So if you're planning a search, you don't need to distinguish among those, because they take the same amount of steps. So since they take the same amount of steps, we don't need to search further. So we can collapse them. That's called pruning rule (2).

That's also easy to implement. What we do is, we keep track of, for every parent, what are all of its children. If the parent already has a child at that place, throw away the excess children. That doesn't sound good.

So keep track of how many child states I have. Make a list. And if the new state didn't satisfy the goal, ask if it's already in the list of children. If it's already there, pass. Don't do anything.

Otherwise, do pruning rule (1). And then, before you push it into the agenda, also push it into the list of new children. That's a way of making sure that if there's multiple ways to get the same state, you only keep track of one. So that's an additional pruning rule.

So now let's think about how we would implement these. Let's think about the solution to a problem where we want to apply a depth-first search on this Manhattan problem, to get from A to I. So let's think about-- let's go up-- so I want to think about, how do I apply depth-first search to that problem?

So think about the agenda. So the agenda, I initialized it with the node that corresponds to the start state, so that's A. I'm doing depth first. What's the rule for depth first? Pop the last guy, replace it by his children.

OK, so pop the last guy. What's the last guy? The last guy is A. Replace it by his children. What's his children of A? Well, there's two of them, AB and AD.

Ok. So I'm done with the loop for the first level. So pop the last guy, that's AD.

Replace it by his children.

What are the children of AD? Well, what could D do? D could go to A or E or G. A's brain dead, so I don't want that one. So I'll think about E and G.

So ADE, ADG. By the way, stop me if I make a mistake. It's really embarrassing.

OK, pop the end, ADG. Who's the possible children of ADG? ADG? Well, it could go back to D, but that's stupid. So ADGH seems to be the only good one.

Pop the last one, ADGH. And who's his children, ADGH? ADGH has children E, G, and I. But I don't want G, because that's brain dead. So ADGH, E or I. And that one won, right? Because I got to A.

Everyone see what I did? I tried to work out the algorithm manually. So the idea, then, was that-- so how much work I do?

I visited 1, 2, 3, 4, 5, 6, 7. And then I found it. So I did 7 visits. And I got the right answer.

So both of those are good-- 7 is a small number, and getting the right answer. Both of those are good things. And in general, if you think about the way a depth-first search works-- here's a transcript of what I just did.

This will be posted on the online version. So you can see it, even though it's not handed out to you now. So you can look this up on the web.

So in general, depth-first search won't work for every problem. It happened to work for this problem. In fact, it happened to be very good for this problem.

But it won't work for every problem because it could get stuck. It could run forever in a problem with infinite domain. This problem has infinite domain. So if I were to choose my start and end state judiciously, I could get it stuck in an infinite loop. That's a property of depth-first search.

Even when it finds a path, it doesn't necessarily find the shortest path. Well, that's a bummer. But it's very efficient in its use of memory. So it's not a completely brain-dead search strategy, but it's usually brain dead.

So let's think about breadth-first search as an alternative. Again, all we need do is switch the idea of thinking about stacks versus queues. Take off the beginning, add to the end. That's the way queues work.

So now let's do the same problem with a breadth-first search. So I start with the agenda. I put in A. I pop off the head of the queue and stuff the children at the end.

I pop off the beginning and stuff the children AB, AD. That looks right. That's the end of path one.

Now I pop off the beginning and stick in the children. What are the children of AB? Well, AB could go to ACE. A is brain dead, so ABC-- ABC or ABE. ABC or ABE, that looks right.

Now pop off this guy, AD, and put his children at the end. That's ADE and ADG. I don't think I made a mistake yet.

Pop off the first guy, ABC. Stick in his children ABC-- ABC, it could go to B or F. Looks like F is the only one that makes any sense. ABC, that looks right.

ABE, put his children ABE-- E could go to B-- that's brain dead-- D, F, or H. D, F-- wait.

AUDIENCE: AB.

PROFESSOR: AB, thank you. I'm supposed to be doing ABE followed by something, ABE followed by something. I don't want B. D is fine, F is fine, and H is fine. D, F, and H. OK so far?

AUDIENCE: [INAUDIBLE]

PROFESSOR: Oh no, it's not right? OK, what did I do wrong?

AUDIENCE: Just AB. It's the fourth--

PROFESSOR: Oh, here. That's up here. Is that [INAUDIBLE].

AUDIENCE: Yeah.

PROFESSOR: Thank you. That would be embarrassing. OK, next pop off AD-- This is why we have computers, right? We don't normally do this by hand. OK, so ADE-- if I had ADE, I could do B-- that seems OK, D seems bad-- F, or H. So it would look like B, F, H.

OK, ADG. A, D, G. It looks like H is my only option. ABCF. A, B, C, F. Looks like I could do E or I. Finally.

