.slide 1

Hello, everyone, welcome back to AI for games. Last week, we started talking about Monte Carlo Tree Search. Today, we're going to take a look at some possible adjustments of the base algorithm – but first, I want to introduce your final coding assignment, where you are supposed to implement – surprise, surprise –

.slide 2

Monte Carlo Tree Search. You will be doing so in the game of chess and using Unity, which I'm guessing you're all familiar with. But, on the off chance you aren't, don't worry, you don't have to do any modelling or anything like that, you are just going to have to program, the environment has been all set up for you.

Your task will specifically be to adjust the MCTSSearch and MCTSNode classes so that the program correctly performs MCTS.

.slide 3

This assignment is going to going to be split into multiple smaller assignments that you will get every week from today until the end of the semester, with the deadline always being the next week. Specifically-

.slide 4

You will be tasked with programming different parts of the algorithm every week, starting with the nodes used in the tree search and then moving on to the individual steps. Now, before you start looking for the nearest room with a ceiling low enough to hang yourselves from, allow me to clarify a couple of things. First of all, I expect that these assignments will be very easy for you, because they will *not be required to work*. In fact, you will have no way of knowing whether a part you've coded works before you assemble the entire algorithm. So, the only requirement for passing is that you send me code that *compiles*. That's it.

If that still seems like too much to you, you can also skip two of the four small assignments with no repercussions – skipping more result in failure however. Also, if you skip an assignment you have to also code the part you skipped when handing in your next assignment.

This may seem like a strange approach, so let me explain what the point of it is. There are three things that I hope to achieve this way. First of all, this algorithm isn't particularly easy to wrap your head around. By having to sit down and think about it over a longer period of time, I hope that you will come to understand it, at least up to a point, before you actually get down to trying to make it work. Second, the API isn't all that easy to grasp either – not because there's anything particularly complicated going on, but because you just have a lot of methods and properties to get familiar with. It therefore often happens that someone uses a method in the wrong way and this allows me to point that out to you before you spend hours or days trying to debug it. And the same goes for any problems with understanding the algorithm. And third, the algorithm is going to be mentioned again in a few lectures, so by engaging with it algorithm on a weekly basis, you should hopefully be able to remember how it works and better follow along.

I hope it's clear from this that, if you hand in your first draft of the algorithm next week and then not look at it for the rest of the semester, you fill fulfil the short assignments, but that approach will defeat two of the three points I just mentioned, so I don't advise doing that.

Now that I've explained the rationale, the plan is as follows: today, you will be tasked with implementing the node of the search tree that the algorithm will use. Then, the selection step, the expansion step, the simulation step and the backpropagation step. After all of that, you will be tasked with putting it all together. The final deadline will be the end of February, which should be something like two weeks or a week and a half after the end of the exam period. This deadline is fixed and will not be extended without sufficient reason. You are of course free to finish it sooner.

.slide 5

.slide 6

Ok, what follows is a description of the project you'll be working with, the API, etc. You can go through that at your own leisure. The API should list all the methods and properties you'll need to work with. If you arrive at the conclusion that you want to use some other method or property, it's probably safest to ask. Now, let's skip to…

.slides 7-39

.slide 40

the End Notes.

.slide 41

There's some general debugging advice here which you should read before trying to put the whole algorithm together, but what I want to mention here is this-

.slide 42

Your program will be evaluated using 15 tests where it will be supposed to checkmate its opponent. It will have a limited number of simulations at its disposal, and if it won't be able to find the correct solution within that limit, the test will fail. I tried to set the limits reasonably, but, since this is a stochastic algorithm and our implementations may differ, it can happen that your solution will not pass all the tests, despite being correct. You therefore *don't need to pass all the tests* before sending me the solution.

What's more, this is going to be a difficult assignment for most of you. It is therefore perfectly ok to contact me when you'll get stuck or when you'll need something explained more thoroughly. I created this assignment expecting that to happen. In fact, you should try to think about this like if you're working on an actual project in some company – there's going to be people who understand parts of the code better than you and you can ask them to explain it. Well, in this case, that person is me. Try not to wait for the final deadline though, because if you all start messaging me three days before, I probably won't be able to help you all.

.slide 43

Once you finish, send your solution to me on Discord or through e-mail. *Do not send me the entire solution*, just the files that you changed. And, of course, if you discover some bugs or things that could be done better, let me know. And that concludes my explanation of this assignment. Are there any questions?

