The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or to view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

PROFESSOR WILLIAMS: OK, so today's lecture-- we're going to be talking about probabilistic planning later, and in these cases where you're planning a large state spaces is very difficult. You do the MVP planning. It could be stress that activity planning, or the likes. But you have to be able to figure out how to deal with these state spaces.

So Monte Carlo tree searches is one of the techniques that people can identify, over last five years, is having an amazing performance improvement over other kinds of sample-based approaches. So entity is very interesting from that standpoint. And then if we [? link it to ?] the last lecture, then the combination of something, we just learn about [INAUDIBLE] and combine it with search, is very powerful, in this case, through the state-of-the-art techniques for that, as much as tree search [INAUDIBLE] later [INAUDIBLE]

PROFESSOR 2: Good morning, everyone. As Professor Williams just said, we are going to be talking about Monte Carlo tree search today. My name is Eann and I'll be leading the introduction and motivation of this presentation. By the end of this presentation, you will know not only why we care about Monte Carlo tree searches. As Professor Williams said, there's so many algorithms out there. Why do we care about this specific one?

And second, we'll be going through the pros and cons of MCTS, as well as the algorithm itself. And then lastly, we will have a pretty cool demo on how it's applied to Super Mario Brothers and the latest Alpha Go AI that built the second best leading Go player in the world.

So the outline for today's presentation is, first, we're going to talk about pre-MCTS algorithms. There are other algorithms that currently exist out there, and just a few of them to lead into why we do care about MCTS and why these other algorithms fail. And second, we'll talk about Monte Carlo tree searches itself with Yo. And lastly, Nick will tell you more about the applications of Monte Carlo tree searches.

So the motivation of these kind of algorithms is we want to be able to play games and we want to be able to create programs to play these games, but we want to play them optimally. We

want to be able to win, but we also want to be able do this in a reasonable amount of time. So these three can train itself leads to different kinds of algorithms, and different algorithms with different complexities and time, or times to search. And so that's why today we're going to be talking about Monte Carlo tree searches. And you'll figure out in a few slides why we do care.

So these are the types of games we have. You have this chart where there's fully observable games, partially observable games, determinstic, and games of chance. And so today, the games that we care about are the games that are fully observable and deterministic. And these games are games like chess and checkers and Go. And we'll also be talking about another example with Tic-tac-toe.

So these pre-MCTS algorithms include deterministic, fully observable games, like we said earlier. And the idea of this, and the nice thing about these games, is that they have perfect information, and that you have all of the states that you need and there's no opportunity for chance. And so the idea is that we can construct a tree that contains all possible outcomes because everything is fully determined.

And so one of these algorithms, to address this, is the algorithm Minimax, which you might have heard before. And the idea of Minimax to minimize the maximum possible loss. That sounds a little weird in the beginning, but if you take a look at this tree, this red dot, for example, is the computer. And so in the computer's eyes, it wants to beat its opponent. And we're assuming the opponent wants to win also, so they're playing their best game as well.

And so the computer wants to maximize his or her points, but also knowing that the opponent, or the human, wants to maximize their own win as well. And so in the computer's eyes, it wants to minimize the maximum possible lost. Does that make sense to everyone? Yes? OK. And so in the example of Minimax, we're going to start with a connect, or a Tic-tac-toe board, where the computer is this board right here, and the blue Tic-tac-toe boards are the states that the computer finally chooses. It's anticipating the moves a human could play.

So if you take a look up here, here's the current state of the board. The current state of the board. And the possible options for the human are this guy, this guy. Nope. Possible options for the computer, we have three different options. And so you'll notice that this is clearly the obvious winner. But in the state of Minimax, it goes through the entire tree, which is different from depth-first search. It goes through the entire tree until it finds the winning move and the minimize of the maximum possible points it could win.

So is there a way we can make this better? Yes. I'm sure you've heard about pruning, where, in our human intuition, it makes sense. Well, why don't we just stop when we win, or when we know we're going to have a game that allows us to win? And so this idea is the idea of simple pruning. And so when we combine Minimax and simple pruning, we have-- anyone know?

AUDIENCE: Alpha, beta.

PROFESSOR 3: Yes. Our 6.034 head TA knows about this. We have alpha-beta pruning, where we prune away any branches that cannot influence the final decision. So in other words, you wouldn't keep exploring the tree if you already knew that a previous term would allow you to win. And so this idea in alpha-beta pruning, we have an alpha and a beta. And so the details aren't important for you to know right now, but the idea is that we stop whenever we know we don't need to go on any further.

So in the games that have Tic-tac-toe and Connect 4 and chess, we have relatively low branching factor. So in the case of Tic-tac-toe, we have 2 to the fourth branching factor. But what if we have really large branching factors, like Alpha Go? In Alpha Go, we have 2 to the 250. Do you see that Mini Max, or even alpha-beta pruning, would be an optimal algorithm for this? The answer is?

AUDIENCE: No.

PROFESSOR 3: No. And this leads us to out next section. Our goal is going to talk about how we can use the Monte Carlo tree search algorithm for games with really high branching factors, and using the random extension to allow us to see, ultimately, how Alpha Go, which is Google's AI, was able to beat the leading Go player in the world.

PROFESSOR 3: All right, guys. So this is the part where we re-explain the algorithm itself. And before we dive into this, I want to make something really clear, which is that because these are technical details and because we actually want you to understand them, and because I definitely didn't understand this the first three times I read the paper. I really want you to feel free to ask any questions on your mind, with the knowledge that, in my experience, it is very rare that someone asks a question in class that's [INAUDIBLE] OK, so really, whenever you have one.

OK. So why are we doing this? Well, the ideal goal behind MTCS is that we want to selectively build up different parts of the tree. So the depth-first search way, the exhaustive search, would have us exploring the entire koopa tree, and that our depth is limited by looking at all the

possible nodes of that level.

But what we want is we want-- because the amount of computation required for that explodes really quickly. With the number of moves that you're basically looking into the future, we wanted to be able to search selectively in certain parts of the tree. And so for example, if there are less promising parts over here, then we care less about looking into the future of those areas. But if we have a certain move-- in chess, for example, there's a certain move where in two moves, you're going to be able to take the opponent's queen. You're really want to search that region and figure out whether that's going to end up being a significantly positive group for me.

And so the whole goal of our algorithm is going to be growing this asymmetric tree. How does that sound? OK, great. So how do we actually do this? We're going to go over a high-level outline, but before we do that, let's talk about our tree, which you're going to get very familiar with. Can people see that this is red and this is blue?

So this is our game state when we start our game. We can be given a Tic-tac-toe board with a [INAUDIBLE] place, a game of chess with the lose configured a certain way. And so our player, which is the computer, has three separate moves that it can take. And so each of those moves are presented by a node. And each of those moves have response moves by the opponent.

So you can imagine that if one of these is a Tic-tac-toe board with just a circle, that one of these is with that circle and the next place right by it. And as you go down the this tree, you start understanding basically, it's the way that humans think about playing these games.

If I go here, then what if they go there, and then what if I go right here. You try to think through the set of future moves and try to evaluate whether your move will be good in the long term sense. They way that are going to expand our tree, as we said, to create an asymmetric tree is first of all, we're going to descend through the tree.

We're going to start at the top and we're basically, jump down some sequence of branches until we figure out where we're going to place our new node, which seems like a key operation here. To create an asymmetric tree it's all about how you [INAUDIBLE].

For example, in this case, we're going to pick this sequence of nodes. And once we get to the bottom and find every location, we're going to create a new node. It's not very hard.

Then we're going to simulate a game from this new node. And this is the key part of MCTS. Once you get to new a location, what you're going to be doing then, is you're going to be simulating a game from that new location.

We're going to talk about how you go about simulating a game from this more advanced game state that what we started out with. Does anyone have any questions right now? We will be going in depth into all of these steps, but just in a high level sense.

AUDIENCE: Just a quick question.

PROFESSOR 3: Yeah.

AUDIENCE: To create the new node, is it probabilistic, just creating a new node as the most probable [INAUDIBLE]

PROFESSOR 3: No, no. You're creating some new node. We'll talk about how we pick that new node, but we're just making a new node and we're not thinking anything about probability.

