JOE LEAVITT: Good afternoon. My name is Joe Leavitt. My group members are Ben Ayton, Jess Noss, Erlend Harbitz, Jake Barnwell, and Sam Pinto, and we're going to introduce you to the topic incremental path planning. Listed on screen are just some of the references we used in putting together this lecture. The top three are the main references we used in developing our material on the D* Lite algorithm, but also take note of the bottom reference, which is a paper done within Professor William's group here at MIT. It talks about all-pairs, shortest paths incremental search method.

So if you're interested in more incremental search methods than those we're going to talk about today, please get into this. Especially if you're going to go ahead and do our P set, these references will be very helpful to you.

So today, we're going to first Monday problem, then we're going to introduce the idea of incremental search. Then we're going to go into detail on the D* Lite algorithm, which is type of incremental path planning, an algorithm used for incremental path planning, run through an example of D* Lite, talk about when it is good to use incremental path planning when other path planners might be better, and then I'll go into some algorithm extensions and related topics, and some applications in mobile robotics.

So in order to motivate the problem, we're going to use a robot living in a grid world. It's at a start location trying to get to a goal location. It can move up, down, left, or right, or diagonally. And it can only see the eight cells around it, as it's moving through this grid world. And we're going to give some pretty fine map of the world. This is a map that has the world before it starts moving.

And it's possibly moving through some of these, new obstacles could appear, or obstacles could go away. And the idea is that it's going to make observations as it's moving to this world and update its plane to get to the goal based on what it sees as it's moving along.

So I must correct myself. The map that you're seeing now is the map that the robot has ahead

of time. The previous map is what the environment actually looks like. So as it starts moving, it's going to make a plan to get from the start to the goal, and that's what we want it to do. And it's going to move to the next position and then realize that there's an obstacle in the path that it previously planned, and now it must replan in order to find a valid path to the goal.

And so now, it replans, and continues to move on the new plan, and now it finds another obstacle that it has to replan around, and it does the same thing. And then it continues to walk through until it gets to a point where we have a change in the environment. The change is represented by these two cells going black here, and now its original plane is no longer valid again. And what we want it to do is to replan to find a new valid optimal path to the goal.

This is just showing here what happened to our environment. And then the robot replans and continues to march towards the goal So this is the behavior we want from a mobile robot, whether it's a two-dimensional path-planning problem, or we're talking about multi-dimensional robot arms, planning, working with humans on a manufacturing environment, or an activity-planning process. The idea is that we want the robot to be able to account for new obstacles or changes in its representation of the environment and continue to plan on the fly quickly and optimally.

So there's methods that accomplish this. One method used to in path planning is rapidly exploring random trees, or RRTs. RRTs are basically going to plan, replan, on every time step based on changes in the environment. And one problem in my experience here, though, is that RRTs are suboptimal, even though fast.

See, here you see the robot has come up with a plan. It's seeing new obstacles, the table in front of it. The coffee cup. And its planning for those things, so the plan it came up with is clearly not optimal.

So we can perform some additional computation. We can use a method that builds upon the RRT, RRT*, that it finds an optimal path, but that requires a pretty significant amount of offline computation time, and then you can accomplish something that looks a little more normal. Again, we have to allow the robot to perform a lot of the computation off-line.

So how do we accomplish what we want to? We want to compute quickly. We want to compute optimally, and then have the robot continue to move as it's received new information coming to the environment.

Some problems we're facing, there's changing environmental conditions. There's sensor limitations. You might not be able to see everything in the environment around you, and then we have limited computation time. And the solution that we're going to explore here is to reuse that from the previous search, in order to speed up the search process, and continue to plan optimally.

And the method we're going to use to accomplish this is incremental search. So now, we're going to talk about what incremental search means. First to show you kind of how powerful incremental search can be, this is one example of an incremental search method, incremental all-pairs shortest paths, which allows you to online compute paths between any start and goal location.

The idea is that you have a robot that's trying to move around one obstacle. That's what you see in the upper sequence of events there. And another obstacle is added, and now it has to plan around this new obstacle. In using this algorithm, you have a significant increase in performance and computation time using the all-pairs shortest paths versus an RRT. RRT can act as another version of an RRT, where you're planning basically from both the goal and start and connecting your paths.

It's on the order of what you get with RRT, but this RRT connect algorithm doesn't give you an optimal result. And then it's even fast than another algorithm probablistic road mapping, where you're randomly sampling the environment, and connecting points to produce a path. So the idea here is that we can really leverage this idea of reusing results from previous searches to accomplish fast replanning online.

So the whole idea of incremental search is that we're going to form a normal graph search, just like anybody's used to. And then we're going to repeat as the robot moves through the environment, or you're executing your plan, you're going to execute the next point in the plan, or the next action in the plan. We're going to receive changes from the environment, either through our sensors or information from the outside. And then we're going to use that information to update our previous search results instead of completely replanning, and then continue to form that loop, until you reach your goal.

So before we get into too many of the details of how we're going to accomplish this, first, we want to review the graph search problem. So in graph search, you have a graph consisting of vertices or nodes. We're going to use nodes throughout this lecture to talk about the vertices

or nodes in the graph that are connected by edges.

And then, there's an edge weighting function that describes the cost going from one node to another. And then, a heuristic function is an estimate of the cost to get from a current node to the goal node. And then we a start vertex and a gold vertex.

In the graph search problem, we're also going to have this idea of a g value, or a cost so far. We're going to use term g value, which is essentially the cost that it's taken to get to some predecessor node, plus the cost of the node that you're actually expanding.

So you would have this W start to S1 cost to get from start to S1. And then the g value for S goal would be the weight from S1 of the goal plus the S1's g value.

Yes, go ahead.

AUDIENCE: What is W?

JOE LEAVITT: W is the weighting function that I talked about previously here. So the weighting function describes the cost to get from one node to another along any edge in a graph. And then we'd use those waiting functions to develop the idea of the G value.

So W is just the edge cost. And then our heuristic value, like, we use our heuristic function to establish a heuristic value from one goal to another, and we're going to call that our h value, or h. And then the total estimated cost to get from a node that we're currently expanding in a graph search is f, which is your cost so far, plus your estimated cost to go. I think I might have messed that up in there. Call g value associated cost so far, not cost to go, and then h is an estimate of cost to go.

So in order to reason about a real-world environment, we first have to take that environment and convert it to a graph, so we can form graph search, and then do everything we're talking about. Some of the examples, we're going to talk about, and some of the ones we've already talked about, we're using this notion of a grid world. You can have a four connected graph or a grid, where a robot can move up, down, left, or right, basically in x or y directions, or an eight connected graph, where you can also move along the diagonals.

