**PROFESSOR:** So hello. Today I want to talk about, and I want to finish talking about, search algorithms. So last time we started to think about a framework within which to think about search and the important thing was to figure out a way to be systematic about it. So we figured out a way to organize the way we think about searching by way of a tree. Put all the possible places where we could be in the search, consider all the possible actions that we could take, so if we started it at A, there are two different actions we could've taken, we could have gone to B or D Think of the actions as the edges of the graph and then think about where we land.

Then, by way of that graph, think about the shortest path to the goal. So that was the idea and the big outcome was order matters. So if we were to construct an agenda, which is the list of nodes that we are currently considering, if we started at A, we would start by putting A on the agenda. Then we would pop A out of the agenda and replace it with its children, its children are AB and AD.

If the algorithm was replaced, the first node in the agenda by the children, the first node is AB. So then we would pop AB and replace it with its children, AB has children ACE, etc. And the result would be something that we call a depth first search, because what's happening is we're following line lines deeply before we look crosswise. And there's a million varieties of this that you could think of. You would get the same kind of algorithm if you replaced the last node by its children. Then you would start with A, replace it by its children, which is still B,D -- but now expand D in terms of its children.

Then you'd take the last trial, expand it and we would get another depth search. So the idea is whatever order you choose affects the solution that you find. Generally,

we're interested in finding a search that locates the best, shortest path, and so a good method is to remove the first node from the agenda and add the children to the end. That's a cube-based ordering where the first out is the first in.

Then if you imagine starting at A, replacing it by its children, B,D, take the first guy out-- that's AB-- consider its children and put them at the end of the list. Then we go back and expand AD before we think about the children of AB and so the result-- if you just follow the red up here-- the result of that search is what we call breadth first. So we systematically propagate down the search tree looking at short paths first. So that's the idea, the idea is search is easy, you organize it around one of these graphs and just systematically run through the search by keeping track of what we call the agenda, the nodes under consideration, and the only trick is that order matters.

We found two useful orders last time, first in, first out and last in, first out, one of those giving depth first, which is generally not a good idea. The other giving breadth first, which is generally a much better idea.

Today what I want to do is generalize that structure to take into account a much more flexible group of problems. That'll give rise to something we call uniform cost search and the other thing that I want to think about is, again, the order. By thinking about the order, we can drastically improve the efficiency of a search and that idea gives rise to something that we'll call heuristics.

So the idea then that I want to look at first in terms of uniform cost search is the idea that so far, we've only looked at problems where the cost of each child is the same. We were only looking at how many children in total, how many generations did we have to go through to get from the starting point to the goal. That's an important class of problems, but it's by no means the only important class of problems and in fact, the most trivial problems you can think of don't fall into that class. So for example, imagine that what we were doing, so I motivated the entire problem last time in terms of searching for a path on a Manhattan Grid. So if I wanted to go from A to I, where all the distances are equal, then minimizing the number of

generations, minimizing the number of intersections that I go through, will give rise to the best search.

However, imagine that one particular node is way off the map. Then the number of generations that I go through is obviously not the right answer because there's a penalty for taking a path that goes through C because the distance between B and C is so much bigger than the distance, for example, from D to G.

So the first thing that I want to look at is how do we incorporate that kind of new information into the search algorithm? So before I do that, think about how the breadth-first search-- the thing we found last time that was the best-- breadth-first search with dynamic programming -- think about how it would approach the problem and think about as we go through step by step, what is it doing wrong? OK? So I'm going to go through an algorithm that is known not to work -- with the idea that you're supposed to identify as I'm doing that what step was wrong.

So imagine that I'm doing this and then I'm going to compare it in a moment to C being off the map. So if I do this problem with dynamic programming, I have to keep track of how many nodes I already visited and I have to keep track of all the nodes under consideration. The nodes under consideration is what we call the agenda and I will call the list of nodes that we're keeping track of for dynamic programming, I'll call that the visited list because we'll add nodes to the list. We'll add states to the list as we visit them.