The next thing is that we're going to update the tree. So whatever the value of the simulation delta was-- delta, remember-- we're going to propagate that up and basically add that to all of the nodes that are in that parent of that node in the tree and update some information that goes in there and that they're storing.

This is going to be good because it's going to mean that-- it's a lot like in search algorithms where you have trees that then the entirety of the tree remains up to date with the information from every given simulation. And we're just going to repeat this over and over and over again. And slowly, our tree will grow out until whenever we feel like stopping.

This is actually one of the nice things about MCTS, is that whenever we decide that we're out of time, like for example, if you're in a competition playing a champion Go player, you can stop the simulation. And then all you have to do is pick between one of the best first moves that you're going to make. Because an the end of the day, after you're doing all the simulation, we're still right here. And we're still only picking between the movies that go immediately where we started. Yeah.

AUDIENCE: Could this [INAUDIBLE] good tree? And then on some initial region of interest, or is it arbitrary how you get to create it?

PROFESSOR 3: We'll go through how you pick where to descend right now. I guess, it's any possible move that starts at your starting game state. Does that make-- great.

Before we move on to the algorithm itself, let's talk about what we store in each one of these nodes. So now we've added these numbers. And these numbers represent is that nk, as in the value of the right, is the number of games that have been played that involve a certain node.

So for example, if I look this node, that means that four games have been played that involve this node. A game that has been played that involves the node just means that one of the states of the board at some point in the game was the state of the board that this represents. For example, if I have a game that was played here, if I know that I've played this once, then that guarantees to me that I played this game once because this is a precursor state to this one. Make sense? Yeah.

AUDIENCE: How can the two n's below that node not add up to a value of [INAUDIBLE]

PROFESSOR 3: That will come when we start expanding our game. But that's a great question. And intuitively speaking, it should.

AUDIENCE: You're saying you're storing data from past games about what we've--

PROFESSOR 3: Yes.

AUDIENCE: --done before.

AUDIENCE: If past game's outside of the script simulation?

PROFESSOR 3: No, no, no. Past game's in the script simulation. And then the other value is the number of wins associated with a certain node. And these are going to be wins for player one, which is red in this case. It would get confusing if we put both of them, but they're complementary.

So for example, three out of the four times that the red player visited this node, they won in that node. And these are the two numbers that we're going to store. And we're going to see why they're significant to store later.

So first, descending the key part of our algorithm that we're talking about. And when descending, there are these two counterbalanced desires that we have. The first of them is that we want to explore really deeply into our tree. We want to think about, OK, if they do this then I'll do this. And then, well, then I'll do that unless I want it to forth.

And we want to think through a long term strategy. But at the same time, we don't want to get caught in that. We want to make sure that we're not missing a really promising other movie that we weren't even considering because we were really going down this certain rabbit hole of the move that we had thought about before.

This is illustrated by the x case [INAUDIBLE] SMBC. The SMBC comic about academia and how someone tells you that a lot of really great work has been done in an area, that means nothing about how promising the future will be. It's all about expansion and exploration.

And the way that we're going to balance expansion and exploration in order to create our really nice asymmetric tree is the following formula. And it's fine if that looks really confusing and messy. But actually, it breaks down quite nicely into two parts.

This formula is known as the UCB. You don't need to know why it's the Upper Confidence Bound. Let's just talk about what's inside it.

So first of all, you have this term on the left. And this term on the left is the extension term. It's basically proportional to the likelihood that the expected number of times that you're going to win, given that you are in a certain node and that you were a certain player. It's basically the quality of your state in some abstract level. If we knew this perfectly, then we would be doing great because that's the thing we're looking for on some grand level, The expected likelihood of winning from a certain state.

On the other hand, you have this exploration term. And you may not be able to read the font there. But what this is basically saying is that it looks at the number of games that I have been played through, and it was the number of games that my parent has been played through. And it tries to preserve those numbers at a certain ratio, at a log ratio.

And what that effectively means, is that the number of times that I have been-- if I have been visited relatively few times, and the denominator is small. Whereas my parent has been visited many times, which means that my siblings have gotten much more attention, then the likelihood that I will be visited again actually increases. So this is biased on the one hand, towards nodes that are really promising, and on the other hand, towards nodes that haven't been explored yet, where there's a gold mine and all you need to do is dig a little bit, potentially.

We don't actually have an analytical expression for this. But we can approximate it because

you can think that the expected value from a certain node is, roughly speaking, approximately the ratio of wins at that node to the ratio of times that that node has been visit at all.

Let's talk about actually applying this statement. Because what the statement is going to give you, is it's going to give you some number for here and some number here, and some number for here, and so on. When we start descending through the tree, we're going to start at the top node. And then we're going to look at the three children of that node. And we're going to compute this UCB value for each of these children and pick whichever one is the highest.

So just as a thought for a moment, what if we ignore this one? And what if we're just computing the UCB of these two? Does anyone have any intuition on whether the UCB would be higher for this node or for this node?

AUDIENCE: The left node.

PROFESSOR 3: The left node? OK. So why is that?

AUDIENCE: It has a win [INAUDIBLE]

PROFESSOR 3: Yeah. It has a win.

AUDIENCE: And they both have a [INAUDIBLE].

PROFESSOR 3: Exactly. And so clearly, you think the exploration term is the same because you know it's not that one child has been loved less than the other, but the expansion term is going to be different. And so it's definitely going to pick this one. In this case, what we're going to say is actually that this is so much more promising than the others that it's actually going to pick this left node.

And so it's going to expand, and it's going to look down. And then when it looks down, it's going to compare between these two. And this time, remember, that this is a parent. A parent want to minimize the number of wins that we have. Which means that our opponent is going to want to pick the one that were less likely to win in and they're more likely to win in. This is the idea of mini-max, minimizing how well my enemy does in this game.

Although again, the expiration term might counterbalance it a little bit because, technically, this has been explored more. We're going to pick the one on the left again. And we're going to get to that location that we got to originally.

Now when we're comparing between these two, between a node that has been visited once and a node that has never been visited, can anyone guess which one of these it is going to pick? Yeah.

AUDIENCE: Never has been visited.

PROFESSOR 3: Yeah, exactly. Because this number is zero. And so if the parent has ever been visited but the node hasn't, this is going to be infinite and it's going to have to pick the node that it has never seen before. So that's how we descend through the tree.

Does anyone have any questions on that. Really, it's totally fine. We're going to be talking about this for a while. Yeah.

AUDIENCE: With the left node that has the four for n sub k, wouldn't that be three because there's two and one below?

PROFESSOR 3: No because of the way that we're going to be updating the tree. Next, we'll talk about some [INAUDIBLE].

AUDIENCE: I like the concept. But if it's a deterministic game, why couldn't it hold it's [INAUDIBLE] pretty strictly?

PROFESSOR 3: That's a great question. That's really up to computer memory limits. As I think that Leah mentioned, the number of stakes in the game of Go-- it's a 19 by 19 board, and you can play something at every state. It's only like 2 to the--

PROFESSOR 2: [INAUDIBLE]

PROFESSOR 3: What?

PROFESSOR 2: 250.

PROFESSOR 3: 250. You could never explore the entire search tree.

AUDIENCE: [INAUDIBLE] over the first few layers or are we going polite. We try to do this real time where you could have done something offline.

PROFESSOR 3: It's definitely true. If you know a state that you're going to arrive at ahead of time, then you can totally do that. But in a game that's large enough that to do that for all the possible states

would take that much more time and take that much more memory. It doesn't end up making that much sense.

Also, something to point out here, is that for most of the games that we're talking about, simulating a run through of the game is really fast. So if you think about it-- let's actually get to that in next piece. But the point is that building up this many levels of a tree for a computer takes probably on the order of less than millisecond. So doing this for a really, really huge tree, it's peanuts because their such simple operations. But it won't get expensive when we start building up the tree to serious depths.

AUDIENCE: But a game like Go, how many nodes would you have?