And then had here, we just showed the graphs associated with those, with the nodes and edges. And then you would have an associated admissible heuristic with those if you're going to form a heuristic search.

So just know up front that any time we're moving from a real world environment, that we're trying to reason about these search methods, we have to develop an associated graph. And one more thing that is important as far as understanding incremental searches, is the notion or idea of relaxation.

We're going use Dijkstra's algorithm. Everybody here familiar with Dijkstra's algorithm, I'm assuming, in the past. We're going to use Dijkstra's algorithm to kind of illustrate the idea of relaxation and remind everybody of what that means, because it's an important concept when we're talking about incremental search algorithms.

So essentially, in Dijkstra's algorithm, it's a single source, shortest path algorithm, a best first search. So we're going to make eight expanding nodes according to their best g value, or cost so far. And we have our start node initialized to zero cost so far, because at your start, haven't accrued any cost. And then, every other node is going to be initialized to infinity. And the idea is we're going to find the shortest distance to all of the other nodes by relaxing the cost to those nodes from infinity down to their actual shortest or lowest cost value.

So we start by expanding the start node, and then you find the cost so far to each of the children nodes of the start node. So you have an edge way to one, and so the node on the bottom as a value of 1. Similarly for the middle node at 5, and the top node for 3.

Now, the next node we're going to expand, can anybody tell me what the next node we're going to expand based on performing best first search here using cost so far?

AUDIENCE: The 1? 1?

JOE LEAVITT: Yes. So the node labeled 1 here, that has a g value of 1 is the lowest out of 1, 3, and 5, and infinity. So we're going to go ahead and expand 1 next. And now, we find that using this value of 1 plus 3 equals 4, which is less than what we previously had here as 4. So now the center node has been relaxed from a value of 5 down to 4.

So now the shortest path we found to the center node is 4. And we're going to continue to perform this by going through each node in order of the lowest cost. And similarly, you saw how the top node was relaxed to 6 by going through 1, 3, and 2, where as previously, it had a lowest cost of 7 going through 3 and 4. And if we format through the whole graph until all nodes have been unexpanded, then we've relaxed every node in the graph to its shortest path value.

And then, we can just do a greedy search from a chosen goal node, back through the graph to find shortest path.

AUDIENCE: Is the term relaxation standard? Because the fact that you're bringing numbers from infinity double to small value sounds more like tightening as opposed to relaxation. So I was just wondering if people use it as like, a standard--

JOE LEAVITT: I'm not sure if it is in all areas, but I have seen it in multiple places. When you're talking about graph search, specifically with respect to Dijkstra's algorithm, that is where the term relaxation is used. And we're going to use that as we go throughout and talk about incremental search. We're kind of repairing the results because of new edge weights and things like that. We're going to figure out which nodes we have to relax back down to their shortest value to be able to find the new shortest path.

So that's the term we're going to use here if it's not used universally. I can't say for certain. But we will use relaxation to talk about reducing-- it's kind of related to-- think about like, a tension spring. As you reduce the constraints on it, it'll slowly relax to its completely relaxed position. That's kind of the idea here.

So now that we have talked about the ideas of relaxation and graph search, how do we reuse the results from a previous search in a new search field to employ an incremental search method where you have done a search, you found the shortest path, and now, you had some changes to your graph, and you want to use your previous results to come up with your new shortest path.

So the way we're going to do that, is we're going to store our optimal results from a previous search. This can be done in a number of ways. It's going to depend on the algorithm. But you can soar like, a list of shortest paths as an incremental all-pairs shortest paths problem. The one we're going to talk most about is storing g values, and that's what's done in a D* Lite algorithm, where you're finding the shortest distance from any given node that you've expanded so far in your search to the goal node.

And then, you'll use that data to find out, find the shortest path. And then when you go to perform your next iteration, you'll look for inconsistencies in that graph. And then in order to find a new shortest path, we'll leverage that previous data and the inconsistencies for relaxations to come up with a new optimal solution.

And that's all we're showing here. On the right side of these two graphics is you have the initial search. You have an obstacle that is now in a place that was wasn't there previously, which causes you to update edge weights in the graph associated with this grid world. And then using these values which represent the g values in the graph search, you only update the ones you need to update to find a new optimal path. And so completely reperforming our search.

So what incremental search methods exist, now that we've kind of talked about what they are and what we can do with them? So the whole idea of this slide here is to show you that incremental search can be used across many domains. It's not just specific to path planning. It's used in temporal planning to determine temporal consistency. It can be used propositional satisfiability. Instead of every time you're updating another algorithm, you can use your previous results.

And the one we're going to focus on today is D* Lite, which is specific to mobile and robots. But the ideas and concepts we're going to talk about there are applicable elsewhere, and you can use to help you learn about and expand your knowledge in other incremental search.

So the D* incremental path planning approach. The whole idea behind it is that we're going to take the idea of incremental search that we just talked it about, and we're going to combine it with heuristic search, which is an optimal solution. And the idea is that two things are orthogonal so we can combine them, and then we can form efficient incremental path planning, getting both an optimal result, and quickly getting a new path when things change in the environment without having to completely replan.

And so what that looks like, when we have a grid world, which is shown in four instances here up on the board, and we're trying to plan from a start node to a goal node, and showing each of these sections. And in the upper left or on the left axis, we have-- whether we're performing a complete search or an incremental search. And on the horizontal axis, whether it's an uninformed search or a heuristic search.

So the upper left is a complete search, just using best-first search and no heuristics. And what it's showing is all the nodes that are expanded, and trying to find a path from the start to the goal. In the upper right is a heuristic search, A*. Employing the heuristic, we're able to collapse the number of nodes expanded by quite a bit.

In the bottom left is an incremental search method that's using uninformed search or best-first search as an underlying search algorithm, so it's going to expand the same nodes as the best-first search. And then in the bottom right, you see an incremental heuristic search, the lifelong planning A* algorithm, which is kind of the baseline for the D* Lite we're going to talk about, that the initial search will expand the same nodes as the heuristic search, since it's using A* as its baseline.

So now let's say the robot moves. We get an update to the environment. There's a bunch of changed edges. And now best-first search, as you can see, is going to have to expand pretty much the same number, if not more nodes, then our previous search. And A* also is going to expand about the same number of nodes as it did previously.

The incremental search method using best-first search, by using previous results, reduces significantly the number of nodes that needs to be expanded. And then, lifelong planning A*, the incremental search plus the heuristic search, reduces that number by even more.

Even when there's quite a few changes in the environment, you have very few nodes that you need to expand on the next search in order to find a new optimal path. With that, I'll turn it over to Ben to talk about the D* Lite algorithm.