…

If not, let me formally assign the first of these small tasks.

(switch presentations)

.slide 1

.slide 2

Which, as I said, will be implementing the node of your tree. So, you should think about what data should be stored in a node and what methods you want it to have, then write something that compiles and send it my way.

With that out of the way, let's jump into today's main topic.

(switch presentations)

.slide 1

Last week, I introduced you to MCTS and said that it can be applied to various different games and provide good results. However, the basic version that I showed you can only be applied to games that satisfy certain conditions. Today, we are first going to go over what those conditions are and I will show you some ways of adjusting the basic algorithm to be applicable to games that deviate from them. Then, I'm also going to show you a few ways of making the algorithm more efficient, given certain assumptions.

Now, before we dive in, I would like to explain the big picture – what's the idea here, why are we having this lecture? Because I am going to show you a bunch of algorithms today and you're going to learn them for the exam and then probably forget them before you get home, so what's the point? The takeaway that I hope you get from today is, first and foremost, that algorithms aren't set in stone and that if you encounter a problem that doesn't match an algorithm you know exactly, you can always try to adjust it, sometimes even in substantial ways. And, hopefully, you'll also remember at least some of the core ideas behind the algorithms I'll present here.

So, to start with-

.slide 2

let's have a quick refresher.

.slide 3

Monte Carlo Tree Search, or MCTS for short, is an algorithm for selecting moves in a game. It does this by a) performing random simulations to ascertain how good a move is – that's the Monte Carlo method –

and b) making progressively smarter move choices by biasing its move selection in favour of good moves, based on data gathered from previous simulations.

It has a main loop which operates in 4 or 5 steps – depending on whether you ask me or anybody else – and runs for some number of iterations and once it is finished, the algorithm picks a move. The 4 steps are: Selection – here the algorithm keeps selecting moves until it selects one which it hasn't tried before. Expansion – the algorithm performs the new move, creates a new child from the new state it reaches and appends it to the tree. Simulation – a simulation called a playout is run from the new state, which ends in some terminal state. This is then evaluated. And, finally, Backpropagation – the result of the previous step is backpropagated up the search tree and used to adjust the scores of nodes along the way.

Any questions about this?

…

Ok, with this out of the way, we can move on to discussing-

.slide 4

The domain of applicability of our algorithm. I.e., what kinds of games can our algorithm be applied to? What conditions do they have to fulfil?

.slide 5

Well, for example, the game has to be zero-sum. This means that when one player wins, the other has to lose. When one player gains something, the other loses something. So, the sum of the rewards is always zero. This is necessary because it means that we can derive the rewards of one player from the rewards of the other player. A counter-example would be some sort of competitive game where you are both trying to maximise your score, let's say – I couldn't find a proper example.

.slide 6

The game has to involve exactly 2-players. I suppose this one's rather obvious – if it doesn't, we can't straightforwardly switch between the players in the game tree.

.slide 7

It has to be sequential. This means that only one player can make a move at any given time, there are no simultaneous moves. If this condition isn't satisfied, it once again destroys the structure of our tree. A counterexample for this would be rock-paper-scissors.

.slide 8

Next, how about discreteness? If a game isn't discrete, meaning it isn't clearly divided into moves, we can't really build a game tree for it in a straightforward way, so that's definitely important. A counterexample would be any real-time game.

.slide 9

The game has to be deterministic. If it isn't, that means we can't predict what state a move will lead to, so, again, we can't build a game tree in a straightforward way. An opposite to this one would be any game that relies on dice throws.

.slide 10

The game has to offer perfect information, meaning there is nothing hidden. An opposite of this would be most card games, for example, since you can't see the cards of the other players.

.slide 11

And, last but not least, it has to be single-objective, meaning there is only one objective with respect to which we are trying to optimise. To be more precise though, the quality of having a single objective isn't really inherent to a game, it's more about what we are trying to achieve with our implementation. If you're not sure what this means, fear not, I will explain in more detail once we get to it.

So, these are the 7 conditions that a game has to satisfy in order for the basic version of MCTS to be applicable to it. Now, we are going to go over these one by one and try to tweak the algorithm so that it can be applied to games that break that rule. The approaches I'm going to outline here aren't going to work for every possible scenario, but they should give you some idea of how you can adjust this algorithm, and hopefully others, to suit different kinds of games.

.slide 12

So, to start with, we have the zero-sum requirement. Negating that, we get-

.slide 13

Non-zero-sum games.