PROFESSOR 3: On each level, in the beginning, we have something on the order of 400 nodes. And we have a depth of about, I think most games have up to 250 steps, or something like that.

AUDIENCE: So just to build, if you go in there blank, without any nodes built, you have to in the computer, like you said, it hasn't visited a node, it has to go there before it descends further. Basically, like breadth first.

PROFESSOR 3: It's sort of like breadth first but not quite. There's an important distinction here, which is that it doesn't have to build up this or this node. It doesn't have to build up all of the nodes at a certain level. All it has to do is, if it branches down to a certain sub region, then can't descend in that sub region below one of its siblings without having at least looked once at all its siblings. After it looks once it can do whatever it wants.

And the point is, that it doesn't mean the tree has to be kept at an even level. All it means is that the tree, in order to descend on a specific part of the tree, it has to have at least visited direct neighbors once before. Any more questions on this before-- Yeah.

AUDIENCE: What's the advantage necessarily of having to visit every single?

PROFESSOR 3: The advantage of having to visit every single-- the way that I think of it, is that you don't want to be missing out on potentially being interested in some of the things and not others. It comes back to the exploration versus expectation distinction. We do want to descend into the region of the tree that is really valuable to us. But at least have explored a little bit, at least maintaining some baseline, which really isn't that costly compared to the size of the tree. 400 moves is not that bad compared with 400 and 250.

AUDIENCE: Are these simulations, they're just random simulations?

PROFESSOR 3: We're going to talk about that in a minute. Any more questions before I move onto that?

Next step is expanding. And this is very simple. You just create a node and you set the two initial values. And the initial values are the number of times it's been visited is zero, and then number of times that someone has won from there is zero.

AUDIENCE: [INAUDIBLE] So the easy part is solving it.

PROFESSOR 3: Now, simulating. Simulating is really hard. You can imagine that if you get to a single node and you've never seen that node before, and you don't know what to do from this node onward, that if we knew how the game was going to play out, that is exactly what were searching for, and we would be done. But we don't. And in fact, we have no idea how to go about simulating a realistic game, and a game that will tell us something meaningful about the quality of a certain state.

And so, as you correctly guessed, we're going to do it randomly. We're going to be at a certain state. And then from that state, we're just going to pick random nodes for each of the players until the game ends.

And if we, as player one, win then we're going to add one. Then we're going to say delta equals plus one. And if we don't win, or if we tie or lose, then we're going to call it a zero.

You can in this graph, we're descending randomly and not thinking about it. And it turns out that this is actually great because it's really, really computationally efficient. If you have a board, even if it has 400 open squares, populating it by a bunch of random moves doesn't take you very long, on the order of not that many machine can.

AUDIENCE: That's why does you don't score-- if you go down a tree randomly, you already have a simulation. So the node's going to get to someplace. But you don't store it because it would lose the randomness?

PROFESSOR 3: You're totally right, actually, in this case. I've thought through this, and I can't come up with a reason why you wouldn't store it, that's it's temporary values that you find all the way down the tree. But they don't in most of the literature [INAUDIBLE] But you're totally right about that.

Does everyone understand that distinction? The fact that we only hold onto the result here and

don't theoretically make nodes for every place down in the tree just because we could, just because we've seen them before. We don't, and it doesn't really matter in this case. But it's theoretically a slight speed up that you could do.

AUDIENCE: But you reduce that question to generalities?

PROFESSOR 3: Yeah, a little bit. So we can look at an example of simulating out a running game. We get some intuition for why a random game would be correlated with how good your board position is.

For example, here we have a Detecto game. Circle is going to move next. But as hopefully you can see, because you have played Detecto before, this is not a particularly promising board for x. Because no matter what circle does, if x is an intelligent player x can win right now. It has two different options for winning.

And so, if you simulated this forward randomly, what you'll get is that 2/3 of the time, x will in fact win, even if the players aren't really thinking of it ahead of time. Yeah.

AUDIENCE: Then why not do n simulations at a node instead of just a single simulation?

PROFESSOR 3: You totally can do that. That's in fact, something that make sense to do and that some people do. Although what you'll find somewhat soon, is that considering that we're going down the tree, and that sometimes soon we're going to explore all of its children, there's a good question of why you end simulations now when you could just descend through the tree n times and thereby do n simulations by going through the thing and also building out the children? This case is-- yeah.

AUDIENCE: This gives more importance to why you do randomness. Because if you're doing random simulations you would ignore the possibility of the best one. When you first ran a simulation here was that o wins. If I ignore this node--

PROFESSOR 3: Absolutely. Which is why it matters that we do this so many times that we drown out all the noise that is associated with playing a game out randomly. Let's talk about that. If there's a lot of distance between where we are right now and our end result--

For example, in this game, if I were to tell you how good is this board position, if you are one of those people who played out every game of Detecto, you'll know that this is great if you want it to be [INAUDIBLE]

Anyway, the point is, that is not easy to do if you are doing random simulations from where you start. The correlation between your friend's board state and the quality of that state actually drops precipitously. And this for me is one of the hardest parts to study about Monte Carlo Tree Search.

Although, as Nick will explain to you, it actually works quite well. And one of the reasons that it works quite well in practice for more complicated applications is they do away with the assumption of random simulation.

Because even the random simulations does allow you to explore all the states, if you have some idea of where a reasonable quality approach would be, then using that, as long as it's not that much more expensive computationally, can help you with your simulation. Right now we're still talking about total randomness. How are people doing with that idea?

Now we're going to update the tree with the results of our simulation. So given that we had some result lambda, we're going to try to get up the parents. And for each parent we're going to add that the game has been played there once, and that the result of that simulation gets added if it was a one.

So for example, if there was a win in this game, than this becomes one, one because now it's won once and it's been visited once. And these two get incremented by one, and these two get incremented by one. That in itself comprises a complete iteration, the complete single iteration of running Monte Carlo Tree Search, which means that now we can keep doing this over and over again, building up the tree and slowly making it deeper, and making it deeper in selective areas. And having these numbers increase and increase. And be more and more proportional to the actual expected value of the quality of the state, until-- does anyone have any questions about this idea?-- until we terminate. And we have to come up with a way to terminate it.

Now again, we said we're going to pick what the best child is going to be, what the best immediate move from the start state is going to be. That's the move that were actually going to play. And so, how do we determine what the best is?

Well, the trivial solution is just the highest expected win given k. What that, in our case, is going to be is the ratio of number of times that I've win from a given early state to the number of times that I visited. However, this doesn't actually work as well as we might hope.

Let's suppose the following scenario, which is that you have the Detecto game like this. And

you have been exploring the tree for a while. And you're really mostly looking at these two nodes. One of these nodes, if you think it through, this node is quite promising and you've been exploring it for a while.

There is a winning strategy from this node. It's that circle goes here, and then x goes here, and then circle loses because x has two options to win. However, if you explore this a bunch of times, and for some reason, due to the randomness, this is at 11 out of 20. Whereas this state, which is inherently inferior, is at three out of five because of a bunch of randomness and because it hasn't been explored as much.

And if we had looked at this one as exhaustively we had at this one, that you probably would actually say that this state is actually better. And so, you can create an alternative criteria, which is that it's the highest expected win value of one of the children. But also, that value has to be the node that has been most visited so that they aren't explored by different amounts.

What this sacrifice is however, is that this means that we can't terminate on demand. This is not always going to be true, and therefore, we're going to have to let the algorithm run until that's true for some start state, which means that maybe is not a criteria that we want to apply even though we know that it would be wise to do so. Are there any questions about how we pick the terminating guide?

That was the whole thing. And now we're going to do it lots and lots of times until you guys are sick of Monte Carlo Tree Search. So this our tree. It's more or less what we've had before.

The first thing we're going to do is we're going to look at the top. And then we're going to pick one of these children. Now let's say that we looked at this, and it turns out that the one on the left is really valuable. I think it's the one. Nope, yeah. Never mind. It's wrong.

The one on the left has been explored a whole bunch of times. Remember, this term starts becoming larger than the ones that haven't been visited as much. And so we're going to descend from this one. And now we're going to descend, and we have these two options.