BEN AYTON: Thanks. So I'll be walking you through the D* Light algorithm today. Now as we're emphasizing, the D* Lite is not the only incremental algorithm, as we mentioned previously. But it is a widely used incremental algorithm, and you'll see it's quite efficient in what it does.

D* Lite is an algorithm that searches from a single start node to a single goal node. But as we see changes occur in the path along the way, we'll adjust that. So it's nice to think about D* Lite from two perspectives. One is to think about it as a modification of the principle space.

Now, you'll see that D* Lite in its first iteration or its first run through before any modifications behaves very similar to A*, and that's where it's nice to keep that main model in mind. And you'll see that in an example further along.

The second perspective to look at A* is from the perspective of Dijkstra's algorithm, from which we see many of the principles of relaxation that we covered beforehand. From this perspective, we see that when we do reparations to the graph or changes to the graph, our methods of repairs the graph is essentially to relax down our changed edge weights until they reach their truth.

Now, where we differ from Dijkstra's algorithm with something like D* Lite is selecting only the nodes that we want to hit efficiently. So whereas Dijkstra's algorithm tried to fit the entire graph, we only want to travel from a single start node to a single goal node. And so D* Lite behaves like Dijkstra's, but a guided Dijkstra's.

So I'd like to just remind you of several concepts from A* first, because it's easier to introduce them in that context. And then we'll modify them a little bit. So Joe mentioned before that for A*, we're searching from a start node to a goal node. We're sorting the nodes in our queue by total cost, which is the sum of what we're calling g, which are costs to get to a single node, and our heuristic, which is the cost to get from a node, s, to the goal node, your s.

Now, what we don't usually consider with A*, but is very important for D*, is how to distinguish between ties when two different nodes on the search cube have the same total cost. So here's what we're doing is we're introducing a couplet, which consists of two values. The first, which I'm calling f1 here for any given state, is the same as what we're usually used to. It's the linear combination of your cost to reach the goal node and heuristic function.

But we also introduced a second value for this couplet that's used in the case for only those to draw. And that is simply the cost to go, that g value. So the order of expansion of nodes from the q will order them in terms of f1 first, and if two nodes at the front of the cube tie in terms of their f1 value, we select the node with the lowest g value.

So looking at our graph down here, we're looking at our states s1 here and s2 here. Can anyone tell me the total cost for our node s1 which, remember, is a couplet of two different values?

5, 3. So five, yes, is our cost to reach this node one, plus the heuristic, which is 2 here. And also, 3 is our second value, because it's only our g value there. And so what would the value be by the same logic for s2 down here?

AUDIENCE: 5, 2.

BEN AYTON: 5, 2. That's exactly right. And so, bearing in mind what I told you before, where should we take off the q first?

AUDIENCE: S2?

BEN AYTON: That's exactly right, because the first value, f1, for these two nodes is exactly the same. But

the second value is lower is f(s)2.

I'd also like to introduce the concept of successors and predecessors of any given node. In general, a graph consists of directed edges. And we could have a graph with undirected edges, but we can think of as just any edge being directed in both direction. So here, we define the concept of a successor of a node, which is every node can be reached from a given node, along edges that can be found.

So the successors of this given red node are these two blue nodes here. We also introduced the concept of predecessors, which is every node from which that node can be reached, meaning nodes from which we can follow our directed edge to our node. So for our red node, the predecessors are these two nodes.

What I'd like to emphasize here is comparing the two, that this node here was both the processor and the successor of our red node. And that's fine. That occurs when we have undirected edges or edges that direct in both directions.

So think about D* Lite in the sense that is a repeated best-first search, which is where the A* principles come in through a graph with changing edge weights as that graph is traversed, meaning that as we travel from our start node to our goal node along a path that we are planning, we can see that edge weights are changing.

And in this example, I'm modeling an obstacle coming in place between this node here and the goal, which means that we have removed that edge. And that's effectively the same as changing an edge weight. We're just modeling it as changing this edge weight to infinity, which means that edge can't be traveled along.

And as I mentioned before, we also view this as replanning through relaxation of path costs. So once we have moved to this point, how do we find the path from here to here when this edge has been removed, essentially?

So we want to make D* Lite efficient. And it's a repeated search through the graph as we traverse it. So you'll see that since we're using these g values, if we maintain a formulation where we're measuring the distance to go or to reach a node from our start node, this is not preserved when we move our start node. And we move our start node when we're updating the position of our vehicle.

So let me show this through an example here. The g value, or cost to get from one start node to our current node, s, here is the combination of these two edges. And so, it sums to 4. Now when we move along that path that we've planned to this next node, the g value to get to that same node is here too. So this isn't preserved. And that means that we'd have to potentially reformulate things in our cue, which would be very inefficient.

So what we do is we reformulate our search. And it's perfectly valid to reformulate our search to search in the opposite direction. We're essentially measuring for our g values now the cost to reach the goal from a specific. And instead of heuristics, then measure the costs to get from this specific node, or to get from the start node to our specific node. And so we search from the goal backwards through the graph.

In this case, we see that regardless of whether our start node, we preserve our g values, and our cost to travel from s to that goal. That's what we like, and we we'll keep that reformulation to maximize efficiency. So here, I want to give you a kind of overall understanding of the D* Lite algorithm before we dive into specifics of what it's doing.

First we want to initialize all nodes as unexpanded. And what exactly, it means expand a node neighbor or cover. And then we're doing a best-first search from the goal node to the start node until the start node is consistent with its neighbors. And I'll again say what that means later.

We then move our start node to the next best vertex, essentially our node moving along the plan that we created. And then we have to see if any of those edge costs change. We track how our heuristics change, and then update the source nodes of those changed edges. And we essentially repeat this process until, again, we've converged to a solution, and we keep moving.

So this best-first search, the A* Lite part of the algorithm, is where most of the computation is going to be occurring. However, the incremental component is actually how can we adjust our edge costs to make sure that we're using most of the information that we previously gathered as efficiently as possible.

So I'm going to walk you through various steps of this. First, I'd like to walk you through this step of moving to the next best vertex, just so you understand that. So we can extract a path, given the path costs that have, through this simple argument, which says that we map the successor, by which I mean we restate what the start node is, because we're moving from one

state to the next state, which becomes the new start node, by this form, which essentially says that we're taking the best path in the sense that we're minimizing the total cost that would be gained traveling by that path to this node.

So in a sense, the g value to get to this node, to the red from the green, or color stretching back through the graph, by this path here, is 7. But to get to this node from the lower node is 10. And so our g value that is optimal is 7 in this case, and know that we pick the successor as the green node up here.

So now to walk through some principles that appear in both of these steps. I want to show how we handle weight changes locally. And we do this because we don't want to track weight changes throughout the entire graph. We want to handle this efficiently and separate them, so we only need to handle the changes that really impact our problem.

