MITOCW | Lecture 14 | MIT 6.832 Underactuated Robotics, Spring 2009

RUSS TEDRAKE: OK. Welcome back. Well, sorry, I wasn't here. It's more dramatic for me, I guess, than if I had been here on Tuesday. I hope John did a good job covering for me. I'm sure he did.

So you've learned about walking robots. You've learned about lots of robots. You learned a handful of very powerful techniques, I think, for designing motions for these robots.

Some of my favorite these days are things like the dir call type methods. So let's consider the problem of taking some robot described by our standard, some non-linear dynamics form, and taking that robot from an initial condition x0 and getting it to some final condition. I don't even care about the time. So you're just getting it to some other state, x goal.

So what tools have we already got that could help us drive this system from some initial condition to some goal state? The answer is a lot of tools, right? If the state space is small enough, you could imagine designing an optimal control cost function that we could do dynamic programming on. That would get us to that goal if we only rewarded ourselves from being away from the goal.

We could do a shooting method, right? If we're using our SQP, our SNOPT type interface to the shooting methods, then we could provide a final value constraint, which says I want the endpoint of my trajectory to be exactly at the goal. And that would solve this problem in some cases.

But both of those methods have problems, right? The DP doesn't scale. The shooting methods scale nicely. But if you ask your shooting method to drive you from x0 to x goal, there's some chance it's going to just say, I can't do it, right? I can't satisfy that constraint because it's based on a local gradient optimization.

And if there's local minima, SNOPT will just say, I can't satisfy the constraint x at time final equals x goal, happens a lot. If you're looking at hard enough, interesting enough, problems, it'll happen to you, OK? So what do you do? Do you just say, oh, find a new research problem, tell my advisor I've got to find something else?

OK, there's more things you can try. The methods I want to talk about today are the feasible motion planning algorithms. So motion planning is a very broad term. It's used all the time in robotics.

I'd say roughly everything we've done to date has been a motion planning algorithm. It's like saying it's a control algorithm. It's a very general term.

Some people debate whether an algorithm is a motion planning algorithm or an optimal control algorithm or whatever. I think that's sort of a waste of time. I would say they're all motion planning algorithms.

Most of the things we've talked about so far I would call optimal motion planning, where we actually had a cost function flying around to do it. Oftentimes, motion plans go from a start to a goal. Oftentimes, they're open loop, but that's not really defining. You can do feedback motion planning. Motion planing is just a very general term saying I'm designing the motions of my machine.

The reason I want to use motion planning in the title for this lecture and next lecture is because there's an important class of motion planning algorithms that are not covered by the optimal control algorithms. And that's these feasible motion planning algorithms.

So the optimal motion planning algorithms try to get me from x0 to x goal in some way that's good as scored by some cost function. Feasible motion planning algorithms aspire to do less. They just say, I'll get any trajectory that gets me from the start to the goal. Because they surface their claims on optimality, oftentimes these algorithms will work for more complicated systems.

So the message I want to sort of deliver today is that there are a lot of good feasible motion planning algorithms out there in the world. We'll talk about some of the most exciting ones, I think. And actually, even if you care about optimality, even if deep down in your core you say this is a cost function that I must optimize for my life to be complete, you still should care about feasible motion planning algorithms because they can do something like seed a dir call method, let's say, and get it out of local minima.

So let's say I have an Acrobot that I'm trying to swing up. And I've got a table right here, so I can't swing that way. And whatever my initial guess at the trajectory, the system keeps banging up against this. It can't figure out a way to turn around and go back the opposite way to get to the top. Or maybe there's a table that you have to go through just at a certain way.

I would say, if you're shooting method is having trouble finding its way to the goal, maybe you should back off on optimality. Just find any old trajectory that gets to the goal. And then take that result of that motion planning algorithm, hand it to dir call, and let it then do the rest of the optimization.

We'll say more about this. I just wanted to give you some context about why we want to talk about this extra set of algorithms. In general, we're going to expect these to scale to slightly higher dimensional systems than the optimal control algorithms we've talked about so far and to be more global than the shooting methods.

There's a great book on all of this stuff called-- I guess it should be underlined-*Planning Algorithms.* So I just spelled planning wrong. So I'm completely sleep deprived.

If I do stupid things like that, call them out. I'll fix them. Something about kid number two, the first kid stays awake while the other kid sleeps and vice versa. It just makes it that much more interesting. Yeah.

And *Planning Algorithms,* the book by Steve LaValle is actually-- Steve's got it on his website, the full version. You can download for free. So it's nice accessible text that you can check out if you like it.

OK. So before we get into our first feasible motion planning algorithm, let me give you just a little sort of culture. The motion planning algorithms, I think, actually grew up not so much in the controls community, but in more the artificial intelligence community. And there's a reason for that.

