**PROFESSOR:** Today I'd like to talk to you about some techniques that you can add to basic search to enable your systems to make more intelligent decisions and save computation time. Last time, we introduced basic search and the basic idea of how to encode search so that our system can use search when encountering an unknown territory or state space. And today, we're going to review some things you can do in terms of using the information that you know and also using estimations of the information that you would like to know to improve your chances of discovering the path that you're interested in the fastest.

The first thing that we can do in order to improve our search is use dynamic programming. Dynamic programming refers to the idea is that once you've done a computation for a particular kind of problem, you can save that computation and use it later, as opposed to having to engage in that computation a second time. The way this manifests in search is that once you visited a particular state, you don't have to visit that state again. Because the way your agenda's set up, you found the fastest way to that particular state, as far as you're concerned.

So the general idea is that once you visit a particular state, you don't have to visit it again. If you're building a normal search tree, it can be difficult to keep track of where you've been, especially if you're doing something like that depth-first search. To get around this and to enable dynamic programming, the easiest thing to do is just keep a list of the states that you visited as a consequence of this particular run of search.

I'm going to demonstrate dynamic programming by running depth-first search on our state transition diagram from last time. The first two steps are the same, except for the fact that we're going to keep track of the fact that we visited both A, B, and C as a consequence of the first two iterations of search. If I'm running depth-first search, my agenda acts as a stack, which means I'm going to take the partial path that I added most recently to the agenda, pop it off, and expand the last node in the partial path.

When I expand C, I'm going to visit B and D. However, B is already in my list of states that I visited, because it's already in one of the partial paths in my agenda. So I'm not going to visit B again. Instead, I'm only going to add one new partial path to my agenda, ACD.

I'm also going to add D to my list of visited states. If I run another iteration of depth-first search, I find my goal. For completeness, E is added to list of visited states. This took even fewer iterations that the original depth-first search. We didn't waste time expanding different nodes.

We also used less space. In the general sense, the concept of dynamic programming is great to use whatever you can. And in search, it can save you a lot of time and energy.

Another way we can intelligently improve our search technique is by making considerations for costs that we know that are associated with particular transitions. In this state transition diagram, I've indicated some costs associated with the transitions between particular states. If we know the costs associated with transitions, then we can use a cumulation of the weights accumulated through traversing a particular partial path to prioritize which paths we're going to explore first in an effort to reduce the amount of cost associated with our final actions. In order to do that, we can keep track of the value associated with that cumulation, and sort the agenda at every iteration based on that cost.

If we're using dynamic programming, while making considerations for both cost and heuristics, and also when we're running the goal test when making considerations for cost and heuristics, we're actually going to make our considerations when we expand the node as opposed to when we visit the node. This difference is very important because it provides us with the most optimal solution. If it's possible for us to visit the goal node, but it's 100 cost units away, It may be worth our while to search for alternatives that provide a much shorter path to the goal node. That's why we switch from considering when we visit to considering when we expand.

Let's look at uniform cost search run with dynamic programming. In my first step, I expand A and add two partial paths to my queue. I'm also going to keep track of the cumulative cost associated with that partial path. When I expand A, I add A to the agenda.

Note that I haven't talked about stacks or queues or depth first or bread first or anything. We're working with a priority queue. And what that means is that things with a higher priority float to the top. Or things with the lowest cost associated with them we're going to consider first.

This means that I'm going to expand B in the partial path AB first. I'll add it to my list. When I expand B, B has two child nodes, D and C. ABC has partial path cost 3 associated with it. And ABD has partial path cost 11 associated with it.

That means when I reprioritize my queue, I'm going to end up sorting everything such that AC comes first, ABC come second, And ABD comes third. Previously, with our strategy for dynamic programming, C would not have been added to this partial path, because we've already visited it with path AC. At this step, we're going to expand the path AC, add C to the list, and any other time that we end up visiting C, we will not add it to our paths.

If I expand C, I have a transition to B, which is already in my list, and a transition to D that has cost 7. ABC is going to float to the top of my priority queue. ACB is not going to be considered, because B is already part of my list. And ABD is going to remain with cost 11.

At this point, you might say, but Kendra, why am I considering partial path ABC when C is in my list-- or C should be in my list, excuse me-- as a consequence of expanding the partial path AC? Even though we've expanded the partial path AC, if we have not made any considerations to weed out our list at every iteration, this is still going to float to the top.

And we're still going to have to deal with it, even though we've already expanded C. Since we've already expanded C, we're going to ignore this partial path and just

move ACD 7 and ABD 11 up to the front.

D is not in our expanded list. When we expand D, we have one child node E. Because we're working with cost and heuristics, we do not actually evaluate the goal test when we visit a node. We evaluate it when we expand a node.

So I am going to add ACDE to my agenda. It's going to have cost 8. And ABD11 is still going to hang out here at the back of the priority queue.

At this point, I get to expand E. I skipped adding D to the expanded list when I expanded it from ACD to ACDE. At this point, I'm going to expand E. And the first thing I'm going to do is test and see whether or not it passes the goal test. At that point, I stop search, return that I successfully completed the search, and that my partial path is going to be ACDE.

That covers uniform cost search. At this point, you might say, Kendra, this is bearing a lot of similarity to things like maps. And I would really like to be able to use my knowledge of things like the Euclidean in order to make more intelligent, even more intelligent decisions about where it is that I go with myself.

And I would say yes, you should be able to. In fact, people do. In fact, people do all the time.

They say well, some thing's this far away as the crow flies. So I know that if I've gone further than that at any point, then I've wasted some amount of time. But it represents a good underestimate of the distance that I'm going to cover.

This is the basic concept of heuristics. If you're attempting to find a goal, and you have an estimate for the remaining cost, but you know it's not exactly right, you can still use that information to attempt to save you an amount of computation or amount of search. In particular, you probably shouldn't be using a heuristic, if you know it's perfect. Because if you know the heuristic is perfect, then you should be using the heuristic to solve your problems, instead of doing search in the first place. Or if the heuristic already tells you how long it's going to take to find something, then it probably also has the path that represents how long or that represents that

amount of cost.

If you want to use a heuristic effectively, you have to make sure that your heuristic represents a non-strict underestimate of the amount of cost that is left over.

And what do I mean by that? I mean that if you have a heuristic, and you're using it as a thing to tell you whether or not you're wasting your time, if your heuristic represents information that is bogus or says this particular path has more cost associated with it than it actually does, then it will lead you astray. Or you don't want to use a heuristic that will prevent you from using a path that actually costs less than the heuristic advertises.

This is what's known as an admissible heuristic. An admissible heuristic always underestimates if it makes an error in estimation about your total distance to the goal.

All right. So at this point, I've covered dynamic programming. I've covered costs. I've covered heuristics. And it turns out, you can use all of these techniques at the same time. When you use both cost and heuristics, in combination, while evaluating your priority queue, that's known as an A-Star search. And you'll see a decent amount of literature on A-Star.

This covers all of the intelligent improvements to basic search that I will talk about in this course. We hope you enjoyed 6.01.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011