So this is the same formula that defines our successor, or our g values for a given node. But we can see that this may no longer hold when edge weights change. The example that I have here is that I've changed edge weight cost from 6 to 1. And so now we can reach the red node, searching back through graph by a new optimal path, coming down from this node here, which would make our new g value five instead.

So this hasn't been preserved. And then these changes then propagate to our predecessors, because the costs for the processors are each dependent on the g value for their successor. And so when we hit a node, we have to propagate that change all the way back through the graph.

So to do this efficiently, we take an A* like approach, where we update the lowest cost nodes first, and we don't expand a node until it is the next lowest cost. To help us do this, we're going to store additional fact. We'll call this rhs. Now the meaning behind rhs, it comes from the term right-hand side, which in the paper world, was first introduced and made more sense than it does so here.

What you can think of your rhs value as is essentially you're corrected your g costs. When edge costs change, how G should be signed so that it's correct for a different graph. And when your rhs value is not equal to your g value, that means that we have what is called a local inconsistency. So given the logic that your rhs value should be what your g value would be corrected to be, what would your rhs value be for this graph here?

AUDIENCE: 5?

BEN AYTON: 5, yes. So this is the same graph that I presented previously, and I walked through how g should be 5. rhs should be 5 here. And what we're tracking is rhs and g differently. So g is what it should be, but we haven't got the dated g to be 5 yet.

So we have two different types of local consistencies that we distinguish between. We have what we call local overconsistency, where g is greater than your rhs value. This is going to behave more similarly to Dijkstra's algorithm, where the value that we're associating to any given node now is higher than it should be, and we're relaxing down to the true value.

However, we have a lovely underconsistent case that we have to handle slightly differently. So now I'm going to walk through updating and expanding nodes. What I mean by updating a node is recomputing the rhs value, and then placing that node on the priority queue, if that node is locally inconsistent.

So to give an example here, I've changed this value, moving from 6 at the bottom to 1, and recomputing rhs, as we did beforehand. Now we see that the node is locally inconsistent. In this case, it's locally underconsistent. And so we place that on the priority queue.

The priority queue values are based on this new formula. This new formula essentially handles how we see G and rhs. In essence, we want to take the minimum value of one of these, so that we can handle it the first time that it will cause changes that propagate through the graph.

And so this is essentially our ordering on our priority queue which is essentially the same coupling that we saw for A*. But here, we just have g of s replaced by the minimum of g of s and rhs of s. And so, here your minimum of g and rhs is 5, plus a heuristic, leads to a total cost associated with this node of 9, 5.

So we expand our five prior nodes by then taking them off the priority queue and changing g. Now I said here that we have an inconsistent. And so, we update this in much the way that we would expect. Your g values are relaxing down to your rhs values, and so we just set g to be equal to rhs.

In this case, our node is now locally consistent, and it's going to stay that way. So can anyone think of some good reasons why I go through the process of changing rhs and keeping track of that new value, only to put on the queue, and then take it off again, instead of just recomputing what g of s should be in the first place? Does anyone have any ideas?

AUDIENCE: [INAUDIBLE]

BEN AYTON: What you can think of is that you're putting something on the queue. If we update that value immediately, there are multiple nodes that could change the same node that we just considered. By, which I mean more work changes as they propagate back through the graph, could lead to changes in rhs value of that specific node.

And so, what we would need to do if we can recompute g of s every time, is put it on the queue to signal that we're recomputing g of s, and do that recomputation, and then new changes we've done, perform that again, do that again and again and again and again. And we want to avoid that situation.

So what we're going to do is we're going to keep our rhs as essentially a temp value that can be adjusted when it's in the queue. So when something is on the queue, it's on there once, and it can be updated and taken off. But we know when it's been taken off, that it's been taken for the correct time, and won't need to be handled again.

AUDIENCE: So if I understand this correctly, so if you had a single edge cost that we changed, then you could just do what you just said, in terms of, you could just update the thing, and probably should be predecessor rate.

BEN AYTON: Yes. So if you knew for sure that nothing else was coming back and down, then we could just do that change and propagate to all the predecessors, because nothing upstream of that node is affected.

AUDIENCE: So am I understanding correctly-- so you're basically running the programming, because you're keeping the costs to goal to devote, and then what you're doing is every time you have an edge that changes to a value that could potentially make that node better than it was before, so if it changes the value of that node to something better than it was before, you would change that node, and then queue all the predecessors, because those be changed as well.

BEN AYTON: Yes.

AUDIENCE: I mean, if those nodes do not change, you don't do anything. You're moving from the queue. They also change a better bell and keep propagating until you have nothing else to

[INAUDIBLE].

BEN AYTON: Yeah. But we only do the propagation, or we queue changes that occur earlier in the graph, but we only actually perform the change resetting the g value, when it's expanded.

AUDIENCE: And then the reason for that is for efficiency when you have more than one edge changing?

BEN AYTON: Yes.

AUDIENCE: So you could basically have multiple paths affecting the same node. And in this situation, if you actually allow your changes will occur and multiple edges, that's when the importance of the queue comes.

BEN AYTON: That's exactly.

AUDIENCE: Which is actually, it prevents you from redoing the same work over and over again.

BEN AYTON: That's exactly right. Because if another change came through, then we'd put all the predecessors back in the queue again, and we'd do all of our updates for all of them as well. We're avoiding that process.

AUDIENCE: And the queue ensures that we're only updating-- and I think that'll cover it. But in the manner in which we terminate, we don't necessarily-- we don't expand every node in the queue. We only expand those we need in order to find the shortest, and then we retain the queue for that search in case we need to update those in the next round of the search.

BEN AYTON: Right.

So I have to correct a mistake in my language that I said before. I said that the node in my examples that I gave before was highly consistent, [INAUDIBLE] wasn't consistent. I apologize for that mistake. Other consistency is the easy case that we're handling here, where when we expand that node, we have to take g of s and set that to be equal to the rhs value for expansion.

And then we propagate that effect to all of the predecessors at that point, which means we update the predecessors and put those on the queue if it hasn't had an impact on them. This means that the node is not low key inconsistent, and it's going to remain that way in that fashion.

The underconsistent case, which was the opposite of what I showed, is the more difficult case. In this case, the old path cost was better than the new rhs value. Rhs value has increased to above [INAUDIBLE]. In this case, we can no longer relax down to that g value from our Dijkstra's algorithm respective. So we have to think of a new way to handle this.

What we do is we essentially set our g value to be equal to infinity. And this is an extra step that's going to cause the node to go back on the queue an additional time. But what this mechanism assures is that that node goes back on the queue at most one additional time, which means that for D* Lite, we only ever examine any node at most twice on the queue.