Given what you know, would you expect that this is going to pick is going to be the one on the right or the one on the left?

AUDIENCE: [INAUDIBLE]

PROFESSOR 3: On the right because it's never been visited before. And so, this term is going to explode. And

so, we're going to build a node there.

And then we're going to simulate a game. And the result is a win, which is bad for this player. That means that he probably didn't want to make that move.

And so we're going to propagate that value up. And we're going to start the algorithm again. And it's going to compare between these three.

And now it's going to pick the one on the left. Now that it picked the one on the left, it going to compare between these two states. Which of the two is going to have a higher expansion factor?

AUDIENCE: The left.

AUDIENCE: Don't you invert it, though, because this is the opponent.

PROFESSOR 3: Exactly. Because two out of three is actually better. Because it's one out of three for the opponent that's currently making the move. So the one on the left is going to have a higher expansion factor, and the one on the right is going to have a higher exploration factor. Does that make sense for people? It's OK if it doesn't.

So we're actually going to pick the one on the right because the other one was is doing three and has lots of it's mother's love than that one's. Anyone else need a drink? We're going to expand that node. It doesn't matter. They are both equally likely to be expanded.

We're going to simulate forward, and it's going to be one. Which means that that was probably a wise countermove. Yeah.

AUDIENCE: So when it's the opponent's turn versus your turn, the exploration factor is the same but we complement the expansion factor, right?

PROFESSOR 3: Yes. So the key here being that this takes in both the state that you're talking about and the player that you're talking about.

AUDIENCE: But regardless of the player, the exploration factor will always be like this is.

PROFESSOR 3: Because it's only the number of visits it's. It has nothing to do with results of exploration.

AUDIENCE: If you win and you have the plus one, double plus one, and you've propagated out, but I'm wondering-- so if the opponent wins do you also propagate out the win increment itself? If the

opponent's winning, wouldn't you want to [INAUDIBLE] node here?

PROFESSOR 3: If the opponent wins then what you do is you propagate up a zero. Which means that wk is not incremented, but nk is. Have we seen a zero yet? There's one soon.

But the idea is that rather than subtract or anything, all you do is propagate up the result of the game, which in this case is zero. Which means that all of those states seems to become more valuable to the blue and less valuable to the red. Because these numbers are lower than the other ones were.

AUDIENCE: OK.

PROFESSOR 3: So we propagate this up and this becomes better. What we've done here is we've figured out a theoretical countermove to blue moving here. That's how you should think about this whole tree. It's really a lot like the way the humans think about these things.

If I do this, then what if they do this? Well, then I'll do this. And I see that I'm successful when I do that.

We're going to look again at the top. And we're going to pick the one on the left because it's really promising. Five out of six is a good number. And we're going to look at both sides.

And which one is blue going to pick now? Well, it's going to pick the one that it's going to be more successful in, which is two out of three. I realize that this is actually not the kind of thing where I could necessarily ask people because I'm the one who's decided which node to stop.

Then we go down here. And there's an equal likelihood of picking either of those nodes. And so we're going to pick one at random. So that's going to be the left one.

And we're going to create an empty node. Then we're going to play it out. And it was a success for blue, which is amazing because what this means now is that suddenly, in this tree of this really good move that red could make the blue wasn't find a response to, suddenly there's hope because we're going to propagate this back.

And that means that blue actually has a response move to that sequence of red's moves. And so it's going to propagate up. And this state's going to be more promising to blue and less promising of red. That region of the tree that we had dug into is a little less promising.

We're going to look back up. And this time, instead, we're going to evaluate the thing that is both promising from the expansion factor, and also promising because we haven't looked at it very much [INAUDIBLE] exploration factor. We're going to pick between these two. Which one is going to be picked here?

AUDIENCE: [INAUDIBLE]

PROFESSOR 3: Because the exploration factor is the same but the expansion factor is higher for the one on the left. And it's going to show us a node. And the result is going to be a win for a red, which means that red has found a good countermove to the thing that was previously promising for blue. And we propagate it back up.

And finally, we're going to pick the one furthest on the right. Because even though it's terrible for red, and even though it's never won when it's tried it, it has to obey his idea of the exploration mode to find out whether maybe there isn't something possible there. So it explores, and it goes down, and it has to pick the one on the right. And so it does.

And it plays this game out. And it's a loss, again. Which goes to show you, that blue has found yet another superior move to this really bad move of red, where probably this move of red, if this is a game of chess, is like putting my queen directly in front of the opponent's row of pawns, and I just leave it there. There's nothing good that's ever going to come of it but we have to explore it just to find out whether there isn't some magical way that I should protect.

And as you can see, we've built up this tree over and over and over again. And it's starting to look asymmetric. And we're starting to see that there's really this disparity between exploring the regions that are crossing this tree and exploring the regions that are not and that don't really matter to us very much. And that this is exactly what we wanted from Monte Carlo trees. That was why we started the whole endeavor in the first place.

The next thing I'm going to talk about is the pros and cons. But before I do that, does anyone have any more questions about the algorithm? Yeah.

AUDIENCE: It's still not clear how we're getting nodes with different denominators-- [INAUDIBLE]

PROFESSOR 3: The reason for that is because of the way that we're simulating through. We're actually not holding onto to the results of the simulation as we're going farther down the tree than the lowest node we expand.

For example, when you simulate from here, you're going to propagate that value here and here, and so on. But then when we expand below, even if in the course of this guy's simulation it happened to go through one of the states that we expanded below, it will not have incremented the values of that state because we weren't keeping track of it. Theoretically, if we were to keep track of all of the simulations that we have in fact run, the numbers beneath these things would be higher.

AUDIENCE: If you've already run a simulation from that-- if you've already run a simulation from that red node when you first built it, and then when you created those two ones, each of those have [INAUDIBLE]

PROFESSOR 3: OK. I see.

AUDIENCE: So would the denominator always be one more than the sum of the children?

PROFESSOR 3: Yeah, in [INAUDIBLE] Yeah.

AUDIENCE: I understand how you built that. Is there a rule of thumb, like it's time to choose a move? And it seems like you have very low numbers here to make a [INAUDIBLE]

Is there a rule of thumb on giving games like it's 2 to the 4 or 2 to the 350, whatever it is. What kind of numbers do you need for that first row before you [INAUDIBLE]?

PROFESSOR 3: What we'll get to soon is that isn't one. That's one of the problem with MCTS. But in terms of which of the moves you will choose, there are actually variants of MCTS that suggest that you more selectively age or insert new children based on something more than just the blind look right now. In terms of, if I'm here and it's creating my next children as the equivalent, then there are some intelligent guesses that you can make in terms of which one you should score first. Although it doesn't particularly matter.

AUDIENCE: I'm just saying computational time being what it is, you might say, OK, if this is the timeline of this game I can expect to do a million simulations, which will give me if there's 400 nodes, I'm going to have so much use. In other words, is that enough time to say that I can play through a game? I couldn't play through a game with 400 options if I've gotten five out of seven [INAUDIBLE] three out of four [INAUDIBLE]

PROFESSOR 3: Absolutely. And I would say that so far as I know, that's something that's basically very high experimentally. They don't have good balance on it. [INAUDIBLE]

So let's get on the first comment because that is a computer element. So why should you use this algorithm? Even though we've seen tremendous breakthroughs in this algorithm, and you're going to have to ignore everything that I tell you and remember that this does actually work quite well in certain scenarios. Should we use it or not?

The pros are that it actually does the thing that we want it to do. It grows the tree asymmetrically. It means that we do not have to explore. And it doesn't explode exponentially with the number of moves that we're looking into the future. And that it selectively grows the tree towards the areas that are most promising.

The other huge benefit, if you'll notice from what we've just talked through, is that it never relies on anything other than the strict rules of the game. What that means is that the only weight of the game that's factored in is that the game is what tells us what the next moves we can take from a given state are, and whether a given state is a victory or a defeat. And that's kind of amazing because we had no external heuristic information about this game.