So we start out the algorithm with node A being on the agenda, we're trying to go to node I. And by the time I've started, I've already visited A. So the starting point is that the visited list has one element in at A, the agenda has one element in it, A. So the algorithm is going to be pop the first guy out of the agenda and add the children to the end, paying attention to the visited list. So pop A out of the agenda, the children of A are B and D. As I visit them, they get added to visited list, B and D. So I pop the first person out, that's AB. Then I want to think about the children of AB, that's ACE. A is already visited, so I don't need to add that again, but C and E are not, so I add them and add them to the visited last.

Then I pop AD out, well the children are AEG. A and E are in the list already, G is not, so I end up adding G to the list. Next is to pop out C. The children of C are B and F. B was in the list, F was not so I add the trial that has F at the end. Then the next one is E. E has children BDFH. B,D,F, the only new one is H, then G. Take G out, the children of G are D and H but they're already in a visited list, so I don't need to worry about them. Then A,B,C,F, so F is this guy, C,E,I. CE I is not in the list, and that's my answer. Yes?

**AUDIENCE:** [UNINTELLIGIBLE] once more, then you get another path that's equally short. So why is that not the correct answer?

**PROFESSOR:** A slightly different algorithm might give me that solution. All that I'm tracing here, I'm trying to be consistent with the algorithm that we discussed last time. But that's a very good point. So the breadth-first search with dynamic programming is guaranteed to give you a solution that has minimum length. It's not guaranteed to give you a particular solution of minimum length. So when there exists multiple solutions with the same length, this search algorithm might to give you any of them, and that's an important thing to keep in mind.

So with regard to the problem, I'm trying to think ahead. I'm trying to think ahead to where this C is off the map, it's over here someplace. So what I want to do is stop thinking about how many hops there were and start thinking about how many miles there are. The first thing I want you to notice is that this search pattern that we did created a visitation list. We visited the states in the order of increasing number of hops. That's obviously a good thing.

If we can always keep in the agenda the smallest number of hops to the next place, and if we faithfully visit states starting at the minimum number and proceeding up, so we started with A, there's no hops in getting to A, so that's 0. In going from A to B, there's 1 hop. In going from A to D there's 1 hop. In going from A to B to C, ABC, there's 2 hops. So what the algorithm that we described last time-- breadth-first with than dynamic programming does-- is it visits the states in the order of increasing number of hops. That's obviously a good thing.

OK so what's my next slide? I want to think about C being off the map. So what I'd like to do is think about what order would this algorithm visit number of miles? So If I think about replacing the metric in the bottom with the number of miles, whenever the different actions that can occur incur different costs. Think about what I'm saying. So I'm saying that if I were at B and I think about my children ACE, which action I take, go from B to A or go from B to C are go from B to A, which action I take incurs different cost.

So what I want to do now is think about, change the focus from thinking about how many hops is it to thinking about how many miles is it. So now, if I replace the metric-- so over here, the metric was how many hops, replace how many hops with how many miles-- and what you see is that the algorithm is not picking up. It's not visiting the states in order of increasing path length. That's what's wrong. So what we'd like to do is modify the algorithm somehow so that it proceeds through the paths shortest to longest. So that's the goal.

And that's pretty easy to do. The first thing we have to do is put that new information somewhere, and I've already alluded to the fact that the way to think about the new information is that it's associated with actions. It's not associated with states. States are where we go to in the diagram, like state E. The extra cost is not summarized in the state, the extra cost is summarized in the exact path that we took. And the way we'll think about that is incrementally. So we create the past by doing actions and each action has a different cost.

So the first thing we do is associate this additional cost with the actions. Then we are looking for a search procedure that will enumerate the paths in the order of path cost. And the way to do that is to think about the basic ordering schemes that we had before, which were the stack, last in, first out versus the queue, first in, first out. What we'll do is we'll make a trivial modification to the idea of a queue and we'll call it a priority queue.

So a priority queue is basically like a queue, except the things that are queued have priorities. So the idea will be that when you push at possible action, say I had

actions A,B, or C -- in addition to pushing them onto the queue, which is how we would have done breadth-first search, in addition to pushing them on the queue, I'll also associate with them a cost.