Once we've set g of s to be equal to infinity, then we can essentially relax down to the rhs value from the perspective that we had before. So we set g of s to be equal to infinity, and we update all the predecessors of s and s itself this time. As I was saying before, we need to put that node node back on the cue so that we handle it when rhs has hit its minimal value.

And this means that that node is now going to locally consistent or overconsistent. And we handle overconsistency in the case that I've shown you before. So to give you an idea of how exactly this works, I'm showing you an example here. We start off with this graph, and we introduce two changes. We have increased this edge cost to 5. And so rhs has now increased to a total of 10. But I've also changed a value here for edges that you can't can see. But assume that the rhs value of this node has also changed.

So now we can put all of these nodes on a cable ordered in this way, computed using the formula that we've seen for it. We expand the node s1 first. And because it's underconsistent, we set g to be equal to infinity. Then we propagate to all predecessors and s1 itself, meaning those go back on the queue. Now, assuming that node changes of any relevance happen in these nodes, we recompute the costs for s1, which has now changed to 14, 10, because we're using the minimum of infinity and 10, which is 10 plus 14, going behind the node s2.

Next, we expand this node s2. And because that was an overconsistent case, we can just set g to be equal to our rhs value. We then have to propagate that to its predecessors, which includes s1, which has updated its rhs value.

So if we didn't do this, we wouldn't have handled this effect correctly. When we set our rhs value here, we then have to put this node back on the queue. So previously, s1 was already on the queue, but the effect has to be propagated to this node, which puts it on the queue again, which would mean adjusting the values, because rhs has been reduced by 1. Both the

costs here have been reduced by 1.

Then we expand that node. We have an overconsistent case. We're setting g to be equal to rhs. And finally, we've completed our graph search problem. So now what we have to handle is an additional complication that occurs between several steps, tracking how our heuristics have changed.

So we want to carry over prior queue, which was what was mentioned previously, between our different searches as our stock node moves. But the problem is that we're moving from our start node to a different point. Recall that our heuristic value is measured from any given node to the stock node. So when our stock node has changed, the heuristic value is not going to be the same value to the new stock mode.

So we want what's already on the queue to be comparable to what is being computed for our new value, so we can use the math and the algebra that we've already done. So how we handle this is we introduce something called the key modifier, where we move from the previous start node, which here I indicated as s last, to the new start node, all the heuristics for admissible heuristic are lowered by at most a heuristic from the last start node to the new start node.

So now, when we add new nodes to the queue, instead of subtracting that value from everything that's already on the queue, what we do is we just increase for all nodes that we put on the queue, their values, by this new modifier. Because essentially when we're taking off of the queue, all that matters is the relative differences between the two. And so, instead of subtracting a value from everything on the queue, adding that value to all new values that occur on the queue will achieve the same purpose.

And so our key modifier is initialized to 0 at the start of the algorithm, and is increased every time that we update our start by this new value, by the heuristic between the last start node and the new start node. And then we update our total cost to include the edition of the key modifier fire for our f1 term.

So now we've covered all of the D* Lite algorithm basics. And we can actually walk through the slide that I showed you before with the math. So when we start, we initialize all of our g values to be equal to infinity, and all of our rhs values to be equal to infinity, except at the goal node. And that's to ensure that we always have a node to start from our search.

We best-first search from the goal node to the start node until that's locally consistent and expanded. We should now know what that means. And that signal that we have found our best path, we move according to our movement rule, and then if any of our edge costs have changed, we update our key modifier, and update our rhs and queue positions for the source nodes of change in edge costs, meaning the nodes from which those edges originated.

And then, we simply repeat from two, following best-first through the principles that I've walked you through for overconsistent and underconsistent nodes. And as a reminder down here, we always search or sort by this very long couplet of two values.

So now, we're going to walk through an example of [INAUDIBLE].

JESS NOSS: OK. So again, here's the pseudo code that Ben was just presenting. And we're going to go through a quick example with five nodes, so this isn't going to show all of the properties of why you would want to use incremental path planning, because when you have only five nodes, you're going to end updating a lot of them a lot of the time. But after the example, you'll see how it makes more of a difference on a larger graph.

So in this graph, we're going to have prior nodes that go A, B, C, D, and G is our goal node. The robot will start at node A, and its goal is to find the shortest path to node G, except that the graph might change as the robot goes through, or the robot might gain new information.

So initially, the robot thinks that every node is available. And it knows that the path lengths are all 1, except for from D to G, the path length is 10. So this is a bi-directional graph. Every edge goes both directions. And the heuristic, as Ben described, will be with respect to the start node. In this case, we'll just say that the heuristic is the number of nodes to the start node. So we're ignoring the fact that this edge length is 10.

So first, we'll initialize all the g and rhs values. So we have infinity at every node, except for the goal node, which has an rhs value of 0. So this means that all of the nodes are correctly locally consistent, because their g and rhs values are the same, except the goal node. So we put the goal node on the queue using the formula that is on Ben's slide, which I'll put here for reference. So the key is going to be these two terms.

First, we have the minimum of g and rhs plus the heuristic value plus the key modifier, which is initially 0. And then we have the second term, which is just the minimum, g of rhs. So you can see how we would get 3, 0 here, because the 3 comes from the heuristic, and everything else

was 0, so we just added a bunch of 0's.

So that's everything what happens for the initialization. And now we have one node on the queue. So we'll have to dequeue first. This is an easy question.

AUDIENCE: The node on the queue.

JESS NOSS: The one node on the queue. So we dequeue G. So we're going out to update its g value. So as Ben said, the rhs value is sort of a look ahead. You can also think of rhs as being the shortest path that we found so far, and G is the guaranteed shortest path to that node. So this one is trivial. The shortest path from this node to itself has length 0, because its the same node.

So now we're going to update the rhs values and its neighbors. So it has two neighbors, C and D, and in each case, the rhs value will be the minimum-- this is not actual math, but its the minimum of the edge cost, the d value of a neighbor plus the edge cost. Which is the same thing as the edge weight or the weight w.

So we rhs is 10 here, because it's the distance from g to do, and then it's one, which is G to C. So again, we can calculate keys for those. So now we'll dequeue another node. What should we dequeue next?

AUDIENCE: C.

JESS NOSS: C. Because it has a smaller key. So we dequeue C. We update its g value. So this is saying, now we know that the shortest path to C actually has length 1. And we can also compare this to A* star algorithm by drawing a search tree. So in the beginning, we had A*.

In the beginning, we just had node G, because we were searching from G backward through the goal. And then we expanded that back to get C and D. And then A* to keep track of the path length to that node plus the heuristic at each node. So the pathway to C was 1 plus the heuristic 2, give us the value of 3. So if we go back a moment, we can see-- is that OK?