There are serious people out there that think the only thing there is to intelligence is the ability to do search. So I think that artificial intelligence via search, to some extent, sounds ridiculous. But it's pretty hard to disprove and some people really think it. But that's the way it happened.

So let me give you some cultural examples of where that came from. Some of the original efforts in artificial intelligence, they thought, my computer program will be intelligent if it can play a game like checkers or chess. And as early as the '50s, Art Samuel was building checker playing robots that are basically doing search in order to try to figure out how to beat their opponent in chess-- or in checkers, sorry.

Checkers-- Samuel in the '50s. Chess, we can leap forward to IBM's Deep Blue, let's say in the last decade. These are computer programs that, at times, can play with the sophistication of a trained human.

And they're based on simply having the rules of the game programmed, some databases of moves that they've stored up, a lot of them in the Deep Blue case, and search. Finding the way to move my robot from x0 to x goal you can imagine being very, very analogous to taking my computer game and finding from an initial board configuration to a winning board configuration. They're exactly the same problem.

So because of things like computer games, the artificial intelligence community really started building up these search algorithms. And they were very fundamental. The other big push, I think, from the AI community was in ideas like theorem proving.

So an AI zealot of 4 years ago might have told you that mathematics would be obsolete by now. Because all you have to do is program the basic rules of mathematics into a set of rules that your computer program could evaluate, and then proving whether something was true or not was just a matter of finding the right chain of small logical deductions to put together to be able to prove that p equals np or something like that.

OK. And actually, I joke, but that's actually a very powerful thing that works today. If you've use Mathematica or Maple and you've asked it to simplify, that's a result of theorem proving work that people did in artificial intelligence a long time ago. And again, it's a matter of designing the rules of the game, which is a collection, in that case, of mathematical primitives, and finding a path from your initial knowledge to some proof, let's say.

But even going beyond that, there's people out there that-- my favorite instance of this is there's a guy, David Lenat, who's somewhere in Texas right now typing-- well, he's got teams of people typing random factoids into computers. He's got this project called Cyc. How many people have heard of Cyc? Yeah?

So I think this guy believes that, if I can get enough people to type in things like, dogs chase cats and balloons go up or something-- I don't know, enough random factoids into this computer-- and he builds up an ontology of knowledge to some big collection of facts and a good search algorithm to back it up, that his computer will achieve intelligence. It'll be able to answer any query.

It'll be completely indistinguishable from a human. You could ask it any query. And as long as it's got the right chain of factoids stored away and a good search algorithm behind it, then this thing is intelligent.

And people have been typing things in for a really long time. You can go, and you can get the student version or whatever and ask it questions. And you can get a research version and ask it questions.

And people use it. And I mean, Google, I think, works better because it's accessing WordNet, which is basically an ontology of synonyms and things like this. These sound crazy, but they're the backbone of some of the things that you use every day.

OK. So for whatever reason, for these kind of reasons, the search algorithms, the motion planning algorithms that we're going to talk about, which are today used a lot in robotics, grew up under the computer science umbrella under artificial intelligence.

So the term motion planning typically implies that you've got some dynamical system you're shoving around, let's say, but it's clearly just an instance of this search problem. When people talk about motion planning in robotics, there are problems that people care about a lot.

A majority of the motion planning literature in robotics is concerned with kinematic motion planning, often with obstacles or something like this. So if I have a robotic arm with 10 degrees of freedom and I want to reach through some complicated geometry to turn a doorknob or to reach down and pick up some part from assembly, then that's a complicated geometric problem.

Typically, you assume that the systems are fully actuated. You don't worry about actually handling the trajectory once you get there. That's normally assumed away. And they assume that all trajectories are feasible. They're just trying to find some path through the obstacle field.

So one of the key problems that you can think about in that world is the piano movers problem. How many people know the piano movers problem? OK. How many people have ever moved a couch into a dorm room?

You all know the piano movers problem then, right? Basic task is you have some 3D shape. You have a bunch of obstacles. You have to find a position and orientation trajectory that will maneuver your big piano through a world of obstacles.

It's just like going up the stairs with your couch, you know, marking the walls and bending your couch as you go. And that's sort some of the driving problems in the motion planning space. So it's things like that, finding a collision-free path for a complicated geometry through an obstacle based environment, just trying to give you a little culture here.

And in that world, one of the most important concepts, I think, is the configuration space concept, which actually-- you know who came up with configuration spaces? He's sitting right upstairs. It's Tomás Lozano-Pérez is, I think, credited with the configuration space idea.