So when I push A on the queue, the priority queue, I'll associate with that a cost, which I've arbitrarily said here, the cost is 3. When I push B, I'll associate a cost 6. When I push C, I'll associate a cost 1 so that when I do the first pop, what will come out will be the element that I pushed that has the smallest associated cost. That'll be a way that I can order then my search through the search tree in terms of the minimum costs. Is that clear?

So the first time I do a pop, the element that pops out is the one with the least cost, which is this one, so I get C. C is then removed from the list and the second time I do it, when I do a pop, I pick out the one with the cost 3 which is A. That's an easy modification to the schemes that we used before. Just like before, we can implement a priority queue with a list. Here we've put all the complication into popping. So we just push like we did before, just jam it in. So just add it to the end of the list. And we put the complication into pop, pop looks through the list and finds the one that has the biggest negative cost. That's just because we've got a utility routine that gave us the biggest. We want the smallest. We know that the costs are non-negative, so we implement the find the smallest by calling the routine find the biggest with a negative cost.

So notice that all the new complication is in this pop routine, and if you think about it, pop is doing far too much work. The way this routine works, every time you do a pop, it goes through the whole list looking for something that is small. Obviously it ought to be able to keep track of progress that it's made in that previously. There are much better algorithms, but for the purpose of this illustration, we didn't bother with it. So this is not a very clever implementation. We're not trying to be clever, we're trying to illustrate the point.

So if you were seriously try to do a big search, if you're writing code for Google, you would never do it this way. But the idea again is in abstraction. We're going to bury

those details in the way the queue is implemented and then at the next higher level, we don't need to worry about those details. We'll abstract them away. Is that all clear?

OK. So this then is the way we end up doing the search. So when we do the search, when we create a new node, the idea is going to be that we can generate a cost. Remember, nodes in the search tree summarize paths. Nodes are different from states, right? States are places that we can visit, nodes are paths. So the thing that we need to keep track of now in addition to what we did before, before we had the idea that nodes had states, they have a place where you are currently in the search. They have actions, which is which direction do you go next and they have parents and that was enough information to create and maintain a search tree.

Now what we need to do is also keep track of cost. So we do that by adding instantiation time. When we create a new node, we have to also pass it what is the action cost. So in the previous example, the action cost would be the distance from B to C, for example, being 5, which is different from the distance between B to E, which is 1. So we have to associate at the time we create a node, what is the action cost associated with this new node.

And then the node keeps track of the total path cost, so that's what the red stuff is doing. So it's a very small change to the code that we used for creating nodes in the previous two searches, and then we have to also change the way we do the basic search algorithm. And here too, the idea is pretty simple. It's almost exactly the same thing that we did before, with the idea that we substitute keep track of the agenda with a priority queue rather than with a queue or a stack.

There's one more complication, and that is that in the past, we knew that all children added the same penalty. Because we only keeping track all of how many generations, how many hops are there to the current node, all children were in some sense created equal because we knew they all incurred one more hop. Here, because the children can have different associated action costs, we don't know at the time we've picked up the parent, we don't know at that time which child is going

to end up being the shortest one.

So previously, the goal test was performed when the children were pushed onto the agenda. Here we have to wait until we look at all of the children before we will know which child has the shortest length. And that just means that we take the goal test, which had been inside the 'for a in actions' loop. We have to factor that out and defer until the next time.

So that's the only change to the algorithm that we need to make. So is that clear? So the idea is it's a pretty simple modification of the algorithm that we have so far, but it gives rise to a much more versatile kind of search.

So last time we saw that there was really no good point for not doing the search, the breadth-first search with dynamic programming, we have the same thing here. So when you're doing the uniform costs search, there's no real good reason for not implementing that with dynamic programming, so we also want to think about the dynamic programming algorithm. So you remember in breadth-first search and in depth-first search, dynamic programming referred to the principle that the shortest path from X to Z through Y, so if you have a path from X to Z and you know it goes through Y, the shortest way you can get from X to Z through Y is to add the shortest path from X to Y to the shortest path from Y to Z.

Sounds trivial, but it has a tremendous impact on the number of states that can be omitted from the search. Here, when we do the uniform cost search, we can do the same sort of thing but except that we run into the same sort of problem with not all paths from X to Y are created equally. We don't want to remember any random path from X to Y, we want to remember the best one, which means that in general, we're going to have to expand all of the children before we'll know which child gives the minimum length path.