So you can see that this 3 is the same as the first point of the key, and path length one is the same as the second part of the key. And then similarly at D, we have the path length, which is 10, plus the 2, give us 12. So this 12 is the 12 here, and then the 10 is the path length.

And so you can see how if we were doing this the A*, you would also dequeue C next. OK. So

now, we dequeue C, and we're going to update the rhs and its neighbors. So B is pretty simple. Previously, it's rhs was infinity. Now the shortest path to B that we've found so far has length 2. So here we have B was a path length 2, plus a heuristic of 1 gives us 3.

But what about D? Previously, its rhs value was 10, but do we now have a shorter path to D, given that we're expanding C? Or is 10 still the shortest path?

AUDIENCE: That is the shortest path, through C.

JESS NOSS: Yeah. So now that we're expanding C, we see that you can actually get to D in only two units instead of 10 units. So that's better. So we'll update the rhs value to 2. And that means that we're going to change D's key accordingly. And the way that maps to A* is, so if you already have the path length is 2, plus the heuristic is 2, gives us 4. And the fact that we're changing the key to 4, 2 means that we're never going to expand this D, because A* uses an extended set, so it would always expand this D with a shorter cost before this one. So that's indicated by-- removed by changing the key on the queue.

So what do we do next? Which node do we dequeue?

AUDIENCE: B.

JESS NOSS: B? Yeah, because it has the smaller key. So we'll dequeue B. We'll update its g value, which is again saying that we know now that the shortest path to B is 2. So we dequeue B, and then we expand it. How many neighbors does B have?

AUDIENCE: Three.

JESS NOSS: Three. So we're actually expanding to all three of those neighbors. We have A, C, and D. Can we still see this? We can. So at A, it's pretty simple. We have the rhs value is 3, because it's the shortest path through GCB to A. So that's 3, plus the heuristic 0 plus 3.

What about-- oh, see, it doesn't change, because-- you could go through B and back to C, but-- oh, C shouldn't actually be on here. Sorry. C shouldn't be on here, because we don't want a loop in our path.

OK. So at D, can we do better than this path length of 2?

AUDIENCE: No.

JESS NOSS: No. Because what happened was originally, we had the path length of 10 through G, and then we had the path length of 2, 3 C, and this path through B is not as good. So we're going to keep the same rhs value and key at D. And we can see that in a lot of he A*, where we would have, if we did do GCBD, we would have a pathway of 3, plus the heuristic of 2 is 5. But we would never actually want to expand that D, because it's not as good as this one.

OK. Now what would we dequeue? Looks like we have two nodes in our queue currently.

JOE LEAVITT: A.

JESS NOSS: A. So we dequeue A. That's where we trying to get to, because that's the current start node. So we set the g value to 3, which means that that's actually the shortest path. And we're done. We found the shortest path. Well, we're done, except that the path might have to change. So one thing you notice here is that we never have to actually set D's g value. So it's possible, maybe there's still a shorter path to D, but it doesn't matter. So now the robot will move to its next best place, which is B, and we'll update the heuristics accordingly, because this is our new start node.

So now, again, our heuristic is the number of nodes away from that green node. And we're keeping the same items on the queue from before. But we also have to update the key modifier. So now what happens is an obstacle appears. So there's an obstacle at C that the robot can now see because it's moved closer to node C, but it couldn't see it before.

So now, we're going to indicate that by keeping all the edges, but changing their rates to infinity. And that means that the robot could try to get to see, but it wouldn't ever get there. So what we do next in the algorithm is we're going to update all of the rhs values associated with the source nodes of these edges. And in this case, the edges go both directions, so all four of these nodes are affected by the infinities.

So let's start with node C. What would be its new rhs value, given that rhs is the minimum from one of its neighbors to it of the g plus edge cost?

AUDIENCE: Infinity.

JESS NOSS: Infinity. because all of the edge costs going to C are infinity. So what would be the key? The tricky part here is the first part of the key is the minimum of g and rhs. So the minimum of g or rhs is what?

AUDIENCE: So it'd be 3, 1? Yeah.

JESS NOSS: Yeah, 3, 1, because the minimum of g and rhs is g, which is 1. The heuristic is 1, and the key modifier is 1. OK. How about node B? What's the new rhs value? So it has a neighbor here with g as infinity, a neighbor here of g is 1, except the edge length is infinity. And then this guy with g is 3.

AUDIENCE: So it's infinity, 4?

JESS NOSS: 4. Yeah, because currently, we still think that you can get to A and B units. So this is actually going to be 4. So this is one of the sillier things that D* does, but luckily, it doesn't do this very much, so it doesn't actually really cost too much time.

And then how about D? What would be the new rhs value?

AUDIENCE: 5, 3? rhs.

JESS NOSS: So it has been 3 neighbors. This edge length is infinity is useless, so that's useless. This one has g of 0 plus 10 would be 10. Yes, so this is the best one. G is 2, plus edge length of 1. So that would give us 3. So now what do we dequeue?

AUDIENCE: C?

JESS NOSS: C. So this is one of those tie breaking cases, where both of them have the first value is 3, but C has a better second value in its key, so we'll dequeue C. Is this over or under consistent? So the definitions of over and under consistent that we had earlier, we had if g is greater than rhs, then that's overconsistent, and we set g equal to rhs when we're updating g. So that was the case that we've seen so far. But then if g is less than rhs, it's underconsistent, and we set g to infinity. So what is this?

AUDIENCE: Underconsistent.

JESS NOSS: Underconsistent. So the reason intuitively why we're going to set g to infinity is because we're saying there's some sort of contradiction here. This one doesn't make any sense. It's less than the rhs value, so we're going to just reset g and sort of start over with this node. So we set that to infinity.

And in this case, we don't actually need to add the node back onto the queue, because it's now consistent. It's now locally consistent. So next, we'll dequeue the one with the smallest

key, which is B. B is also underconsistent, so we're going to set its g value back to infinity. So this is sort of resetting the weird thing we did earlier that didn't make any sense.

We're going to update its neighbors' rhs values. So what would be the new rhs value for A, now that we know that we don't actually have a path B?

AUDIENCE: Infinity?

JESS NOSS: Infinity. Yeah. And then, how about rhs value to D? Because currently, D got its rhs value by going from A to B to D.

AUDIENCE: 10.

JESS NOSS: 10. 'Cause now the only way-- rhs is 10. The key would have that one also. Is that right? Yeah. Because currently, the best path to D, and actually, the best path to D you can see is from G to D, because you can't go through C anymore.