And the idea is that these problems can get a lot simpler. Let's say I have a square robot going through some complicated obstacle field. And I want to find a path from some start to the goal.

Well, instead of reasoning about the geometry of this relative to the geometry of this all the time, the configuration space tells you you should actually add the volume of your robot to your obstacles coming up with configuration space obstacles. If you actually add the volume of your robot to the obstacles-- I'm doing this very hand-waving here.

But then you can turn the problem of finding a big geometry through another geometric field to taking a point and driving it through the geometric field. And that's a critical, critical idea in the world of kinematic motion planning. We don't really care about kinematic motion planning in this class.

We've got enough troubles without obstacles. We just care about having an interesting dynamic system to move around. So the things we're talking about in this class fall under the heading kinodynamic motion planning.

And of course, the nonholonomic motion planning ideas are very related, too. We care about the dynamics. We care about things that are forced to follow some trajectories, like nonholonomic systems are, too.

So if you're out there seeing papers and talks and everything by motion planning, I just want you to sort of see that they're certainly related to the optimal control things we've talked about. But sometimes they let go of optimality. They're often about kinematics, but there's a good subset of them which is thinking about exactly the problems we care about with this kinodynamic motion planning.

Excellent. OK. So now, let's do some motion planning. Culture is there.

When we did dynamic programming, I already sort of forced upon you the idea that we could potentially, at great cost, discretize our system and state and actions potentially. So let's do the same thing when we start off with motion planning and start with discrete planning algorithms.

Let's say we've got our phase plot of our simple pendulum. For kicks right now, let's just start off with the trivial discretization, let's say. We're going to bin up our space like this, call each one of those a state and try to start talking about a graph-based representation of the dynamics of the system that tries to find a path, let's say, from our initial state to some goal state.

This isn't the only way to do discretization, but it's a good way to start thinking about it. So that turns the problem into some graphical model, where the control actions determine our state transitions. I like to actually call them s and a in the discrete world.

So people know a lot about doing search on a graph. I don't want to bore you with it, but I want you to at least know the algorithms that are out there and be able to call upon them if they become useful. So let's see we've butchered up our system like this, and we want to find a path from some starting discrete state to some goal state with a discrete set of actions. How can we do that?

AUDIENCE: [INAUDIBLE] search algorithm, like A star.

RUSS TEDRAKE: A star is a perfect example. Yeah. Dynamic programming is on the table, too. Dynamic programming, that's actually the way I-- the reason I drew that picture before is because we talked about that as a way to think about dynamic programming.

In dynamic programming is incredibly efficient if what you care about is knowing the optimal feedback policy, how to get to the goal from every possible initial state. I mean, the problem with that is, if you have to look at every single initial state, then it's not going to scale the high dimensions because there's going to be a lot of states.

If you have a slightly higher dimensional system and you want to find a start to the goal, but just a single path-- you don't care about every possible state-- then you can do not better than dynamic programming. But you can be more selective with your dynamic programming algorithm by doing these Dijkstra type and A star type algorithms. So let me just sort of do that quickly, so you know that that's there.

So dynamic programming, again, it's very efficient. It goes to a global optima. Typically, DP is used to describe the version where you're trying to solve for every state simultaneously.

You can do a bit better, you can be more selective, if you just care about from a start to a goal, like 0 to xg. It's actually a little bit surprising to say that I can actually find my path from a start to a goal and know that I'm at a global optima without ever expanding every single node.

And then sometimes we don't even care about optimality at all. Like I said, some of these algorithms are just trying to find any old path to the goal, and then you can certainly imagine finding your path to the goal without expanding every possible node.

They all have a very simple recipe. Basically, you keep a prioritized Q. How many people know what a prioritized Q is? I should say it's often called Q, but it means queue. It's another sleep deprivation thing there.

So a prioritized queue is a data structure that we use in computer science that I'll describe quickly here. So we basically have a list of nodes that we want to consider expanding next stored in some list, but they're stored in that list in an ordered fashion, ordered by how likely it is that we want to check that node next.

Depending on the ways that you add nodes into the prioritized queue, you can do any number of the standard algorithms, the breadth first search, the depth first search. Dijkstra's, let me make sure I spell it right. D-J-I, right? What is it?

AUDIENCE: D-I-J.

RUSS TEDRAKE: D-I-J? OK, good. Thanks. And you could do A star. There's more. There's iterative deepening, things like this.

They all go basically like this. I have a queue of things I'm about to explore. I start off with the starting state. I put that in my queue.

Next step of the algorithm-- take something out of the queue. Consider its children. And then take the first one out of my priority queue, repeat.