So that means that the dynamic programming principle that we'll use is to remember the state when it gets expanded, because it only gets expanded after its already being compared to its siblings. So wait until a state, wait until a path has been compared to all the siblings of that particular path before you consider it for

dynamic programming. So we'll do that by not keeping track of states as they are visited, but instead keeping track of states as they are expanded. That's the same idea that we had to do when we had to reorder goal test. Defer the goal task until after you think about all the children. Defer putting it in the dynamic programming list until after you've looked at all the children and so that means that we think about expansions instead of visits.

And so that looks like this. So just like we took the goal test outside the action list, in the previous breadth-first search we did the test goal state in this loop, here we defer it until we've looked at all the children and fetched the parent. So we defer it into the higher loop, we take it out of this moving into this and similarly, we take the dynamic programming memory. We now call it expanded to remind ourselves that what we're doing is keeping track of states after they were expanded, not after they were visited. And updating it again, not when we look at the individual actions, but only after we've decided which trial is the winner. OK?

That's probably confusing. That's not important at this point, because I've written an example and hopefully by thinking through the example, you'll be able to see what the code is supposed to be doing. And then a good exercise is to go through the example, which will be posted on the web with all the gory details, and make sure that you can match up the search-- the partial results of the search-- to the way the algorithm is written. OK?

So now I want to do the problem of interest. Think about what if one of these nodes, one of the states, state C, is very distant relative to all the rest. How will the new search, the uniform cost search, work with this problem? So we do the same sort of thing we did before. Now we have an agenda to keep track of the nodes under consideration. We have a dynamic programming list that is going to be called expanded because we don't add to it until we expand states. It had previously been called visited. And we also keep track of the metric, which is the path costs.

So if we start by putting node A on the agenda, its path cost is 0, because it doesn't cost anything to get there because that's where we started. And we haven't

expanded anybody yet. So notice that the starting state is a little bit different here because we're keeping track of expanded. The expanded list started out with 0 elements in it. Previously where we were keeping track of visits, it started out knowing already what was going on with A.

So now we expand A, think about A's children, B and D, put them on the agenda, and add A to the expanded list. We expanded A so it's time to add it to the dynamic programming list. Then also keep track of the costs of these paths. AB costs 1, AD costs 1. That's the end of the first pass. Now pop the first guy off the queue. Expand it, that means we're expanding the B state. We go from A to B, so we're going to expand B. B Has children A, C and E. A has already been expanded so we don't need to think about A anymore. C and E, so it was A, C and E, C and E have not been expanded, so I expand B to get these two and I associate their total path costs. So the path ABC, ABC has length 6 and the path ABE has length 2.

Then I take the first one off the agenda again, that's AD. I expand AD, which is expanding D. Put that on the expanded list, on the dynamic programming was list. D has children A, E, G. A is already on the expanded list, so that means I have to worry about E and G. Those paths have length 2 and 2.

That was the same so far as the previous search. Now I see something different. Because I'm using a priority queue, I skip ABC because that's just not the right one to do next. So ABC, I'm keeping track of with the priority queue. I know that that path is already length 6. It could end up being the optimum path, but at this phase of the search, it's not the one I should think of next. The one I should think of next is the shortest path. So what I'm trying to do is search the search tree in order of shortest path. So I skip the ABC because its path is long and the minimum length, so back up one. So in the priority queue idea, I want to extract from the queue the first item with the minimum length, so that's ABE. Everyone's with that?

So then I want to expand E. E has children B, D, F, H. So B and D are here, F and H are not, so I add ABEFH to the agenda. Each of those have length 3. So what's the next guy out of the agenda?

**AUDIENCE:** A and E.

**PROFESSOR:** I need the first one with the smallest path, the smallest cost possible. The smallest cost possible is 2, so ADE is the next guy. So I expand ADE, but I've already expanded E. I Don't need to do anything. OK? The dynamic expansion list is keeping track of the nodes I've already expanded. I already did E, there's nothing new to be learned. So that doesn't do anything. So the next one is G. Well I didn't do G yet, so add it to the expanded list. G is shorter than D and H. D was already there, H was not, so add H.