.slide 14

So, what's the problem with non-zero-sum games? Well, our algorithm relies on the ability to derive the rewards of one player from the rewards of the other player. If one player wins, the other loses and vice versa. That's why we can store just a single reward value in each node. So, the solution to that is pretty simple – just compute the reward separately for each player and then backpropagate each value. That's it, problem solved. Moving on.

.slide 15

Next, we have the 2-player requirement. There's two ways to negate that.

.slide 16

.slide 17

We can either have single-player games or games of more than two players. First, let's take a look at single-player games. The way to solve this issue is once again quite simple – the only problem to tackle is that, in basic MCTS, we alternate between the perspectives of the two players when picking moves. So, here, we dispense with that.

The multiplayer case is a bit more interesting. First of all, this obviously means that we have to switch between all the players in the search tree, not just two. Second of all, it means that, like in the case of non-zero-sum games, evaluating states from the point of view of one player is no longer sufficient, since we can't derive the rewards of all the other players from that. So, if we have *n* players, we have to compute the scores for at least *n* – 1 players and store all of them in nodes. However, there is a caveat.

.slide 18

What about when we have too many players to explore the tree sufficiently? For example, let's say you're playing a game of 5 players. That means that you pick a move for the player that you are trying to model on every fifth level and, if there are many moves or harsh time constraints, you may not even get deep enough to do that twice. So, what to do about it?

Well, one thing you could try is using the so-called 'paranoid assumption'. This means that you assume that the other players have all formed a coalition against you. There are different ways you can utilise this in your algorithm – for example, one minimax-based program, aptly named 'Paranoid', uses this to pick moves of other players such that they minimise the utility of the player that the program is modelling.

That probably wouldn't help much with MCTS, though – you'd still have to pick moves for all the players. However, experiments with these kinds of programs showed that letting them reach deeper into the search tree only improved their performance when they reached another layer where the modelled player's moves were chosen. This seems to suggest that, in such games, you aren't all that interested in modelling the other players' moves correctly, it's your moves that you are interested in.

.slide 19

Inspired by this, some researchers created a new approach called 'opponent move abstraction', or OMA for short.

This adjustment works as follows. You build the search tree as usual, but when you backpropagate, you aggregate the moves that were preceded by the same moves from the modelled player. So, you take the entire sequence of moves that lead up to a given state, throw out all the opponents' moves and compare what remains. And if the sequences are identical, then you aggregate the results. Let me show you an example.

.slide 20

Here we have a hypothetical game tree constructed for a game of 3 players. ROOT is the player that we are modelling and then there are two opponents. Now, let's say that we got to either state 1 or 2 and then picked move a. With normal Monte Carlo Tree Search, the value computed for these two uses of move *a* would be completely independent, because we got to them through different paths. One path consists of moves *x*, *m* and o, the other of moves *x*, *m* and *p*. However, what OMA does is it only takes into account the moves of the modelled player when doing backpropagation. So, we do not care that we performed moves *m* and *o* to get to state 1 and moves *m* and *p* to get to state 2. We only care that this move was chosen after the root player chose move *x*. Therefore, whenever we perform move *a* here, we update its score here as well and vice versa. We do not, however, update its score in state 3, as this move *a* was preceded by move *y* for the root player.

.slide 21

These values are computed separately from the UCB values and are then combined in a straightforward way. Here you can see the original formula and beneath that, the new version. The Q^{OMA} part is the new value we've been talking about and beta is a tuneable weighing parameter. It doesn't have to be a fixed number and it's actually a good idea to set it up to decrease with number of simulations, so that, with increasing number of simulations, the algorithm relies more on actual mean rewards, rather than the OMA estimates, thereby guaranteeing its convergence to standard MCTS in infinity.

Ok, so that's OMA and multiplayer games. Moving on, we have-

.slide 22

the sequentiality requirement. And the negation of that gives us-

.slide 23

simultaneous-moves games.

.slide 24

At first glance, you may think that using normal MCTS wouldn't be such a problem here, you just apply the players' moves sequentially, is that really that bad? But if we take the example of rock-paper-scissors – which is a simultaneous move game, since both players reveal their choices at the same time – applying this approach would mean that (draw this on the board), first we would pick the move for one of the players and then for the other. But this would mean that the second player would have knowledge of the first player's move and could therefore always choose the optimal response. So, that wouldn't work, we have to come up with something else.