If I consider the children in a first in, first out kind of way, if I add nodes and just say, as soon as I pick up a node, I'm going to go ahead and stick those nodes into my queue in a sort of a first in, first out kind of way, then what I'm going to do is I'm going to proceed to look for the goal by going here, then here, then here, then here, then here, right down the tree. That's a breadth first search.

If I do a prioritized queue where, when I add the children, I just do a last in, first out, then I'll go here. Then I'll go here. Then I'll go here, and I'll go as deep down the tree as I could possibly do before I come back and go down to the other nodes.

If I keep along, let's call it, a cost to come to differentiate it from the cost to go, if I keep a record of how much cost I incurred for my cost function getting to that node and I always select to expand the node that would be the cheapest, then that's called Dijkstra's algorithm. It's exactly equivalent to dynamic programming, but it's in the forward sense.

And then A star is an even nicer way to do that, which combines a cost to come plus some heuristic.

My goal is to make sure you guys know these are out there. There's lots of good easy places to read about these algorithms, especially Steve LaValle's book. The only surprising thing really about these algorithms is, first of all, that you can often very efficiently find your way to the goal without expanding as many nodes as you might think.

A star, in particular, if you can find a good heuristic-- a heuristic, in this case, is an estimate of the cost to go, a guess at your value function. If you can guess your value function in a way where you always underestimate it-- again, I'm not going to bore you with all the details here-- then actually you can sometimes expand very few nodes and find your way right to the goal. They're very efficient algorithms.

People out here use A star all the time. For the Grand Challenge, I think half the teams had A star planners running on their vehicle. For LittleDog, we're picking footholds to get from the start to the goal. We have an A star planner in there, discrete time planners.

These things are real. They're good. They're fast. If you can find a good heuristic, they can be very, very efficient.

If you care about a continuous time, a continuous state, continuous action plan that gets me from the goal, then they're not as satisfying because they relied on this discretization. So we know, when we're discretizing this thing, we're going to have some problems. I'm going to just assume that there's a control action that gets me sort of squarely in the center of this next discretized cell and then go like this.

If I were to execute out my actions from my discrete planning algorithm on the continuous time system, I'm going to quickly deviate from my plan. But these things are an excellent way to start to seed something like a direct co-location method. Find yourself a feasible optimal plan. And then stabilize it, let's say, with the LTV feedback.

Good. So that world of planning algorithms is out there. But about, I don't know, 10 years ago now, something big happened in the robotic motion planning world. People started using sample-based motion planning algorithms. And there was sort of a revolution in people using motion planning in robotics.

It's also sometimes called randomized motion planning. And these are we are going to dig a little deeper in because we use them all the time. I think they're an important class of algorithms.

So two of the most notable sample-based motion planning ideas are the rapidly exploring randomized trees and the probabilistic roadmap.

Another poll, how many people know about RRTs? Awesome. How many people know about PRMs? OK. They're pretty related to each other.

Let's talk about rapidly exploring randomized trees first. So both of these are an attempt to get away from these very coarse discretizations like this and sort of embrace the fact that we're working in a continuous state space. Instead of discretizing it at some known sites, we're going to discretize sort of at randomly chosen samples.

And as we add more and more samples, we're going to worry less and less about the discretization. And eventually, we're going to have something nice to say about the continuous space. So the rapidly exploring randomized trees, the RRTs, are a very simple algorithm to explain.

Let's think about it in terms of moving a 2D piano through a 2D world in configuration space. So we've got a point moving through a world. It should really be 3D, actually, if there's an orientation to the piano. But let's just look at the 2D problem here.

So I've got a point, an official state, an initial goal in a 2D world. Let's call it xy. And I've got some obstacles.

I'd like to find a path from the start to the goal without explicitly discretizing the space. The RRTs do it like this. Pick a point from random in the space. Try to connect your current tree, which in the initial state is just this x0, to that new point.

Let's make it exactly clear that I can't do that. My attempt to directly connect this is going to fail because it goes through an obstacle. So I can't add that node.

Sample again randomly. At some point, I'll get a node, like, here. I'm going to make a branch of my tree that connects those two points.

Pick a new random point. Let's say you get something here. Now, you have to choose between these two random points, decide which is the closest. The closest isn't always an easy thing to decide if you have complicated dynamics.

In this problem, it's pretty easy. We could just use something like a Euclidean distance. And I'll connect it up like this.

I get another random point. I connect it up like this. I get some random points that go whatever way.

If they end up here, I just throw them out. If they end up something that I can't connect, I'll just throw it out. But at some point, I'll start filling the space with these trees.

Lo and behold, if I do it enough, it doesn't actually necessarily take that long until I get up here and get close in the vicinity of that goal, very simple random algorithm. Do you understand how it's sort of avoiding the curse of discretization? It's just picking points at random from this continuous space.