The next one is F. I haven't expanded F yet, so let's do F. So F's children are C, E, I. C is not there, E is there, I is not there, so I add those. So next is this guy, H. H wasn't expended, so I add H to the list. H's children are E, G, I. EG are already there, I is not, so I add that.

Try expanding H, but already did expand H so that doesn't count. Try expanding C, that's fine. Expand C, I haven't expanded C before, add it to the list. C's children are B and F. B and F are both there, didn't add any new children.

**AUDIENCE:** Why'd you expand that C, why not H?

**PROFESSOR:** Oh look at that, you're right. Oh, thank you very much. My slide is wrong. Ignore that, you're absolutely correct, thank you.

So let's say I guess this proves that 200 people watching the lecturer have greater insight-- well anyway. Yes, that's wrong. Don't bother with that because it's got the wrong priority. Jump straight to that. I'll fix the slide on the web. You're absolutely right. I should have gone straight to there and when I tried to expand I, I realize that that's my answer, so I'm done. OK? Questions?

OK it's a little tedious. The point is that it really wasn't very much different from doing breadth-first search. The same ideas still work. The same idea of organizing the search on a tree, looking for the minimum cost answer on a tree, that's the big picture. That's what we want you to know about. I mean, we may ask you a quiz question about breadth-first search or depth-first search or dynamic programming

or whatever, but the big picture, the thing we really want you to know is how to think about a search.

The way to think about a search is a sequence of states organized in a tree that you can then systematically search. That's the idea, and the point of going through the uniform cost search is to see first and foremost, that it fits the same structure, it's the same kind of problem. Secondly, there are very tiny details and so I've tried go over those details. The details have to do with keeping track of the action cost and keeping track of which one to do next by way of a priority queue. But the big picture is the same idea works. States, nodes, trees, search, cues. Questions? Comments? OK.

The other important thing that I want to talk about today is again, this idea of trying to minimize the length of your search. The other point that you're supposed to get from these two lectures is depending on exactly how you set up the search, you can do a lot of work or less work. And we're always interested to do less, especially because if you can do a lot less, you can do a lot harder problem.

So the other thing I want to talk about next is the idea that our searches so far have been starting state centric. What do I mean by that? Every search has a starting state and a goal, and the ordering of our searches so far have been go to the starting state, think of all the steps you can make from the starting state and just keep widening your search wider and wider and wider until you stumble onto the goal. OK, can you all see that that's what we've been doing? So nowhere in our code was the code aware of the goal other than am I there yet?

So the search algorithm so far has been start at the beginning state, ask if I'm there yet, try the closest place I can go, am I there yet? Am I at the goal yet? Try the next closest place I can go from the start, am I at the goal yet? Try the next closest place I can start from, am I there yet? Everything has been starting state centric. Obviously, that's wrong. Somehow, when you do a search, if you were asked to find the shortest route through the interstate highway system from Kansas to Boston, there's a good chance you wouldn't be looking at Wyoming. Everybody knows

enough geography to do that one, right? So the idea is that in the searches we've done so far, you would look at Wyoming before you'd look at Massachusetts. OK? That's stupid, right? Everybody sort of see that? So the searches we've done so far have been starting state centric. So what I'd like to do is think about a way to undo that. But again, to set things up, let's think about what I mean by that. Let's imagine a search very much like what we did before, except let's go from E to I, from Kansas to Boston, and I guess it's Kansas to Tallahassee, but you get the idea. So we start at E and let's think about how our search proceeds. So if you start at E, think about if we push E in the agenda, so I'm doing the uniform cost search with dynamic programing. I'm keeping track of the expanded state as my dynamic programming state. I'm starting at E. The cost of being at E is 0 because that's where I started.

And now, I think about expanding E. So E goes on the expanded list. I think about all the children of E, the children are B,D,F,H. All of those children have distance 1, so when I look among them to figure out the next one that I should do, I do the first guy, EB. B's children are A and C, so I take off EB from the beginning of the agenda. Start again, B. B's children are A, C, E. E is already expanded. Push A and C. Pop D, D's children are A, E, G push A and G because E's already there.