Well, let's consider where the problem is. In normal MCTS, when we are in some state, we move to a different state by deciding on the move for one player. That means we have one variable that determines where we go next and we can therefore store the next moves in a 1D array. In a 2-player game with simultaneous moves however, we have not just one, but two variables, so that natural way of thinking about that would be a 2D array. So let's try that.

.slide 25

.slide 26

So, this is our regular tree and this is what that our new tree node looks like.

.slide 27

And, when we apply this change to our data structure, we can pick moves by computing the UCB values for both players separately.

.slide 28

This works, to an extent, but isn't guaranteed to converge to a Nash Equilibrium – if you don't know what that is, it basically means that the algorithm isn't guaranteed to find the best, most robust strategy and

can therefore be exploited. This can be solved by using a different tree policy than UCB, but such policies are a bit complicated and I don't see a reason to go into them here, just know that it can be solved.

Alright, next one up is-

.slide 29

The discreetness requirement. Negating that, we get-

.slide 30

Continuous games. There are two ways of interpreting this –

.slide 31

Real-time games, i.e. games which aren't split into discrete turns, and games with continuous action spaces. An example of the former is any RTS game – the clue is in the title there – but also first-person shooter and MOBA games, to name a few. An example of the former could be minigolf – you can shoot your ball in any direction, so the action space is continuous. Technically you could also count, say first-person shooters in this category, but in many of these it's quite easy to discretize the action space, because it's pointless to shoot in most directions – you just want to hit the enemy players.

I'll take a look at real-time games first and then return to games with continuous action spaces. So, when we need to take time into consideration, the question becomes-

.slide 32

When do we actually do our planning? In discreet games, you don't have to worry about that, because nothing's happening while you do your thinking. But in real-time games, things are happening all the time. We could sidestep this by running a search every frame, but for many games, that would be way too costly and it would give us no plan persistency. The latter problem could be solved by caching the plans, but then you have to ask under what conditions the plans should be invalidated, which basically brings us back to where we started.

So, we need a different approach – the one that I'm going to show here was created for a single-player setting, but I expect it should be possible to tweak it for multiplayer as well.

.slide 33

This one is called Variable Simulation Period MCTS, or VSP MCTS for short. The basic idea is quite simple – instead of just choosing actions, you also choose the time that you will have to deliberate afterwards. How is all this implemented though? Well, that's where it gets complicated.

So, first of all, we are now dealing not with *actions*, but with *pairs of actions and time periods*. Our search space is now continuous, so we can't just somehow enumerate all the possibilities. So, we are going to use an approach called 'progressive widening'. What this means is that each node of the MCTS tree has a maximum number of children that is limited by the number of times it has been visited. The authors of the paper don't mention the specific formula they used, but it can easily be something as simple as adding one child for every *n* visits, or using a formula with an exponential fall-off, or something of the sort – this is definitely something to be determined experimentally on a case-by-case basis.

Ok, so we limit the number of children. Now we have to decide which actions and time periods to assign to the children. These two things are done separately. I wasn't able to find a specification of how the authors pick actions in the original paper, but I assume the easiest way would be to just pick them at random, or to start at random but make sure each action gets picked at least once, then at least twice, etc.

Picking the time interval is trickier, because here, we are dealing with a continuous domain. To tackle this, the authors use something called the HOO algorithm, which works is as follows: (draw this on the board) You are starting with a time interval that is bounded by 0 on one side – you can't have negative time – but unbounded on the other. First, you set the maximum possible amount of time, which gives you a closed interval to choose from. Now you start building a tree, where every node contains an interval and its corresponding reward. So, you start just with the root node, which contains this entire interval. Then, you randomly sample this interval – this sampled number will be your chosen amount of time – and you split it in half. This gives you two new intervals, one of which contains the value you just sampled, which means the root node gets two new children. Each of these children will have a mean reward associated with it, which will be computed from the mean observed reward of simulations, just like with MCTS. So, every time you need a new time interval in a given MCTS node, you query an instance of this tree. In it, you follow a policy similar to UCB, but a bit different. I won't go into the specifics here, because that's beyond the scope of this lecture. You land in some child node, randomly sample its interval, subdivide it into two and return the sampled value.

.slide 34-36

.slide 37

One more thing to add here – I promised to return to games with continuous action spaces. In these games, you can also use the HOO algorithm combined with progressive widening to generate new actions from the action space. And if you have multiple continuities, you can use a multidimensional version of HOO.