So there's no constraints, no explicit constraints that it lies in the center of some bucket or something like that. Nodes could be arbitrarily close to each other. I can get arbitrarily fine sort of discretization.

What's really cool, what makes these things tick, is that the idea is so simple. The code is so easy to write. And they really, really work fast. Let me just show you that basic example.

This time, I'm going to do it even without any obstacles, but just to show you the basic growth of these kind of planners. Awesome. OK. So my initial condition is just that blue point.

And I'm going to start growing a tree in every random direction. There's not even a goal in this problem. This is just to show you the basic growth of the tree.

Every point I grow at random, every time I pick a point from a uniform distribution covering the space, I'm going to find the closest point in the algorithm, a closest point on the tree. And I'm going to try to grow my tree towards that node. Yeah. Sort of understand what's happening there? It's pretty faint, I think, but-- all right. So watch what happens if I just let that run.

OK. Trivial algorithm, it has what a lot of people like to call a fractal expansion of the coverage area. Of course, if there's obstacles, it'll take a little longer to get around there, but the algorithm probabilistically covers the state space as I add more and more points.

Some people call them dense trees because they do this. If you noticed something about them, they have sort of surprisingly nice qualities. It filled the space pretty uniformly pretty quickly.

It didn't just go off in one corner and start adding nodes on one side or something like that. It actually has a property that people call the Voronoi bias.

The Voronoi bias implies that the tree has a bias for always growing into the biggest open regions of the graph. And that just comes from sampling uniformly. If I have some tree that has a big, wide open area for whatever region, then with very high probability I'm going to pick a point that's inside the biggest region.

And the biggest regions in the space are the ones that have the highest probability of being chosen. So probabilistically, this thing fills the biggest open spaces in the search tree just by the virtue of sampling uniformly.

So you get these very, very dense, fast trees. And you can imagine, if there's obstacles or something like that inside there, it's going to work its way just around the obstacles. Yeah.

AUDIENCE: Does it slow down as you get more and more nodes in there?

RUSS TEDRAKE: Good. So what is the computation that goes into it? What's the expensive part of the computation would you guess?

AUDIENCE: Finding the nearest neighbor.

RUSS TEDRAKE: Nearest neighbor-- good. So, yeah, it slows down with the nearest neighbor, but not as much as you expect. Actually, there's two expensive parts. Only the nearest neighbor one shows up here.

The other one, potentially, is the collision checker. And in some complicated systems, checking if you're inside a region, especially if you're a big robotic arm or something like that, can be actually the dominant, expensive thing.

So these things, in MATLAB, everything's vectorized. It actually doesn't grow that badly with the number of nodes, but certainly the nearest neighbor calculation gets more and more expensive. By default, it's just checking the sample point at every possible node.

AUDIENCE: You could use a k-d tree or something.

RUSS TEDRAKE: Yeah, good. Yeah. So there are different sort of structures that you can use to make that look-up more efficient. So RRTs have kind of come in and done things to robotics that people hadn't done before. So this is a two-dimensional example.

People would have said before, I can do sort of motion planning effectively and maybe 5 or 10 dimensions, a little bit more than I could do with just DP, but not a lot more. Because I'm still doing these sort of random factorizations. The guys that came out and started doing these RRTs-- LaValle being one of them, Kuffner being another one-- started showing examples of robots with 32 degrees of freedom doing pretty complicated plans. See if I have that Kuffner video there.

It's an animated GIF. This is H7 at the time. This was the University of Tokyo humanoid entry before everybody started giving out ASIMOs. A little before Honda started giving out ASIMOs this is.

And it's doing things like taking that 32 degree of freedom robot, finding a plan to have it bend down and pick up a flashlight under the table. And that sort of shocked people. Now, people have gone off and used RRTs to do things like protein folding. It's was a pretty hot topic in the motion planning community and these other very high complexity geometric planning problems that people just haven't done before.

But even this guy, even though it's a robot, his feet are flat on the ground, yeah? And they're still just assuming that you can search in position space and then find some controller that'll stabilize it. So they're not doing the underactuated planning example.

All right, so I wouldn't be talking about it if I didn't think it was relevant for underactuated planning. So let's show you the basic story here. What happens if we do it on a simple pendulum?

There we go. So you see, I'm plotting only every 30 nodes or something like that. I pick a point at random. I start growing the tree.

The way I grew the tree, now, is I don't go all the way towards the goal because I can't necessarily go all the way towards the goal. I just take a step in the direction of the goal. How do I take a step in the direction of the goal?

I just try five random actions, let's say, and pick the one that got closest to the goal. So if you look closely, every time it expands a node, you'll see a bunch of different arrows coming out. And it picked the one that got closest to that random sample point.