Expand F. F's children are C, E, I. E is already there, push CI. Expand H. H's children are G, I, E, G, I. E is already there, push GI. Expand A. A's children are B and D. They're already there, don't need to do anything. Expand C. C's children are BF. BF are there, I don't need to do anything. Expand A. BD are already there, I don't need to do anything. Expand G. G's children are DH, DH are both there, I don't need to do anything. Expand C. BF, BF nothing. Expand I, I'm there. Yes?

**AUDIENCE:** Were you supposed adding ACE?

**PROFESSOR:** Was I supposed to be adding AC? You add A when you expand it. Ah, yes, yes, yes. Thank you. I'll fix this one, too. So the question was when I expanded A, when I did this one for example, when I tried to expand A, I should have added it to the list here so I don't try to do it again. It wouldn't have affected the outcome, but I should have added it to that list. Thank you. Class -- 2 Freeman -- 0. Yes?

**AUDIENCE:** What if you didn't expand it? I mean, do you consider it [UNINTELLIGIBLE] you should expand it?

**PROFESSOR:** I'm sorry, I didn't hear the question.

**AUDIENCE:** We didn't actually expand it. We considered expanding it, but then we noticed that.

**PROFESSOR:** Correct, correct. So what I really should do is go back and read the code and figure out what I should do. What I'm trying to do is emulate the code. So your point is, it's a technical definition about whether I want to think about whether I actually expanded it or not. If I don't add any children, was it a real expansion? And that's a question that is determined by where was the if statement in the code, and frankly I don't remember. It does not expand it, ah I have the right answer.

**AUDIENCE:** It does expand it.

**PROFESSOR:** It does expand it. So I have the wrong answer. Class -- 2 and a 1/2.

OK so the point of this is that the search was symmetric around E, even though the goal is not. OK? And the question is, how could we fix it so the search is not symmetric around E -- the starting point. How do I fix it so the search is biased toward going toward the answer? And so the way we think about this is with something we called heuristics. So if you think about the searches we've been doing, either breadth-first or depth-first from last time where we counted the number of hops or today where we counted the lengths of paths, each of those considered what thing to do next based on the path from the starting point to the point under consideration.

The idea of a heuristic is to add something that informs the search about how distance, how much distance we're expecting to add from the point under consideration to the goal. So the idea of the heuristic is to put in the second part of the path. The problem with a heuristic is that finding the second part of the path is just as hard as finding the first part of the path, and it would be a terrible idea to -- for every point in the search tree run a new search to find the best answer from that point to the goal -- because that would increase the length of the search time

enormously.

So it's a bad idea. So the problems are of equal complexity, the problem of getting from the start point to the place of interest and the problem of going from the place of interest to the goal are problems of equal complexity. We don't want to try to solve the problem of making the search better informed by increasing the complexity of the search drastically. So that's the issue. So a heuristic is going to be a way of approximating the amount of work we have to do yet, where what we would like it to be is not all that terribly difficult to compute.

So one way we could think about that would be to consider as an approximation to how much work we have to do, the Manhattan distance for example, to complete the path. The Manhattan distance is the sum of the x and y distance. Generally speaking, Manhattan distance is not a good idea for map-like problems because generally you can cut across corners in such searches. In this particular search, since I've excluded cutting across diagonals, since the search space is already Manhattan, thinking about a heuristic that's based on Manhattan distance is probably OK.

So the idea is to develop a heuristic and he what I'm going to think about is, what if I complete the path by adding the Manhattan distance from the point under consideration to the goal. OK, so now if I started at E like before and I'm going toward I like before, then I start with the agenda having just E and having expanded nothing. However, the cost associated with E is no longer 0. The cost of going from E, the starting point to E, the point under consideration is still 0, but I'm estimating the cost of going from E to I by the Manhattan distance between E and I, which is 2. The Manhattan distance between E and I is you have to increment x by 1 and you have to document y by 1.