Which means that if I took a completely new game that someone had just invented, and I plugged MCTS into it, MCTS would be a slightly or someone competitive player for this game, which is a powerful idea. It leads to our next two pros. The first of which is that it's very easy to adapt to new games that it hasn't seen before, or even that people haven't seen before. This is clearly valuable.

But the other nice thing about it is that even though heuristics are not required to make MCTS work [INAUDIBLE], it can work [INAUDIBLE]. There are a number of [? advanced ?] places in the algorithm that you can actually incorporate heuristics into. Nick is going to talk about how AlphaGo uses this very heavily. AlphaGo is not vanilla Go. It has a lot of external information that's built into the way that it works.

But MCTS is a framework-- you can imagine your heuristics you can apply in the simulation, there are heuristics you can apply in the UCB in the way that we choose the next node. There are places that it can fit in. And this services as a nice infrastructure to do so.

The other benefit is that it's an on demand algorithm, which is particularly valuable when you're under some sort of time pressure, when you're competing against someone that's a mathematician, or when something is about to explode and you have to make a decision on which reactor to shut down.

And lastly-- or not lastly, actually, it's complete, which is really nice because you know that if you run this game for long enough it's going to start looking at a lot like a BFS tree. No, it's actually going to start looking like an alpha-beta tree, if it is what it is converted to. It's a nice property to have. Although, this property does slightly get compromised if you remove the red in this idea, and if only simulate these [INAUDIBLE]. Yeah.

PROFESSOR: You made an interesting comment when you said, oh, it looks like -beta tree. So it looked like a mini-max tree. But have they also incorporated notions of pruning in the MCTS, which would make it look like an -beta tree?

PROFESSOR 3: Sorry, you're completely right. It does look like a mini-max tree. I think I've seen variants where they do pruning, but I haven't looked into it as much. But I would imagine that they would converge to whatever you know pruning a certain tree [INAUDIBLE].

AUDIENCE: But people have explored incorporating pruning into MCTS?

PROFESSOR 3: I think so. I can't say [INAUDIBLE] And then lastly, it's really parallelizable. You'll notice, none of the regions of this tree, other than the original choice, ever have to interact with each other.

So if you have 200 processors and you decide, OK, I'm going to break up this tree in the first 200 decisions and then have each one of those flesh out one of those decisions, that actually means that they can all combine information right at the end and make a decision [INAUDIBLE], which is a really nice, powerful principle as you [INAUDIBLE].

It does have its fair share of problems. The first problem being that it does breakdown under extreme tree depth. The main reason for this being that as you increase more moves between you and the end of the game, you're increasing the probability-- you are decreasing the correlation between your game state and whether a random playoff would suggest that you're in a good position or a bad position.

The same goes for branching factors. One of the things that people sometimes talk about it as if MCTS AI's cannot play first-person shooters because the distance between the number of things that you can do at every given moment, and what would be a successful approach in the long term after meeting many, many, many moves that each have many branching factors, is that never begins to explore the size of the search tree. For the most part, it's not really coming up with a long term policy. It's really thinking about what are the next sequence of moves that I should [INAUDIBLE].

Another problem is that it requires simulation to be very easy and very repeatable. So for example, if we wanted to tell our AI, how do I take over Ontario? There's not a particularly good way that you can simulate taking over Ontario? If you try it once, you're not going to have an opportunity to try it again, at least with the same set of configurations.

And actually, one of the things that we really took advantage of, if that random simulation happens really quickly, on the order of microseconds. On other hand, the bigger your computational resources that you have access to, the better the algorithm works. That means that I can't run it off my Mac particularly well. It would be like large games.

It relies on this tenuous assumption of random play be weakly correlated with the quality of our game state. And this is one of the first assumptions that is going to be thrown out the window for a lot of the more advanced MCTS approaches, which are going to have more intelligent play outs. But those are going to lose some of the generality that we had before.

Something that goes off of that is that MCTS is a framework. But in order to actually make it effective for a lot of games it does require a lot of tuning, in the sense that there are a whole bunch of variants. And that you need to be able to implement whatever flavor is best suited for you. Which means that it's not quite as nice and black boxy as we would want it to be as far as give it the rules and have it magically come up with a strategy [INAUDIBLE].

And then lastly, as you mentioned, there is not a great amount of literature right now about the properties of MCTS and its convergence, and what the actual proportion of time to quality of your solution is. This is true of all modern machine learning things, is that there is certainly a lot more work that could be done. But right now, that's a gap in terms of using this for a simulation that's supposed to be reliable. Anyone have any questions on the Pros and Cons?

Before we jump dive into applications, let's talk through a few examples of what games could be solved and could not be solved by MCTS. Do you guys think that checkers is a game that could be solved by MCTS?

AUDIENCE: Yes.

PROFESSOR 3: It's completely deterministic. It's two-player. It satisfies all of the criteria that we've laid out before. Checkers is definitely a game that can and has been solved by MCTS, although not solved to the extent that you can defeat the thing that actually has the solution [INAUDIBLE].

How about "Settlers of Catan?" This one's a little bit trickier. Do you guys think that MCTS is likely to be able to play "Settlers of Catan?" If not, let's throw out reason why or why not it would be [INAUDIBLE]. Yeah.

AUDIENCE: No because there's randomness.

PROFESSOR 3: So yes, that is absolutely the criticism. And that's why we can't apply it vanilla. I put this on here as a trick question, though, because it turns out that MCTS is robust to randomness. That you can actually play-- and I realize that's just me and we do.

[LAUGHTER]

You can actually play through games. If you think about the simulation, the simulation is actually applicable even if the game is not deterministic because it does give you a sense of the quality of your position. And the MCTS-based AI to play "Settlers" is, I think, at least 49% competitive with the best AI to play, at least in the autonomous non-scale space. So it does work.

Let's talk about the war operations plan response. Who here has seen the movie "War Games?" OK. Well, it should be more of you.

The idea of "War Games" is that one of the core characters in this world is this computer that has been put in charge of the national defense strategy with respect to Russia. And that it needs to think through the possible future scenarios and decide whether it's going to launch the nukes or not. Do you think that WOPR can be MCTS-based?

AUDIENCE: No.

PROFESSOR 3: No.

AUDIENCE: It could, it just wouldn't be very good.

PROFESSOR 3: Absolutely. Once you fire the nukes you're not going to get another chance. So you can't particularly simulate through what the possible scenarios are going to be like. Yeah.

AUDIENCE: So what if you had-- I agree you can't simulate it in the real world. But what if you had a really good model and you just simulated based on that model?

PROFESSOR 3: In that case, it probably depends on the quality of your model. If you have a good model for

how World War III is going to [INAUDIBLE].

[LAUGHTER]

AUDIENCE: It is the case that the military does have simulators and they do war games in simulation.

PROFESSOR 3: Yes, that's true. They could certainly try it and run MCTS if they wanted. And that's what happened in the movie.

[INTERPOSING VOICES]

AUDIENCE: And there you're putting your money in the simulation not in the--

AUDIENCE: It's like having an MCTS play *SOCOM* or something like that.

PROFESSOR 3: Yeah. It's definitely about putting money into the simulation and you get really good simulation. If you have a really good simulations then you [INAUDIBLE] to play WOPR. Yeah.

AUDIENCE: Back to "Settlers" for a second. I'm curious if there's a way for the whole player training resources thing, or would it have to be only purely like using the ports.

PROFESSOR 3: That's a good question. I haven't looked closely at whether they do that or not. If it's playing a two-player game, then I would imagine that they wouldn't because you don't really trade in to play a game. But if they weren't, I bet that you can incorporate it with WOPR.

AUDIENCE: Is it limited to two-player games?

PROFESSOR 3: No, not at all. In fact, there are lots of purchases that do only one-player games, where you think of what's the best movie that you can make.

AUDIENCE: I know. But I mean, couldn't MCTS handle three- or four-player games?

PROFESSOR 3: Yeah, it absolutely could. I'm not sure how they computed their head-to-head. That might be completely flat cursors. I'm not even sure how the settlers interact. Yeah.