And it's still just using the Euclidean distance to try to find the closest neighbor. This one I think I wasn't careful about doing the terminal condition. But I want you to see what the most important thing about this is, what do you see in that plot besides a mess as it gets adding more and more?

What I see is I see the fundamental structure of the pendulum. I see these spiral trajectories. It's amazing to me that trivial randomized algorithms like this can probe the basic dynamics of my underactuated system.

All right, so if you see the algorithm is, now, very simple. So I pick a point at random. That's actually a good example I just happened upon.

Pick a point at random. The red is my random sample plane. The blue is the closest plane in the tree in the Euclidean sense.

It happens to be a horrible choice for the closest point on the tree in the dynamic sense because I know, in state space, my dynamics can only go that way. So really, I'd like to have picked a point back there that was farther and Euclidean distance, but closer sort of in dynamic distance. And I'll actually talk more next lecture about how you could do that.

But from that note, I try 10 different actions, see where it would have taken me, and pick the one that ended up closest in Euclidean distance, which happens to have been the one that happened horizontally. That's enough to let this thing just sort of spread itself out into the world like a disease or something, just covering the space and probing the dynamics of the system. And you wait until you get close to the goal.

RRTs are very, very powerful way, now, to start trying to find reasonable plans on a robot. That's the vanilla RRT algorithm. There's a lot of ways you can quickly improve them. And actually, any of the graph-based search algorithms do this.

So if I wanted to do some simple things to try to speed up that algorithm, what might you suggest? There's a couple of reasons why it felt inefficient. One of them is because it's not doing a very nice job of chasing these sort of intermediate goals, like that example showed. The other thing is it seems like it's not really making any effort to go towards the goal.

So there's a handful of heuristics that people use to make these things tick. One of them is a goal bias.

I mean, there's lots of ways to implement it. The standard way to implement it is with, let's say, probability of 0.5 or something like that, 0.05, choose my random sample to be the goal. Otherwise, my random sample is just the uniform distribution.

That would certainly encourage the system to find its way to the goal a lot more efficiently. What's another possible way to speed this up?

AUDIENCE: Bidirectional.

RUSS TEDRAKE: Bidirectional, yeah. In fact, all of the search techniques, the breadth versus depth versus A star on discrete graph searches, a lot of times one of the standard things people do is they'll do a backward search. While you're growing out from the start towards the goal, you might as well start from the goal and go backwards.

The problem with this is it feels like you're trying to find a needle in a haystack. If you can grow a tree in both directions, then at least you got a lot of needles to look for. And you just wait till the trees come fairly close.

And even backward search sometimes by itself can be faster. It depends on the system.

AUDIENCE: Is that because, if your goal is an unstable point [INAUDIBLE], it's more likely to find it? Is that intuitively what happens?

RUSS TEDRAKE: Why sometimes a backward search would work better?

AUDIENCE: In cases like underactuated.

RUSS TEDRAKE: It's a really good question.

AUDIENCE: Because you might have trouble going to some type of fixed point, but that for sure already starts you out there.

RUSS TEDRAKE: I won't put my weight behind it, but that sounds like a pretty good explanation. In practice, we always do bidirectional trees.

AUDIENCE: Even when you want to do both, you never just go backwards.

RUSS TEDRAKE: Yeah.

AUDIENCE: OK.

RUSS TEDRAKE: But certainly, in the kinematic case, you could definitely imagine cases where the backwards could be faster. Let's say it's hiding in some little island, and you just want to get out. And that's much easier than finding your way into the island, let's say.

So certainly, we've noticed sensitivities to growing RRTs around very unstable fixed points. And it might be exactly what you said. It might be. Yeah.

AUDIENCE: So is the Euclidean norm the best measure for this?

RUSS TEDRAKE: Good. The Euclidean norm is a horrible metric for this. What do you think would be a better metric?

AUDIENCE: Energy or times [INAUDIBLE].

RUSS TEDRAKE: Awesome. So energy's a good candidate. In fact, I think, depending on what we put on the problem set, we might ask you to try an energy metric to do it. So almost everybody says this would work better with a better distance metric. And almost nobody can tell you a better distance metric for these kind of systems.

So we actually have a couple of ideas that we've been working on in lab. Elena has been working on some LQR-based distance metrics, for instance. So what did you just say? You said time to arrival. How would you compute a time to arrival?

AUDIENCE: You have to do a moment in time.

RUSS TEDRAKE: Right.

AUDIENCE: Muscle control.

RUSS TEDRAKE: Exactly.