So instead of saying that the cost of state E is 0, I'm saying that it's 2. Is that clear why I'm doing that? So now I expand E, I think about the children B,D,F,H -- BDFH, and those children now are not the same cost. Even though it costs the same amount to go from E to each of its children, going from its children to the goal, going

from each child to the goal does not cost the same amount.

If I were to go from E to B, that's a cost of 1. But the Manhattan distance from B to I is 3. So I'm estimating, then, that the cost of making the decision go from E to B is 4. The real cost of going from E to B plus the estimated cost of going from B to I. Similarly, if I go from E to D, the Manhattan distance is 3, so that's a length 4. If I go from E to F, Manhattan distance from F to I is just 1, so the total cost is just 2.

So now rather than circling out from the starting point, my next step is biased toward the goal. So my next step is biased toward so the remaining items, the smallest cost is 2, so the next place that I expand is EF. F's children are C, E, I. E is already here, so I only think of CI. C and I also have different costs, So the cost of going EFC, EFC, the direct cost is 2 and the estimated distance, the Manhattan distance between C and I is also 2, so that's a cost 4 whereas the cost of EFI, EFI has a direct cost of 2 and the Manhattan distance from I to I is 0.

So now I look for the minimum distance that's remaining, so that's going to be H. I expand H, the children are E, G, I. So E is already here. I think about GI. Same sort of deal, some of them are short, some of them are long and as I proceed to the search, I very quickly find I without having ever looked in the wrong direction, or at least having looked minimally in the wrong direction. Is that clear?

So the idea in a heuristic is to add an estimate of how much it will cost to complete the path so that you bias the search toward the goal. Rather than making circles that spiral out from the starting point, make the spirals biased toward the goal. And here's the way you do that. It's very easy. All you do is every place we would have looked at cost before, add the heuristic function. So you have to add a heuristic function, that's the part that's hard. The code, given the heuristic function, is easy. The really hard part is actually figuring out what a reasonable heuristic is.

The reason that's hard is that you have to be careful not to miss the solution. So the heuristic function can't be bigger than the actual distance. OK, why is that? The agenda is trying to keep track of all the possible places that you could look next. If your heuristic function makes the next step look too big, it'll be taken out of the

agenda and never appear again, and you'll never find that path.

So when you're making a heuristic function, you have to be very careful not to ever overestimate the distance from where you are to the goal. If you ever overestimate it, then you can exclude forever more, so think about the agenda as a pruning operation. We did this starting last time. When we think about pruning, we pop off things from the agenda. We say -- don't ever need to look there, don't need to look there. We pruned it. If you have a heuristic that it is too big you can prune a path that was the answer. So you have to be careful never to do that.

So it's asymmetrical. If you were to put in a heuristic that is too small, that causes you to underestimate the penalty of going in the wrong direction. So if you said, I'm trying to go from Kansas to Boston and you inadvertently said that Wyoming really didn't cost anything, then you would not necessarily exclude the correct answer, but you would include and cause the search to consider a necessary path.

So the idea is that you would like the heuristic to be the same as the real cost or smaller. You don't want to end up solving another search problem in order to calculate it, because that will increase the cost of doing the search too much. So you would like some number that's easy to calculate that has the guarantee that it will always be less than or equal to the actual cost, and that's the art of doing a heuristic.

If you satisfy those, then that previous algorithm that we looked at, which is -- in the literature it's call the A-Star Algorithm for historical reasons. That algorithm, if you obey these rules for finding admission an admissible heuristic, if you find an admissible heuristic, then the A-Star search will be a lot faster and still find a solution with the same length.

OK, so now just to see if you're following what I'm saying, here's a question to ask yourself. Remember the tiles problem? The tiles problem is the first problem that I did last time. The idea was, imagine starting in this configuration 12345678. Move one tile at a time by moving the tile into the free spot, so I could move 8 to the right or 6 down. Keep doing that until you perturb this configuration into this

configuration.

We saw last time that there are a large number of states-- there's a third of a million states-- so this is a big search problem, even though it's something you've almost certainly also solved as a child. And in fact, my version, it was the same as yours, I'm sure, was a 4-by-4. I did the 15 puzzle, not the eight puzzle, but it's the same idea.