So now, we'll dequeue node A, because it has the smallest key. It was also underconsistent, so we would set G to infinity. So we'll propagate that to its neighbors. And if we continue doing this, then we find-- eventually, we dequeue this node, and in this case, we had to dequeue everything, so now the node's are locally consistent. So what's the best path that we've found from B to the goal?

AUDIENCE: B, D, G?

JESS NOSS: B, D, G. And that's also the only path in this case. And we had to update these neighbors. OK. So here's our new shortest path. Now, the robot will move to node D. But now, what's going to happen-- I think we updated our heuristics. But now what happens is it turns out it's a moving obstacle, and it starts following the robot. So now the obstacle has moved to there. So as humans, we can see that, obviously, the shortest path is through C, but the algorithm doesn't know this yet.

So what would we do next? So we just updated our edge weights. So now all the edges to B are infinity. But some of the edges to C are no longer infinity. So we need to update the rhs values accordingly. So we'll update all of the rhs values that were affected by this change. In this case, G's rhs value never changes because the distance from G to itself is always 0. And this one doesn't change because we still use the G value here.

OK. So now again, we can start dequeuing and planning a new path, so we'll dequeue this node. We'll propagate to its neighbors. Dequeue that one. Update the rhs values. But in this case, we don't-- let me go back step. So we dequeue this node, and we update its G value, but we don't actually need to continue dequeuing things. We should-- oh, this rhs value should be updated. Oh, it is updated. Why is this rhs infinity?

AUDIENCE: All edges to it are infinity.

JESS NOSS: Yeah. All edges to it are infinity. You can't get there. But we don't actually have to bother expanding B, because we've already found the shortest path. So this gives you sort of a hint at when D* might be efficient. Because even if there were a whole bunch of nodes over there, I wouldn't need to consider them, because we found the shortest path.

So now, the robot can move up to C. And let's say the obstacle chases the robot again. Do we need to find a new path? No. So intuitively, we don't, because the obstacle's not in the way of our current best path. But we can also see algorithmically why that's the case.

So first of all, we'll update all of our rhs values accordingly, because we've changed all of the edge weights. But the reason algorithmically why we don't actually need to consider any of these change values is because this node has the smallest key, which means that this is the node that has the shortest path to G anyway. It is smaller key than all the other nodes that are on the queue. So dequeuing things wouldn't actually help us find the shorter path. So the robot can move to the goal, and we've succeeded.

Any questions on that? OK. So now then we'll talk about sometimes when you would want to use incremental path planning, other than this five node graph.

ERLEND HARBITZ: Thanks. So as Jess said, there are sometimes good reasons to use incremental path planning, but there are also good reasons not to in certain circumstances. In certain circumstances, we can actually formulate graph problems, where a problem is worse by using incremental planning.

Remember that I said that D* Lite puts any node on the queue at most twice. And so we can compare that to A*, or something like that, that does full replanning whenever a search occurs. That pus every node on the search queue at most once. So D* Lite is most beneficial when putting the nodes on the queue, potentially more than once, leads to less nodes being examined overall.

So here is essentially what I just said. So A* might perform better for certain problems if we only have to consider a small amount of nodes anyway. So putting nodes on the graph twice would actually lead to more expansions. And so, to get at an intuition of why that might occur if changes are close to the start node, I want to show you an example here.

So here, we're actually showing an example that's derived from [INAUDIBLE]. So we're searching forwards here from the start node to the goal node. And that's just a different reformulation, so don't let that confuse you too much. When we walk forward through A* with a good heuristic, we might move directly from the start node to goal node, and find that path.

Now, if a change occurs somewhere upstream of the start node, when we replanning, we essentially have to replan the sections that are beyond the change that's occurred. So everything that's here, to an extent, it's relatively insensitive to this change. It could occur that we would have to take an entirely different route. But for the most part, it's likely that changes are going to couple to our existing path and so forth. And so we have an efficient search that will just go around from the path that we've already planned to our goal node.

So in this sense, we have an example where incremental path planning would be very efficient. In this case, which is a different one, we introduce an obstacle very near our start node. This means that most of our existing path is to charge. So where we start out from our start node to our goal node, we have to do a lot of recomputation from scratch.

And so an incremental searching algorithm is going to have to undo all searches that we've already done, and then search through almost the entirety of the graph again. So intuitively, in this case, just restarting with A* from a blank slate, which we would search directly down to the goal, instead of doing reparations that would send us in one path, would be a better idea in this case.

So since we're powering from the start to the goal, in this example that we're showing, it's worse to use incremental path planning algorithms when your changes happen near your start node. For something like D* Lite, we search back from the goal node to the start node, it's better if changes occur nearer to the start node, which for our vehicles that we normally see that have limited sensor horizons, is what's going to occur in the most case. So in those type of situations, incremental path planning is going to be very efficient.

AUDIENCE: So I understand how this could be a problem for LPA* and D* Lite over finding a single path,

where you have like a single source or destination or something like that. But what if it's incremental, all-pairs shortest-paths, where you've computed-- I'd assume you wouldn't experience the same problems if your algorithm was computing choice paths between all pairs of nodes indirect.

ERLEND HARBITZ: Yeah. It already exists. Well, that depends. When you're redoing recomputation, between all the nodes in the path, it depends if our obstacle has changed [INAUDIBLE] of something there that we to conceive. And by that I mean imagining this graph we've computed, D, the paths and path costs, to get from every single node to every single other node.

Now, if introducing an obstacle here only changed the path cost to get from our stock node to our goal node from example, then replanning that specific pair of paths from start node to the goal node would use data that already exists from the other nodes to the goal node. We already have that.

But if introducing an obstacle would cause changes throughout the entire graph, then we can potentially create what would have to be a very well-connected graph, where again, incremental path planning would probably perform worse in that case.

AUDIENCE: Yeah. So you could still end up in the same situation [INAUDIBLE].

ERLEND HARBITZ: Now these algorithms are interesting, because they allow for some extensions that allow us to do some some very different things quite easily. One example is greedy mapping, in terms of how can we design algorithm using what we know about incremental path planning, to hit every single node on our graph. And what we do is then we introduce a faux goal node, essentially a node that doesn't exist that we can never reach.

And we add to our initial beliefs, our initial understanding of the graph connections between all nodes to this faux goal node. Now, it's false in the sense that whenever we move our start node to one of the nodes that we've examined before, what happens in terms of the changes of the edge costs is removing that edge that connected that node to the goal node.

So what this means in principle is that our vehicle, our poor vehicle, can never reach the goal node. Whenever it reaches a node that it thinks it can get to the goal node from, that path disappears. But what that causes it to do is when it's replanning, it finds a path to a node that we haven't visited before, because it thinks there is a path that leads from that node to the goal node.