Ok, next up, we have-

.slide 38

Determinism. The opposite of which is-

.slide 39

stochasticism, or randomness.

.slide 40

So, how do we approach this? Well, the first idea we might have would be to just take all the possibilities into account. So, let's say that the game we are playing somehow relies on the throw of a six-sided die. This would mean adding one child for every outcome to our tree and always selecting from among them randomly instead of using the tree policy. And this can work for some games, but if there's too many possible outcomes of a random event, the branching factor becomes so large that this just falls apart. If you take some card game like Solitaire for example, this would mean taking into account every possible configuration of cards.

Ok, so, if that is the case, maybe we could-

.slide 41

only take some possibilities into account, thereby limiting the branching factor. We could either do this by discarding some actions altogether, or using the progressive widening technique which I already mentioned in the previous section. In experiments done with Klondike Solitaire, this didn't improve the performance of the algorithm much – it did improve its speed, however, so it should be able to perform more simulations in the same amount of time and therefore achieve better performance.

But, we can do better.

.slide 42

To do so, we need to try a different approach. What we can do, is first create different determinizations of a game – that is, configurations of the game with every random event fully determined in advance. So, in our Klondike Solitaire example, this would mean determining the order of the cards in the deck. And, once we have several of these determinizations created, we can then run separate tree searches over all of them and then average the results. This approach outperformed both of the previously mentioned approaches in Klondike Solitaire.

Alright, next up-

.slide 43

The perfect information requirement. And, if we negate that, we get-

.slide 44

Imperfect information games.

.slide 45

Ok, so, it might occur to you that this actually quite similar to games that involve randomness, which we just discussed. Both involve processes that you can't predict. However, the difference is that, in games that involve randomness, you don't need to model the player's decision-making over hidden information.

We can get around this though, as it is actually possible to model the opponent by creating a probability distribution which will be trained on data from whatever game you're constructing an AI for.

.slide 46

For example, in the previous lecture, I mentioned that MCTS has been successfully used in Texas Hold 'em Poker, which is a classic example of a game with hidden information. That's the approach that the people who created the bot for that game used. They trained two models on data from actual poker games – one for predicting the opponent's next move and one for predicting what cards they would show at the end of the game. And that allowed them to treat these nodes as just random nodes.

Now, this could work with the first approach we mentioned in the previous section – just having nodes in the tree which you sample randomly. However, it does create a bit of a problem when using

determinizations, because, if I use the example with Poker, determining what cards every player has does not determine what actions they will take. I suppose you could try to also determine that but that would probably require a different opponent model to be feasible.

But creating an opponent model from player data can be quite a bother anyway, especially if you're creating a new game. So, is there a way to do this without resorting to that?

.slide 47

First off, let's just try applying the other approach we discussed for tackling randomness here. We create multiple determinizations, create one tree for each of these, then average their results. The big problem here is, that this gives you access to information you shouldn't have – sort of like what we discussed with rock-paper-scissors in simultaneous move games.

Imagine, for example, that you are playing a game of UNO. (draw this) Both you and your opponent have two cards in your hands. You have a +4 and, let's say, a 7, and you don't know the other player's cards. Now, let's say that, in one determinization, the other player will have a +2 and something else, let's say a 5. In the other determinization they have a stop card and something ordinary again, let's say a 3. All the cards have the same colour, let's say blue. It's your move, so you have to decide what to do.

In the first instance, the algorithm will figure out that it's best to first play the 7, the opponent can then play their +2 and you can then wreck that man's whole career by finishing with a +4. In this instance, however, it's more beneficial to start with the +4, making the other player draw cards and then immediately following up with the 7, because, otherwise, they could stop your move and finish first.

In both of these cases, you come to the conclusion that you have a guaranteed win, but that's nonsense of course, because it relies on you knowing the other player's cards. This is something called *strategy fusion*.

.slide 48

To tackle this, I will first have to define a new term – *information set*. An information set for a player is the set of all states of the game that are indistinguishable for that player. So, say you're playing Poker with one other person. You have some cards in your hand, the other person has some in theirs and there are some cards on the table. The game state is defined by the identities of all these cards – and of course the amount of chips that each of you has and that have been bet thus far, but we can ignore that. Now, you can see your cards and those on the table, but you can't see the other player's cards. So, all the game states that differ just in the identities of those cards are indistinguishable to you – and the set of these states is one information set. So, it's a partitioning of the state space into classes of states that are the same from your point of view.