Now the only question is whether I got the right number of states. Let's assume I did. So 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 -- which happens to be the right answer. At least it happens to be the answer I got this morning when I was at breakfast.

So what did I just do? I just did a breadth-first search. Here's a transcript. 16 matches, good. Breadth-first search has a different set of properties. Notice that it took me longer.

But because it's breadth first, and because each row corresponds to an increasing path length, it's always guaranteed to give you the shortest answer. That's good. So it always gives you the shortest answer.

It requires more space. I mean, you can see that just on the chalkboard. And also it still didn't take care of your problem. So this still seems like there's too much work. I looked at 16 different places.

I did 16 visits. There's just something completely wrong about that, because there's only 9 states. How could it take more visits than there are states? So that just doesn't sound right. And it's for exactly your point.

And so there's another idea that we can use, which is called "dynamic programming." The idea in dynamic programming, the principal is, if you think about a path that goes from x is z through y, the best path from x to z through y is the sum of two paths-- the best path from x to y and the best path from y to z. If you think about that, that has to be the case.

And if we further assume that we're going to do breadth first, then the first time that we see a state is the best way to get there. So what we can do then, in order to take care of your case, is keep track of the states we've already visited. If we've already visited a state, it appears earlier in the tree, there's no point in thinking about it further. That's the idea of dynamic programming.

And that's also easy to implement. All we need to do is keep track of all those places we've already visited. So we make a dictionary called "visited." So I initialize before I start looking at the children. I initialize right after I set up the initial contents of the agenda.

I create this dictionary called visited. And every time I visit a new state, I put that state in the visit list. Then before I add the child to the agenda I ask, is the child already in the visit list?

If the child's already there, well forget it. I don't need him. Otherwise, just before you push the new state, remember now that that's an element that's been visited.

So the idea, then, is that by keeping track of who you've already looked at, you can avoid looking-- so if there's a depth-3 way to get to D, and a depth-2, then I don't need to worry about the previous ones, because it's already in the visit list. Yes?

AUDIENCE: Why do we still need the new child states up there?

PROFESSOR: Why do we still have the new child states?

AUDIENCE: The placement was based [? on-- ?] the [INAUDIBLE] state.

PROFESSOR: I think you're right. I should think about that. I think you're right. I think when I was modifying the code for the different places I slipped and could have removed that line.

I think you're right. I'll have to think about it, but I think you're right. So if that line magically disappears from the online version, he's right.

OK, so now one last problem. I want to see if I can figure out what would happen with dynamic programming. So I want to do breadth first. And just as a warning, I'm hypoglycemic at this point. So there may be more errors than usual.

So I need to keep track of two things. I need to keep track of the visit list. And I need to keep track of the agenda. So there's two lists I have to keep track of.

OK. Let's start out by saying that the agenda contains the start element. That's A. That means we visited A. It's breadth first, so I want to take the first guy out of the queue and add his children to the end of the queue.

So take the first guy out of the queue. Add his children. A's children are B and D, which means that I've now visited B and D.

Now I want to take out the first guy from the queue, AB, and I want to put his children at the end of the queue. AB's children are A-- that's been visited, C-- not visited, and E-- not visited. So ABCE. But that visits C and E.

Now I want to take out AD and put its children at the end. AD is AEG. A is visited, E is visited, which leaves just G, so ADG . And that visits G.

Then I want to take out ABC and put in its children, A, B, C. ABC, oh dear. AB-- I'm looking up there. I said I'm hypoglycemic. ABC-- ABC -- could be B or F. Well, B's no good. Which leaves F, but that visits F.

So now, ABE. ABE, so that could be B, D, F, H. B-- visited, D-- visited, F-- visited, H-- OK. That visits H.

Now take out ADG. Children of ADG-- ADG, two children, D and H. D and H, they're both there. That didn't work.

There are no children of ADG. ABCF-- ABCF, three children-- C, E, I. C-- visited, E-- visited, I-- done.

Found the right answer. 1, 2, 3, 4, 5, 6, 7, 8 -- 8 visits. So I've got the same answer, and it's optimal, and I did fewer than 9 visits. 9 was the number of states. So this algorithm will always have those properties. So the dynamic programming-- oh, I forgot a slide.

The dynamic programming with breadth-first search will always find the best. It will never take longer than the number of states. So in this problem that had a finite number of states, even though I had an infinite number of paths-- because you can go around in circles-- it'll still never take more than the number of states. And all that it requires to implement is to maintain two lists instead of one.

So the point, then, is that today we looked at a variety-- we looked at two real different kinds of search algorithms, depth-first search and breadth-first search. And we looked at a number of different pruning rules.

Pruning rule (1) -- don't go to some place that you've already visited. Pruning rule (2) -- if you have two children that go to the same place, only think about one of them. You can consider dynamic programming to be a third pruning rule, because that's the effect of it.

And the final announcement, don't forget about Wednesday. Have a good week.