AUDIENCE: A quick question. So at first you know if I reduce the chess board to only 4 by 4 or 5 by 5, and I run MCTS versus the traditional algorithm that AlphaGo offered as a tree. Do you think MCTS will prefer theory and perform this computational requirement.

PROFESSOR 3: The thing about the way that Deep Blue is, which is the AI that ended the Kasparov thing, a bunch of his chess grand master, is that it has a tremendous amount of heuristic information.

There's a lot of external stuff that's incorporated into the system that makes it able to explore the best paths.

What I would say is that knoledgesless MCTS based on randomness, would take a very long computational time to even become competitive with those kinds of algorithms, and probably feasibly never would. What if you incorporated heuristic information, I think that there's a bunch of hope in terms of getting MCTS to start performing better.

And you can look at what next I'm going to talk about, AlphaGo. It takes inspiration for how we go about incorporating these new circuits.

AUDIENCE: So only the circuit you [INAUDIBLE]

PROFESSOR 3: It definitely seems like if you have a really good heuristic model for what good states in the game are, that if it's a smaller search space, that some other models could perform better. Although, I'm probably going to eat my foot here because this is going to be on OCW some massive amount, massive chess playing algorithms. Eat my shoe not my foot.

[LAUGHTER]

One last game. Does anyone know what this game is?

AUDIENCE: "Total War?"

PROFESSOR 3: Yes. Nice. This is "Rome, Total War II." It's a simulator for this tremendous real time strategy game, where you play, I think, the Roman Empire. And you're controlling armies and huge infrastructure systems that move and conquer states and continents, and meet in the field, and manage resources, and do all of these incredible diplomacy feats.

And so do you think that this game can be solved by MCTS?

AUDIENCE: Yes.

AUDIENCE: Yes.

PROFESSOR 3: Lets say no. But I guess I put it on here. So that's good on you.

The way that the AI in "Rome, Total War II" is built is that it's built on an MCTS structure. And it in fact does do resource allocation and a lot of its political maneuvers based on Monte Carlo

Tree Search moves. There are a bunch of reasons that they explain in the game for why they do this, or in papers released about the game. But one of the nice ones is that it's random, which means that you're never going to play against the same kind of AI twice because every time the set of decisions that it's going to think about is completely different.

AUDIENCE: I have a quick question.

PROFESSOR 3: Yeah.

AUDIENCE: So if I want to model any game with MCTS, does it have to be that the actions in playing a game has to be able to discretize.

PROFESSOR 3: Yes. So far as I know, I haven't seen many continuous variants in MCTS. And so, I think that it is about choosing these reactions, which on it's most narrow level does actually bring it down to here. I think one of the reasons that this is nice is that there are so many different decisions that could be made that MCTS is really the only approach that could even begin to handle the massive branching factor that's associated with the game Rome, Total War. Yeah.

AUDIENCE: This is also the consequence of this year you get the play off when this game comes.

PROFESSOR 3: That's interesting. That's probably totally it. That's cool. That's everything about how the algorithm actually works.

I'm going to pass it off to Nick, and he's going to talk to us about some actual limitations for this game [INAUDIBLE].

PROFESSOR 3: So as you have said, I'm going to start diving into some applications here. And not only applications but also some modifications or augmentations of MCTS. It should hopefully clarify some of the side questions you all have been having on slight tweaks to MCTS.

Now let's get started. Wait for it. Now let's get started.

[LAUGHTER]

Part III, applications. First thing we're going to look at is an MCTS-based "Mario" controller. And "Mario" might seem like some weird thing to test AI on, but there actually is a "Super Mario Bros" AI benchmark, which it used to test a lot of AI on how well they could play this platform.

In case any of you don't know what "Super Mario Bros" is, this is a screenshot. Basically, you control this one character. It's a single-player game.

The ultimate goal is to reach this flag at the end. But along the way there's enemies, there's some bonus shrooms you can get. If you break open some boxes you might get coins, things like that.

But first, let's just highlight some of the modifications that need to be made, or some of the differences between vanilla MCTS and an MCTS that's going to be able to work for "Mario." First thing is that it's single-player. The second is, we use a slightly different simulation strategy than the initial just vanilla simulation.

And someone actually hinted at doing more than one simulation because you, you're watching us to n simulations, I think. We'll touch on that. Then this also introduces what I would consider to be domain knowledge.

Then finally, there's a 50 to 40 millisecond computation time. And that has to do with the frames per second of the game. So you would think that "Mario" is a continuous game, but if we discretize time into these chunks, then we can use MTTS.

Now let's just think about how we could possibly formulate this problem. Can anyone think of what each of these nodes would be if we're playing "Super Mario?"

AUDIENCE: Jump.

PROFESSOR 3: Sorry?

AUDIENCE: Jump. It would be like, first node you're going to jump.

PROFESSOR 3: That might be a way to formulate it. But I think that could get--

AUDIENCE: Oh, it's not your control at inputs [INAUDIBLE].

PROFESSOR 3: Right. So the node itself isn't going to be an action.

AUDIENCE: Equal frames.

PROFESSOR 3: Yeah, basically. So it's going to be the state of a game, what we'll call a state. So it's basically just a screen grab. And it take it, in this case, it's a 15 by 19 grid screen grab of the game.

And it will have information about-- it knows Mario's position, it knows the enemy's position, position of the blocks, et cetera. And then, as Yo was saying, in MCTS we have values associated with our nodes. And so it will also have a value. But we'll get into the value in the next slide because I can't really fit it all in here.

With that being said for our node, that being the state of the game, what makes sense for the edge? Does anyone know? How do we transition from one state to another state?

AUDIENCE: Jump.

PROFESSOR 3: Yeah, exactly. So this is where the jump and all the action have been played. So the actions that you take-- I didn't list all the actions. You can also have a jump left, jump right, all those things.

But basically, the actions are what takes you from state to state. So I just drew out what a node might look like if you used the jump action. You might have Mario go up in the sky. Are there questions?

AUDIENCE: Does it just run the rest of it? Because that little thing's moving as they move on?

PROFESSOR 3: Well, it's not moving in this moment in time. We're discretizing time right now.

AUDIENCE: But I'm saying, if your action is jump, just you would have 1,000 nodes because if you did plan out where that thing's moving, left or right, then it could be--

PROFESSOR 3: Yeah, right. So in each state we have the enemy position. And we know the speed and direction. And so we know when we go from this node to one time step later, we'll know where the enemy's moving. Any other questions?

Moving on. Sorry. Let me just preface this part real quick. So in our other simulations, at the end of the simulation we would get either a one or a zero, if we'd won tic-tac-toe or we lost tic-tac-toe.

But that won't really work too well here because there's a lot of other factors that go into play when you're playing "Mario." Also, if you're doing a simulation, more than likely, you're going to end up hitting an enemy and dying or falling into a gap and dying. So a lot of these simulations might all return zero. And that is, you can't really distinguish between them.

So this is why I say, this version of MCTS introduces what I would consider to be domain

knowledge. Basically, they're assigning scores to potential things that could happened along the way. And this is basically telling the AI that collecting a flower is a little bit better than collecting a mushroom. It's telling it that getting hurt is bad.

Right off the bat, all these things in the score are giving the AI some domain knowledge about "Super Mario Bros," that it's helping it calculate the simulation results. As it says here, it's just doing a multi-objective weighted sum of all these things. Throughout the simulation it's just adding up your score. And then that's the score that is going to be propagated. Are there questions about the score?

AUDIENCE: You said that it adds up all these guys and it propagates it over. Is it possible to just propagate the multi-part sum [INAUDIBLE] as opposed to propagating one value that you create? Are you essentially propagating all-- what's this?-- 15 values upwards at every node, or are you propagating one value--

PROFESSOR 3: Well, it's one value. It's the collective--

AUDIENCE: Then you make them add it together and you got each one of them a sub factor.

PROFESSOR 3: Then also, just one thing to note here, is distance, you get 0.1. And these are all parameters that have been tuned. In the initial version, distance was, I think, a reward of five, but probably realized that that made Mario skip past a lot of coins and things. And so he tweaked the score for that.