So what I want to do is consider heuristics. Consider three heuristics. Heuristic A is 0. It always returns 0. That's an easy heuristic to calculate, right? Heuristic B is sum the number of tiles that are in the wrong position. What is heuristic B for this state?

**AUDIENCE:** 8?

**PROFESSOR:** 8 -- there's 8 tiles that are out of position. So heuristic C is the sum of the Manhattan Distances required to move each title to their respective locations and then calculate two partial sums. Consider MI to be the number of moves and the best solution if you use heuristic I and EI is the number of states that are expanded while you're doing the search. If you use heuristic I, which of the following statements are true? OK, take a minute, talk to your neighbor, figure it out which of these are true.

[CLASS TALKING]

**PROFESSOR:** OK so what's the smallest numbered correct answer? The smallest numbered correct answer. Oh, come on. Volunteer. Explain it with your neighbor. OK, very good. The smallest numbered correct answer is (1). MA equals MB equals MC. Why is that? How can I prove that MA equals MB equals MC. What do I have to do? Yes?

**AUDIENCE:** You'll have to reach the shortest possible solution.

**PROFESSOR:** So under what conditions will they all reach the same length solution?

**AUDIENCE:** They're all doing backflips.

**PROFESSOR:** They're all doing backflips. There's another condition. Yes?

**AUDIENCE:** That they're all either exactly [INAUDIBLE].

**PROFESSOR:** So the heuristics, all three heuristics have to be admissible, which means that they have to be non-negative numbers. To be admissible, a heuristic has to be non-negative. And it has to generate an answer that's smaller than the actual number of moves necessary to complete the path.

So we have to prove that these are all admissible. So, are they all non-negative? Yes. Are they all less than or equal to the number? So let's do that. Let's first of all get an answer. So we'll do that after we do the second part. What's the second smallest correct statement? Let me see if I can answer it, yes? OK, the answer is 4. So, how do you know that EA is bigger than EB is bigger than EC?

So the number of states expanded has to do with the size of the heuristic. If the size of the heuristic is 0, that's the same as not using a heuristic. That will search all possible states in a breadth-first search fashion. If the heuristic is anything that's bigger, the number of searches will go down. What's the greater than or equal to thing doing here? Yes?

**AUDIENCE:** Number (3) is technically also true, because if (1) is true, then (3) has to be true.

**PROFESSOR:** If the number (3) is technically the number M. [LAUGHTER] OK, OK, OK. [APPLAUSE] OK, class, three plus. Freeman, oh well. Yes, you're right. Yes, I agree. Did you raise your hand for (3)? So (3) is technically the second, yes that's right, that's right.

OK so moving on. Number (5) -- the same best solution will result for all the heuristics -- true or false? Come on, things can only go downhill for me. So the same best solution will result for all of the heuristics? How could it possibly be false? Right? Didn't we already said MA was equal to MB equals MC. Yeah?

**AUDIENCE:** It's the same amount of moves, but maybe not exactly the same solutions. So if there are multiple solutions of the same length, the difference heuristics don't have

to give you the same solution exactly. They have to give you solutions with the same length. So what happens with the heuristics is you perturb the order of search. If you perturb the order of search, the only thing that is proved is that you get a minimum length solution, not the same minimum length solution. This particular problem has lots of solutions and so you don't necessarily get the same solution when you use different heuristics, OK?

So the final point is that the addition of the heuristics can be extremely effective. If you run this problem with our search algorithm, you always get solutions with 22 moves in them. That's good because all the heuristics were admissible so you always get a right answer -- a shortest answer. But the number of visited and expanded are drastically different when you add the heuristics.

So if you use here heuristic A, which is equivalent to no heuristic, you end up visiting 170,000 states to find this answer, where if you used the Manhattan distance to the goal, the sum of the Manhattan distances, you do a very small fraction of that. So the point is that this stuff matters, especially when you do a higher-dimension search, which is all of the searches that we'll will be interested in.

So the idea is that the order really does matter and with that, I'll conclude with a reminder that tomorrow evening is the makeup/retake day for Nano Quizzes. Please come to the lab if you'd like to make up or retake the Nano Quiz.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011