AUDIENCE: [INAUDIBLE]

RUSS TEDRAKE: So you just guessed the one that's one of my favorites right now. So one idea here, not fully proven yet, but it's still in the oven here is that, every time, I pick a random sample point I'll linearize the system around that point. Say that's a cartoon of a linearization.

Compute a LQR stabilizer around this. And then use the value function for a minimum time problem. So compute a min time LQR actually around this point. And then use the value function as the distance metric.

So I like that idea a lot. There's a couple subtleties that go into it. So when you pick a random point in state space, if it's got a velocity or-- most points you pick randomly in states space are not stabilizable.

So you can't do an infinite horizon minimum time LQR. You have to do a finite horizon and then potentially search over horizon times, but you can do that, actually, pretty efficiently. The other issue about it is that it's actually not a proper distance metric. Why is it not a proper distance metric? Anybody?

AUDIENCE: It doesn't [INAUDIBLE].

RUSS TEDRAKE: So I think--

[INTERPOSING VOICES]

It probably doesn't follow the triangle inequality. It doesn't even have symmetry. If I were to linearize around this point and compute its distance, linearize around this point, there just absolutely no guarantees that it would be the same.

But I don't care about that actually. I just have to put it in the paper. As long as it works, right? It's also more expensive a little bit, but it can make a dramatic improvement in the way it spreads.

There's another idea. I was actually planning to talk about some of the more subtle ideas next time. But since we're at it, the other way to do it is to change the sampling distribution.

So Alec Shkolnik in our group has got a very clever way to grow these trees, where it uses a sampling metric that doesn't try to go where it can't go. Let me see if I can say that more carefully. I told DARPA about it three days ago. So I just want to show you the picture.

OK. So this is Alec's algorithm for being more clever at expanding. He changes the bias to the point where this is the tree we just saw it, basically, right? And his new tree looks like that. So it looks pretty compelling.

How did he do it? He does this thing he calls reachability guided RRTs. Every time he expands a node, he also expands what would happen if I had applied maximum torque and minimum torque.

So he expands a couple nodes that represent a cone of feasibility. The algorithm turns out to be trivial. So now, I take my Euclidean distance metric or whatever it is, pick my random sample point.

If the closest point is one of these boundary feasible things, that means there's room to grow the tree. If the closest point, however, is one of the nodes in my actual graph, that means I'm not feasible. I said that quickly. But long story short, there's a trivial check here, which effectively changes your sampling distribution just by saying sample a point. If the closest point on there is not one of my boundary regions, but one of my normal tree elements, then I know I can't grow there.

And if I get that, I just throw that point away, pick another point. What that does is it effectively changes the sampling distribution to be the green region. It just throws out all these unnecessary points that you can't get towards and changes your sampling distribution, so it's only trying to grow in places that it can actually go, to where the trees got a chance of growing towards.

So I think the world is still sort of small enough that people are still finding pretty clever little tricks to the RRTs that make them grow a lot better in constrained systems like this. So that happens to be, I think, a very good one for our kind of problems. But it also points out what RRTs are bad at.

So what kind of problems would you expect an RRT to work on? And what would you not expect an RRT to work on? The classic case of something you shouldn't expect an RRT to do very well on is something that looks like this, let's say.

x start is over here. x goal is over here. And my obstacles look like this. In general, I think it's fair to say RRTs don't like tunnels and tubes.

What's going to happen if I run my RRT on this sort of a system? So you know, I got my tree here. Pick a point here, fine. Pick a point here, I'll quickly cover this area.

Pick a point here, I throw it away. Pick a point here, I throw it away. Pick a point right here, can't get to it. It doesn't do me any good.

You know, after a very long time, maybe I luck out and I pick a point here, right? But then I have to wait, and I pick point here. I can't get to it. You know, I'm just completely hopeless.

If it's in two-dimensional space, then you can imagine sort of getting lucky enough or designing a sampling distribution that does this. But let's say it's in some 14-dimensional space that you have no idea where the tunnels, the tubes, are. There are some problems where RRTs just choke.

AUDIENCE: So in general, when there's basic solutions, feasible solutions, it's small? It's that fair to say? Because this is a very small set of [INAUDIBLE] space.

RUSS TEDRAKE: It's a little hard to say that, right? Because, I mean, lots of trajectories work, right? This one works this. This works, you know? But I think, when there's narrow passages, I think that's a more clear way to say it when it chokes.

AUDIENCE: And so when you say chokes, it means it's, as times goes to infinity, it'll finally find it.

RUSS TEDRAKE: Yes.

AUDIENCE: It is [INAUDIBLE].

RUSS TEDRAKE: Yes.

AUDIENCE: OK.