And also, time left is two. So there's some weight there. You want to get to the very end of the game.

AUDIENCE: If you're pushing up this score, it's no longer a win over losses. So it's not w over n. What is it affecting?

PROFESSOR 3: You can just use the score.

AUDIENCE: The score is the--

PROFESSOR 3: Yeah. In MCTS you have this idea of when you're propagating your q value, you could have that to be zero, one.

AUDIENCE: It's like the sum of all the scores and the nodes below over the number of games you win.

PROFESSOR 3: So basically, what you would be getting when you divide by the number of simulations is your average score at that node.

AUDIENCE: OK.

AUDIENCE: When you have killsByFire and [INAUDIBLE] like that, if you have a positive value, then isn't it good to be killed by fire, or something like that?

PROFESSOR 3: This is killing an enemy by fire. Like Mario could collect a certain flower or mushroom? I think flower, then you have a fire breath and you [INAUDIBLE].

AUDIENCE: So that's Mario's status if Mario never dies?

PROFESSOR 3: No. Mario's status is-- I believe, Mario's status is the fact that you could upgrade Mario by collecting [INAUDIBLE] mushroom from a fire Mario. So that gives you a lot of points. Because if you become fire Mario, then you're more likely to not die by running into enemies because you have fire-spewing--

AUDIENCE: You said they spent a lot of time tuning these parameters. Isn't it generally, though, just an optimization framework if that's some formula? So they tuned the parameters just to make behave the way that we think is nice. But if you change the values, they'll do the right thing for that equation.

PROFESSOR 3: Yeah.

AUDIENCE: OK.

PROFESSOR 3: Yes. But they were tuning this to make it play how they wanted.

AUDIENCE: [INAUDIBLE] can't just be a reflection of [INAUDIBLE]

PROFESSOR 3: That's a strategy. If you choose that, I don't see why not. That might affect certain things. Obviously, you can change these to whatever you want. It'll slightly tweak which simulations as to working better, in terms of changing which nodes you end up choosing [INAUDIBLE].

So we move on. So we know about scoring simulations. Now we're going to look at exactly the simulation type that's used to play this MCTS controller.

So the regular version that Yo talked about is just choosing a random node at each level in your simulation. But there are some other strategies. And someone brought one up.

The first is, look at best of n. So in this one, you choose three random nodes at each level, except that you stick with the best of those three. Choose three random nodes, stick with this one. Go to the next one.

You would choose n random three, take the best one, and then go to the next level. You are able to do that in this game because of the way the scoring works, you don't have to get to the end of the game for your score. You actually could collect a coin along the way.

If this is jump, and then it gets to be a coin versus moving left and right. That doesn't give you any points. Then this is the node that would give you the highest scores, so I would choose that one, et cetera.

And then the final one, which is the one that is actually used for this controller, is multi-simulation. This was brought up by him. I don't know your name. Sorry. But basically, you run multiple random simulations from your node. And then you propagate up whichever of those simulations give you the highest value.

And the reason to do multiple simulations is to attempt to increase the accuracy of your simulations. If you just do one simulation you might just get really lucky. But if you do three then you can take the highest value use that as your value. Since the whole point of this is to try make moves that get you the highest values, then that will make your random simulation value more accurate. Are there questions about multi-simulation?

AUDIENCE: So what do you think about the simulation [INAUDIBLE] how many [INAUDIBLE]

PROFESSOR 3: So there's a trade off here. The more simulations you do the more accurate-- the more representative your simulation will be at the end of the game. You could run two to the whatever simulations to try to get every single possible action and then take the max of that. And that would give you the maximum value. That would be ideal.

But obviously, that takes more time. So there's a trade off between computation time and the number of simulations you run. And that's just something that they probably just played around with.

AUDIENCE: Do you use [INAUDIBLE] have to finish the decision losing a couple of minutes or 10 minutes or they're going to take your [INAUDIBLE] away.

PROFESSOR 3: In this competition there is different computation time budgets that you get. And I believe the reason for the different computation time budgets is the frame per second of the game.

I told you all about the setup, we went over, the scoring, the nodes, what the advantages are, what the simulation strategy is used. So you probably want to see it in action. So this is always a risky move trying to get video to play.

AUDIENCE: It's actually in the back up. Hit Escape.

PROFESSOR 3: OK. Got it.

AUDIENCE: And now, I guess, we--

PROFESSOR 3: And drag it over again?

AUDIENCE: Yeah.

PROFESSOR 3: Running this full screen.

AUDIENCE: Hit the [INAUDIBLE]

PROFESSOR 3: [INAUDIBLE] All right. Here's this MCTS-based "Mario" playing controller. You can see he's actually wrecking, so doing some serious damage here.

But those lines that you see, the reason they're different colors it's not showing different players, or anything like that. It's just using different colors so you can see the different layers of this tree search.

You can see he actually went backwards there. And that's because in a simulation, when one of the backward ones landed on an enemy-- and in fact gets you points from our scoring system versus if you had just gone forward you would have gotten some distance points but not-- also, he is just [INAUDIBLE]

The simulation is quickly being able to figure out that he can jump on all his enemies. So he's just wrecking all these guys. Getting lots of points here, collecting the coin, et cetera. You get the idea. It's pretty awesome to watch.

There's that flower we were talking about. So now he's actually a fire-spewing Mario demon. He's doing some serious damage with that. Stepping on missiles. I didn't even know you could step on the missiles.

All right. You could watch this for a while. But we'll exit now. It looks super promising in this video. I don't know how close max stuff.

AUDIENCE: Just click on back [INAUDIBLE]

PROFESSOR 3: There it is. OK. The demo looks really cool, looks really promising. Let's take a look at the charts here because we all want some quantitative stuff.

This is the chart. The score is on the y-axis. The bottom is computation budget, which is something that you were talking about. I just want to highlight to make this a little more visually appealing here.

All of these things that I highlighted, it's labelled as UCT. That's Upper Confidence Bound Tree. Remember, Yo talked about upper confidence bounds. That's essentially what's used in that TTS for guiding your tree search.

So these are all the methods. But then UCT multi, which is this purple square, that's saying it's using MCTS but it's doing the multiple simulations. And you can see this multi plus care is also in the top. Both these use the multi-simulation technique. And then the plus car is they added an extra scoring mechanism for carries. I believe that's probably like carrying a shell. That made it do better.

Then these ones that aren't highlighted are using plain Astar, and then a refined version of Astar. With increasing time, the do increase scores, but they're even worse than just your UCT with just random simulation, no multi-simulations.

We're running low on time, which is not ideal. But another thing that I want to point out is down at the bottom here, these are the multi-simulations. They have the lowest maximal search depth, which at first would seem like, what? I have the lowest search depth but my score is the most?

But that comes into play when you were saying about the trade off between the simulations and the amount time it takes. So because I'm doing multiple simulations, I'm taking more time at each node. But that's giving me a more accurate value assessment. So that let's me choose my actions more carefully, or with more information. And so that's what's able to give me this better scores.

That's all "Mario." So we're going to moving onto AlphaGo. Are there any questions about "Mario" before I go to AlphaGo? Yeah.

AUDIENCE: What's the table [INAUDIBLE] inference?

PROFESSOR 3: That's a good question. I have a feeling it's because if you're doing best of n, that's really heavily relying on your scoring metrics. Let's say at one step if I jump and collect a coin versus if I go left or right and play, I'll get more points if I get that coin. But maybe, a missile is going to hit me in the face if I do that. It gets rid of some of the-- it's forcing you to do certain moves.

AUDIENCE: Is the A* heuristically using the same value, the same value that you're getting by your simulation?

PROFESSOR 3: Yeah. I'm not exactly sure what the Astar heuristic is. The whole reason that A* is difficult is because coming up with heuristics for these types of games are. But this is not his version of Astar. I believe this is the Astar that was used by-- I forget the name of the guy-- but he won the AI competition a couple of years ago.

I'm going to try to move onto AlphaGo. Does someone have how many minutes I have left?

AUDIENCE: Four.

PROFESSOR 3: OK. We're going to power through. Here's AlphaGo. Hopefully, you all know the rules.