.slide 49

Now, with this, we can define Information Set Monte Carlo Tree Search, or ISMCTS. In this algorithm, instead of having nodes correspond to states, we'll have them correspond to information sets. And the trick to dealing with them will be that we'll create a new determinization at the start of every simulation – which, again, means that we'll define the entire state of the game – and run the simulations using this determinization. So we just build one tree, but that tree will aggregate information from playthroughs with different determinizations.

We can see how this helps solve the issue we had before, because our new tree for this case would look like this (draw again). These two states have now been fused into one information set, as have these terminal states and we now get the correct result.

Ok, this is an improvement, but it still isn't perfect, because it treats the opponent's moves as fully observable – which doesn't have to be the case.

.slide 50

We can solve this by generalising the moves too, in a similar fashion. This gets rid of the strategy fusion problem, but now we're left with a different issue – we can no longer model our opponent's decision-making, because we no longer know what decisions they are making. Because, in standard MCTS, we model good decision-making both for us and our opponent and these two influence one another. But, if I can't know what sorts of decisions the other player is making, then I can't model what good decision-making looks like. And I would actually expect this to be a problem even for ISMCTS that doesn't abstract the moves, because the opponent gets dealt different hidden information at the start of every iteration, which should influence their decisions – so, if there are lots of possible determinizations, we may not get to try any single action multiple times.

So, we have two issues here which are seemingly complementary – you either have a good opponent model and access to the opponent's information, or no access to the opponent's information and a bad opponent model. Unless you start using multiple trees.

.slide 51

This extension of the algorithm is appropriately called Multiple Observer ISMCTS. The idea is that you create multiple search trees like the one I just described, one for each player, and you use the data stored in the tree of the corresponding player to choose their moves. So, let's say that you have two players, which means you build two trees. You start traversing them and it's player one's turn at the beginning. You choose a move for this player based on the data stored in their tree. Then, you choose a move for the other player using data stored in their tree and so on. These choices are completely disconnected, so one player has no idea what the other is doing, but they can still inform each other's game outcomes.

Alright, this one is again better than the previous one. But. The opponent model still suffers from the fact that the nodes correspond to information sets, instead of game states. That means that, when you have a given determinization – which determines what cards or other information the player is hiding – it will make decisions based on the same data as other determinizations where the hidden information may be completely different. So, for example, let's say you're trying to simulate poker. In the first determinization, the opponent gets two aces. In the second, they get a six of clubs and a nine of hearts. The opponent would, presumably, play differently under these two conditions. But the algorithm will reuse the same tree for both of them. So, for a given node, it may say 'I've already tried this one, it was no good, let's try something else', even though the context, meaning the player's hidden information, was completely different.

To combat this, we can make one more adjustment –

.slide 52

We can construct a tree not just for every player, but for every information set. This means that we'll have an accurate opponent model for all possible situations. This adjustment is called Many Tree ISMCTS and allows us to do two things that are essential in many games of hidden information – inference and bluffing. To be clear, I wouldn't say that these were completely impossible to do with the previous approaches, but the techniques I will now describe almost definitely wouldn't work as well there, if at all.

.slide 53

So, let's start with inference. This means figuring out the player's hidden information based on their actions – or at least figuring out what they are more or less probable to be hiding. To do this, we first have to make the search trees persist across turns. So, once your turn ends, you don't throw away your search trees, but keep the subtrees that have roots in the new game state. After we do that, the approach is simple – keep track of how many times a node is encountered when a given determinization is used. That gives you a probability that a given determinization is true, provided that the action that led to that node is picked. This can then be used to bias the generation of determinizations in favour of the more likely ones.

So, to illustrate this, let's say you only have three possible determinizations of a game, (draw this) *d1*, *d2* and *d3*. And let's say that, in one of your opponent trees, the root node has two children, *n1* and *n2*. When using determinization *d1*, this node was picked 10 times. When using *d2*, 5 times and when using *d3*, 26 times. From this, you can probably see that if the opponent picks the action which leads to *n1*, it makes it most likely that *d3* was the correct determinization.

I won't show you the actual formula used by the authors of the paper here, because, again it's kind of complicated and beyond the scope of what I want to show you here, but I hope you can see that this is possible to do. Ok, so, that's inference. Now, let's move on to bluffing.