And so this means that it's efficient, but replanning a root from his current understanding to a node that it hasn't examined before. And so it can apply something like the D* Lite or LPA* with this formalism in order to search through the graph in its entirety and make sure that in a greedily efficient manner, we hit every single node in the graph.

AUDIENCE: Is this [INAUDIBLE] TSP problem, or the traveling solutions problem?

ERLEND HARBITZ: I'm not very familiar with the TSP problem.

AUDIENCE: You try to-- and you try to cover all the Cs, and try to come up with a route that is the most efficient, and try to visit every known C value once.

ERLEND HARBITZ: OK. Then in that case, yes. It's not going to be globally efficient. As you said, it's going to be an approximation for it. In a sense, from each node that you visited, you're searching for the next best choice. So it's possible you can corner yourself. But it's a way that we use what we already implemented and the methods that we've already established to tackle a different type of problem.

AUDIENCE: And all this [INAUDIBLE] usually if you have a strategy which is visit the closest unvisited node, instead of making this whole incremental path from this study. If your policy is, always move to the closest unvisited node, how worse does that get compared to this?

ERLEND HARBITZ: I think it would behave quite similarly. What we're just trying to show here is that we can implement that problem as an extension if this happened. It may be an efficient way to do it.

AUDIENCE: [INAUDIBLE] probably like, zigzag.

ERLEND HARBITZ: Also, if you're just going to the nearest connected node, if you corner yourself as we showed here, then you could potentially run into issues if your algorithm doesn't then do an extended search for it. If it only considers nodes in its immediate neighborhood that hasn't visited, once it corners itself and doesn't know where to go.

AUDIENCE: Just keep going through the graph until you hit the next inhibited node, assuming that you know how many invocations you have to do.

AUDIENCE: Sort of like the random walk type approach?

AUDIENCE: Well, I mean, kind of like the greedy version of it. You kind of start from a node, and then you keep searching, you'll see, what is the next unvisited node, and you pick that. Kind of like-- [INAUDIBLE] some you see what you have covered. You see what's the next best move. You make that move. And you're done. [INAUDIBLE] So you probably don't have this [INAUDIBLE].

ERLEND HARBITZ: This basically ensures you're going to find the shortest path back to the next unvisited node, I guess. We also can use these incremental path planning algorithms for anytime planners. And the benefit of an anytime planner is that we quickly reach a plan which is suboptimal. And then from there, we are repairing this plan to approach optimality.

And this is useful in the real world where you want to get a solution that comes out quickly, and we want to start along our solution, and then repair it as we go. And once we start it, we need o save some more time if your reparations might lead to small changes and that type of thing. And this can be handled quite efficiently using an incremental path planning.

So what happens is we apply something, idea, called inflated heuristics. And so these inflated heuristics are essentially increasing our heuristic values that exist on our graph, in our true graph, by a cost of scale factor. And this is typically very large, so differences in heuristics become very large.

This means that once we start along a path, we continue along it. And that causes us to very quickly find the first path that actually works. Then, we can view reducing those heuristics back to their original values as essentially a form of changing our [INAUDIBLE] causes or-- so it's a form of replanning. And then by applying the principles of an incremental planner, we can efficiently use the data that we already have in order to perform the search and repair our solution as we go.

Finally, we want to make explicit to you guys how we can actually apply these algorithms to mobile robotics. Because it may not immediately clear how we go from a full space to a graph. Joe on this kind of briefly when he talked about we discretize the world. But we want to give you guys some ideas of how we might do this cleverly.

So in order to build a graph system out of our world state, we differentiate between holonomic systems and nonholonomic systems, which their difference is just in terms of whether your constraints are differential in nature or not.

Then we are going to outline three methods here very briefly, cell decomposition, using a

visibility graph, and sampling-based construction methods. So for cell decomposition methods, we might to do what you would expect, take the free space in this region and split it up into various different cells. And each of those cells is going to represent a node in our graph. And how we can do this is we split the region up based on each of the vertices of obstacles.

So we draw a line vertically from each of our vertices that goes in the direction away from the obstacle. So at the top here, it's going up and out of the obstacle. But in the lower left corner there, both up and down, don't intercept the obstacles. We draw them in its entirely.

So we can continue this throughout the entirety of the state space, and then we can attribute to each essential cell that we've defined through these regions, a single point that represents the midpoint of that cell, which is going to be our graph node, because we attribute a single set of coordinates to each of our graph nodes.

And we also, to make sure that our transitions between these cells are consistent, additional nodes at the midpoint of each of these lines that we've drawn, where we make the distinction that from here, we've actually drawn two lines. So we draw the midpoint between all of those. This allows us to essentially create a graph through the entire system, and we connect each node at the center of any given cell to nodes that are on the boundary of any given cell.

This allows us to create a full graph through the system, that we can move through, guaranteeing that none of these paths, if we follow them exactly collide with the obstacles. Now, this works in environments where the obstacles are 2D polygons, and the path is far from optimal, as we can see, but we can plan an optimal path over the graph that we have defined. And this only works on holonomic systems.

We can also define what is called a visibility graph, where we essentially draw a boundary between both our vehicle down there at that bottom left, and boundaries around our obstacles. These boundaries are sufficiently sized so that if the center of the vehicle that we're defining is along the extended edge of the obstacle, that the vehicle will not collide with that obstacle.

Then for each of the vertices of the extended boundary that we've defined, we add in a new node, and we draw all lines between all nodes, which can be connected without intersecting the extended obstacles, which we've drawn here. And this leads to a graph that we can, again, plan over.

This, again, works in environments where the obstacles are 2D polygons. But what's nice

about this system is that we do get an optimal path, provided that we assume that we assume that our extension of the robot itself is valid, or doesn't act too much-- isn't too excessive for constraint.

Finally, going back to some of the random sampling methods that we mentioned at the very beginning of this lecture, we've introduced sampling-based roadmap construction, wherein we choose nodes randomly or deterministically throughout our environment. The difference here, though, is that we then create a graph out of this system, and that we reason over that graph using something like an incremental planning algorithm.

And so when we create that graph, after we've sampled, we can use the [INAUDIBLE] neighbors of each node, or we can connect each node to every node within a certain radius, which is what we've drawn here.

Finally, in order to actually move between nodes, we need to have some knowledge about the dynamics of the vehicle, which is what we're discussing here. And one method that we can do this for nonholonomic systems is we can distinguish between different paths that use turning right, turning left, and traveling in straight lines.

It turns out that the shortest path between two points can be defined using one of these systems. And so we just have to search over a finite number of options in order to find the best path that can connect any two nodes. Hopefully, that has given you some intuition into how we can apply these incremental path planning [INAUDIBLE] systems. Are there any questions?

[APPLAUSE]