Just in case, I'll just go through a quick-- 19 by 19. You alternate black stones and white stones. You collect enemy stones by completely surrounding them. You can surround a single stone. groups of stones.

And your score is your territory plus the number of captive pieces. So your territory is just the area that you're surrounding, and then you just add the stones you've collected.

The rules aren't super important. The main emphasis is there's very few rules so you would think it's really simple. But the complexity of the game is quite extreme.

At each turn you have about 250 options that you can play. Each Go game lasts about 150 turns. So that gives you a total of 10 to the 761 games, approximately.

And to put that in comparison, here's chess. You can read those numbers. Chess is also pretty complex. But there's 35 options for turns.

Deep Blue. I think you were talking about building out the whole tree. So Deep Blue would build out the tree for six levels. And then use this hard core chess master inputted heuristic evaluation that it used to find the best move.

Except with Go, you have 250 options, which already is adding a lot more complexity. So that strategy won't work quite as nicely. What do we do?

We use a modified version of MCTS. Well, it's not what we do. That's what Google's DeepMind team did with Go. They combined neural networks with MCTS. Coincidentally, we learned about neural networks last class. Probably not a coincidence.

PROFESSOR 3: It's not a coincidence.

PROFESSOR 3: The we ordered two policy networks in the AlphaGo, and one value network. And another big coincidence here, the two policy networks are actually CNN's, which we learned specifically about last class, convolutional neural nets.

And the reason for that is the input to the policy neural networks is an image of the game. And remember, convolutional neural nets work really well with images. What it outputs, though, is a probability distribution over the legal moves.

And the idea is, that if a move has a higher probability it will be a more promising move for you to take. But another key point is that it's not deterministic. It's not telling you to take this move. It's just assigning a higher probability to this move.

And this network was generated by doing supervised learning on 30 million positions from human expert games. Apparently, there's a giant database of Go expert games. So that came in handy.

And there were two different networks trained. One of them was a slow policy, the other was a fast policy. The slow was able to predict an expert move with 57% accuracy, which to me was mind blowing. Using this neural network, 57% of the time it could pin where the expert would place his move. That took 3,000 microseconds.

Versus the fast policy, which suffered a bit in the accuracy, but it's 1,500 times faster. And we'll see where they used each of these different policies later on. But it could predict the expert move with 57% accuracy.

The other Go team was, that's not our goal. We don't want to predict an expert move. We want to predict a winning move.

And so to do that, they took their policy network, and then they would use reinforcement learning. That's where you play the network against iterations of itself in order to hone in a better policy that's geared towards winning moves. Then they tested this against Pachi, which uses-- for the camera, I have no idea if that's how you pronounce Pachi. It might be Patchey. I'm not sure. But there's 100,000 MCTS simulations at each turn. So this is purely MCTS.

If it were playing just the AlphaGo policy network, the policy network won 85% of the game. So without any sort of trained search or anything involved, it won 85%, which is pretty great. And that suggests that maybe intuition wins over long reflections in Go. And interestingly, if you talk to expert Go players and you ask them why they did a certain move, they'll just say, It felt good, or I had a hunch in this. That's indicative there.

Hopefully, I'm not going overtime. Sorry.

Those are the two policy networks. There's also a value network. What the value network does is it takes in a board, and they'll give you a value, like how good is this board? They'll give you a win probability number. So 77%, it would say, 77% of the time you should win from the board.

That's similar to the evaluation that comes from Deep Blue. But rather than a Go master coming in and telling you, well, if these are connected in this way, and down here we have this certain thing then here's the score we should expect, in chess, they had chess masters, like if the knight is here and the queen is here, all these specific things.

This was actually learned from the reinforcement learning that was happening when the policy networks were playing each other. The value network was learning about those positions during that time. And the predictions get better towards the end of the game, which I think Yo mentioned in his talk.

So how do you combine all these into MCTS? The slow policy network, if you remember, is slower but should give us stronger moves. It is used to guide our tree search in order to help us decide which nodes to expand next. When we expand that node to get the value, the value of the state is the simulation, like before, like normal MCTS, except it's not a completely random simulation.

We use our fast policy network to give us a more educated simulation here. But we're using a fast one, obviously, to save some computation time. It's giving us probably a more indicative random simulation of what's going to actually happen.

And then we also combine that with our value network output. So we run our value network on this node, as well. And we add that to our simulation value and we propagate it.

Interestingly, the AlphaGo team tested out just using the fast policy simulation value and scrapping the value network. And they also just used the value network and scrapped the simulation value. And those both performed worse than if it had these.

And another added interesting point here, is that these two factors in our value have about the same weight. They were both about equally important. I think I'll get into that later. But first--

AUDIENCE: Can I just ask a quick question?

PROFESSOR 3: Yeah.

AUDIENCE: So when you said the policy network is used, is that used when you're navigating to the tree to get to a leaf, or is policy network being used to do the simulation once you're at the leaf, or both?

PROFESSOR 3: The slow policy is done for this part. Then the fast policy is used for the simulation. Because the slow policy does take 1,500 faster than-- or the slow takes 1,500 times longer than the fast policy. You don't want to use that in your simulations. That would just take way too long.

It's basically just a way of making it so our simulation isn't completely random. It has some educated moves.

Why use policy and value network synergy? Why can't we just use the policy network? Why can't we just use the value network?

If we have the value network alone, we'll actually-- here's a side point. Remember, the value network learned from the policy network. And then also, later on, the policy network is improved by our values. They work hand-in-hand.

But if we had the value network alone, when we're deciding on it the next move, we're going to have to evaluate every single move, which would take forever. And so, what the policy network does is project the best move with a probably distribution. And it narrows our search space.

And then, if we had the policy network alone, we'd be unable to compare nodes in different parts of our tree. The policy network is able to tell us a distribution over which move we should take from a certain node. But then, if I ask it if I'm in a better position here than in some other place, it won't know.

That's where the value network comes in. It will give us an estimated number of the value assigned and open an evaluation of that node. And then these values are later used to direct our tree searches based on updating the policy once it realizes, oh, I thought this would be a good path but the value is this, so update all that.

Then why do we combine neural networks with MCTS? Remember, the policy network alone played against Pachi, which was purely MCTS, and it did pretty well. So how does MCTS improve our policy network? Remember, MCTS did win 15% of those games. So already, that makes you think there's something there that maybe the policy network is missing.

Also, the policy network is just a prediction. So by using this tree structure, we're able to use these Monte Carlo rollouts to adjust our policy to move towards nodes that are actually evaluated to be good.

And then, how do neural networks improve MCTS? The point should probably be clear by now. We're able to more intelligently lead our tree exploration. Our simulations are more reflective of actual games. And the value network and our simulation value are complementary, which I've mentioned before.

And just to highlight that, basically, the value network is going to give us a value that is reflective as if we've played the slow policy the whole time. And the simulation is if we used a faster policy. So they are complementary.

And I know I'm over time. So I just wanted to skim through the stats real quick. Distributed AlphaGo won 77% of the games against regular AlphaGo. So it's the only thing that beat regular AlphaGo.

And then distributed AlphaGo won 100% of the games against all these. In a rematch against Pachi, now that we've added MCTS to our policy network and we have our value network, we slaughtered Pachi 100%.

Then we decided to see how we fare against humans. And by we, I mean not me, I mean

Google. And they won 4 to 1. And Lee Sedol rating was 3,520. Now AlphaGo's rating is estimated to be about 3,586.

So you're like, whoo, we beat the best dude. Except we didn't because there's another dude who has an even higher score, apparently, 3,621.

This should be the last part. Here's this timeline. Basically, tic-tac-toe, checkers were conquered in '50. About 40 years later, we conquered checkers, chess.

Then we scroll down to 2015, is when AlphaGo was able to beat Fan Hui, who was a two-dan player, which is considered lower down in the tier of professional Go. But then, Lee Sedol was a nine-dan player. And he was able to beat him literally last month.

PROFESSOR WILLIAMS: So good job.

PROFESSOR 3: We're done.

[APPLAUSE]