.slide 54

The problem here is that, in the default version of ISMCTS and its variants, you only sample determinizations that you know to be possible from your point of view. So, if you're simulating a card game, you don't sample determinizations where you have different cards than you know you do. But this, of course, prevents bluffing, because you don't consider the other players' points of view at all. To remedy this, it's necessary to take these so called *self-determinizations* into account. This means that you'll also create a new search tree for every determinization where your hidden information differs.

If you just do this enough times and then average your results from different trees, it should give you bluffing out of the box. However, it is possible to speed things along by changing the way the algorithm picks moves. Namely, you can look at the moves that were picked in the search trees created for self-determinizations and pick the one that was most visited across all of them, provided that it's expected reward isn't too much worse compared to the actual best action.

That's bluffing. Now, I should probably mention at this point, that this whole MT ISMCTS approach has one very obvious weakness – namely that, for games with many possible determinizations, you'd need to create a potentially enormous number of trees. So, while this is the most theoretically sound model, it probably isn't the way to go for the likes of Poker, for example.

Alright, I've glossed over a lot of details here, but hopefully, you get the general ideas. This is a fascinating area of research and, as far as I know, one that hasn't been explored that well yet, so if you find this interesting, I'm sure there's a diploma thesis topic to be found in there somewhere.

And that's it for games of imperfect information. And, last but not least, we have-

.slide 55

The single-objective requirement. The negation is, of course,

.slide 56

MCTS for satisfying multiple objectives.

In many games, AI has to juggle multiple objectives. We can take RTS games as an example – you have to gather resources, build units, build buildings, update units and buildings, expand your tech tree, etc. All these are competing objectives – if you build a unit, you won't be able to use the resource you paid for it to get a new tech upgrade. Or, to take a simpler example, you can have a game where, instead of just trying to beat every level, you may try to do so in the shortest amount of time while also gathering all the collectibles. So, it can be important for our AI to somehow be able to take all this into account.

.slide 57

This MCTS adjustment for tackling this is only meant for single-player games and, as I'm not too familiar with some of the concepts behind it, I'm not sure how it could extended to two-player or multi-player games, but I expect there must be a way.

Ok, so, we have multiple objectives and that means multiple reward functions. This means we have to backpropagate all the rewards. However, we can't just do what we did with multi-player games, where we kept an estimate of the mean reward at every node. Or, maybe we could, I haven't seen anyone try that, so I'd hesitate to say with certainty that it doesn't work at all. But the problem you have to address here is – how do you adapt the UCB formula? Say that you have two objectives and choosing one action totally satisfies one of them but ignores the other and another action satisfies them both a little, which should you pick?

So, the authors of this algorithm did something different.

.slide 58

In every node, they keep track of what is called the 'Pareto front' of reached end states. Like Nash equilibrium, this is another concept from game theory. If you're not familiar with it, you can think of it as the set of best solutions. Why a set and not a single solution? Well, imagine you have three objectives. (draw this on the board) In one state, the objective A is satisfied this much, objective B is satisfied this much and objective C is satisfied this much. In another, A is satisfied this much, B this much and C this much. You can't say that one of these solutions is better than the other, according to the defined objectives. However, if you have another state where A is satisfied this much, B this much and C this much, then this state is clearly worse than this one, right?

So, the Pareto front is the set of best solutions. This is kept track of in every node. Now, when a new reward vector is backpropagated, there are three possibilities. Either the node already has a reward

vector that is clearly better in its Pareto front. In that case the reward is discarded – this is a big difference to the original version! (The number of visits still gets updated though.) Or there is no such reward vector, but neither is the new reward vector better than any of the ones present in the Pareto front. In that case, it just gets added and backpropagated further. Or, last option – the new reward vector is better than one or more of those present in the node's current Pareto front. In that case, it kicks out the others.

Ok, now that we've got that sorted out, how do we handle UCB? Instead of using the mean reward, we use something called Hypervolume Indicator, denoted HV. I won't write the formula because it uses some advanced math wizardry, but, basically-

.slide 59

if you imagine you have two objectives, you can plot the potential solutions on a 2D graph, like here. The blue dots are the potential solutions which somehow balance the two objectives, the blue line is the Pareto front and the area underneath the curve is the Hypervolume Indicator. And you can hopefully see that the better your solutions are, the higher HV is going be, right? If we take an extreme example, if you had a solution that would completely maximise both objectives, it would be up here and the HV would just be this entire square.