RUSS TEDRAKE: Good. So these algorithms, mostly what you look for in a planning algorithm is completeness. You want to know that, as you expand all the nodes, eventually if there's a path to the goal, you'll find the path to the goal. And even if there's not a path, it'll tell you there's no path, right?

RRTs have probabilistic completeness guarantees. They say, if you expand enough nodes, then with the probability one, if there is a path, you'll find it. Probabilistic completeness doesn't do as much for disproving the existence of a path. It would take a very long time to disprove it.

But probabilistically, these things will find a path to the goal if it exists. That's good. Maybe that'll make you sleep better at night. But if you have to wait for something to be probabilistically complete, it's not going to be useful.

The reason these things are used like crazy is that, in practice, they're very fast. They can be very, very fast for finding good solutions. So really, I mean, the Grand Challenge vehicle is using RRTs all the time to consider possible future paths along the street, trying to avoid obstacles.

It's doing it fast enough sort of in a receding horizon way. It's always planning a minute ahead. I'm not sure exactly what the window is in driving time. And it's doing it every DT roughly, re-computing its plan. These things can be really, really fast in practice.

To some extent, the dynamic constraints which we're talking about on the pendulum are exactly this problem. Because the pendulum, like we said, it's coming here. We know what the phase portrait of the pendulum look like. And we know that, if we're torque limited, there's only a handful of places we can go.

To some extent, kinodynamic constraints, like this, dynamics can look exactly like this. So if you take your vanilla RRT, you saw how effectively it grew on the pendulum. It took a lot a lot of nodes to find its way out. The hope is that we know enough about this sort of dynamical system from some of our other tools, like LQR, to do distance metric from tricks like this that we're effectively guiding this thing down the differential constraint tubes.

AUDIENCE: Have you had any [INAUDIBLE] with relaxing the number? Like, for example, if you have like a 15-dimensional robot, you're going to get to [INAUDIBLE] position. Can you relax enough those dimensions and then use the rest [INAUDIBLE] under control in a very low dimensional space and then use that as a [INAUDIBLE]?

RUSS TEDRAKE: Awesome. Yes. So Alec also had-- I'll tell you what, my next slide, this is Alec's other thing about planning in a task space, which is exactly what you said I think.

There's lots of ways to do it. And I think Alec found a really good way to do it. This is just an example of a five-link robotic arm just in configuration space.

They had to find its way from that endpoint to the goal endpoint. So you see at the top, that little green arm is the resulting configuration of the robot. The standard RRT looked all over the place to try to find that solution.

It's a little subtle how he had to do it. You had to basically change the Voronoi bias to live in the task space, but still sample uniformly from the other space. So that's not enough to tell you how to do it, but that's enough to tell you it's not completely trivial.

Then he found these really, really elegant solutions to plan in task space. And the result, his sort of killer plot, is that this is the number of links in the robotic arm versus the number of nodes it took to find the goal. And the standard RRT went like that.

And after 10 links, it just was hopeless. With n going to over 1,000 dimensions, he was just planning in constant time basically. So, yeah, there's the structure in the problems.

We also, using the partial feedback linearization task space, have tried doing it on underactuated systems. And that sort of works, too. Yeah. But it's not the norm, actually. It's not the accepted version of the algorithm yet.

RRTs are beautiful things, right? They just grow out, find their way to the goal. Potentially, a complaint is, when you're done with it, you've got the sort of choppy discretized path that gets you to the goal.

But then just hand it to your trajectory optimizer as an initial condition and let it smooth it out, still satisfying obstacle constraints, torque constraints, We know how to do that and smooth it out. And you've got yourself a good trajectory that gets from the start to the goal.

It's a beautiful thing, I mean, I think to the point where Tomás Lozano-Pérez invented configuration space, did motion planning for however long. He actually stopped robotics, went off and did computational biology for a while. And he says he came back to robotics just a few years ago because these kind of algorithms made a difference to the point where it was worth doing motion planning robotics again. I don't a more dramatic thing to say than that.

OK. So next time, I'm going to go a little bit more into using these planning algorithms and also feedback motion planning applied to our model systems, but I don't want to completely segue. That's a big idea. These RRTs-- simple algorithm, big idea.

I mean, the idea that a randomized algorithm can do better than our explicit algorithms, it's all these funny things. There's classes on randomized algorithms here. Every once in a while, these things just really work nicely. Randomization can help.

And the reason, roughly, may be is that it's actually pretty cheap to pick a sample point. It's pretty cheap to evaluate if it's in a collision. And so why not just pick a lot?

Instead of trying to reason about the geometry of your obstacles, building configuration spaces, that's hard compared to just saying, OK, what if my robot was here? Was that a collision? Yeah, that's in collision. OK. Throw it out. And just sometimes randomized algorithms are better. OK, see you next week.