So, this gets used in the tree policy instead of the mean observed reward. And that's it.

.slide 60

With that, we have successfully cleared all the requirements laid out at the beginning of this lecture, meaning we can move on to the second part-

.slide 61

Where I show you some ways of making the basic algorithm more effective. An obvious way of doing this is applying domain-specific heuristics or playing around with move-selection strategies, but that's not what we're going to be looking at here.

.slide 62

.slide 63

Let's start with something simple. I hope you know of Alpha-beta search which is just minimax search, but made more efficient by ignoring parts of the tree where it becomes impossible to find a better solution then the best already found. The problem with applying this to MCTS is that the rewards are constantly changing. But, once we get deep enough encounter terminal states, which have fixed values, there's no reason why we shouldn't apply the same principal.

.slide 64

For nodes that don't correspond to terminal states, we can compute their bounds by taking the maximums or minimums of their children, depending on the player. Besides making the algorithm more efficient, it can also help it deal with trap states – which, as I said last time, are something that MCTS struggles with.

.slide 65

Here we have some pseudo-code, we don't need to go into that.

.slide 66, 67

.slide 68

Next up, we have All moves as first, or AMAF for short. This is a clever one.

.slide 69

So, picture a game of Go. When we run simulations of this game, we make sequences of moves which, when reordered, can still make for a valid playthrough of the game – that's not to say that any reordering will be valid, but some will. So, that means that, when you perform a single simulation, you actually get the results of simulations where you perform those moves in a different order as well and can therefore update their values too.

.slide 70

Here we have an example. Let's say that the algorithm starts by picking move C2, then A1, then moves B1, A3 and C3. We can reorder this to C2, A3, B1, A1, C3 – so we update A3. We can also have B1, A1, C2, A3, C3 – so we update B1, etc. Simple, effective, elegant.

Moving on, we have-

.slide 71

combining MCTS with transposition tables and Directed Acyclic Graphs.

.slide 72

So, in MCTS, we are building a game tree, but games can often form graphs with multiple paths to some states – meaning that one can arrive at the same state by different game trajectories. If that is the case, then it would be a waste not to share information between copies of the same state in the game tree built by the algorithm. But how to go about doing that?

Well, the first step is to just keep track of what states we've already seen and every time we encounter a new state, we check if we've already seen it somewhere and if so, we take it into account. We do this using so-called transposition tables. These are tables that store the hash values of our states and for every state we encounter, we compare its hash value to the ones stored there. If we find a hit, we create a new connection to that state.

.slide 73

Now, let's think about where we should store the data. In basic MCTS, the data are stored in the nodes, simply because that is more intuitive. We could also store data in the edges, but, since, in a tree, we have one edge per node, that wouldn't make a difference. It does make a difference here, however.

In the paper, they simply say "storing data in edges is more general, so let's go with that". And this is true, as we can see here, but it doesn't seem like a good enough reason to me, especially since it will introduce a big problem as we'll later see. But, since I don't know of any paper which would examine the properties of storing data in nodes here, that's what we have to go with too.

One problem that I could see with storing the data in the nodes is handling the case when you first make a new connection to a state that you have already encountered. Because, if you're storing data in the nodes, do you backpropagate the data already present in that node? Either way would probably be problematic – in a similar way to what we shall now see with edges.

.slide 74

Ok, so we are storing data in the edges. Now, how do we perform backpropagation? There are two options – we can either update along all the paths from the leaf node to the root, or only the path along which we descended.

The first option seems like a good idea, since it should give us more info per playout. But let's inspect it just to be sure.

.slide 75

.slide 76

Here we have an example of a game tree where we've run 4 simulations. Say we are starting a new iteration of the algorithm. What happens?

.slide 77

We start with the left edge. But, the info from there will be backpropagated along the right edge too. So we will keep selecting the left edge and never arrive at the optimum. So, sadly, this strategy isn't really useful, we have to stick to updating only along the path along which we descended.

.slide 78

.slide 79

Ok, now how about selection? If we just used normal selection while storing data in the edges, that would mean we would lose much of the advantage of building a DAG instead of a tree, since we wouldn't be sharing information about moves between edges that lead to the same state.

For example, consider the graph we have here. We have enough information to conclude that node *b* is probably better than node *c*. However, if we are in node *a*, we don't access this information when using regular UCB. So we need a different approach.

The formula that we'll use looks like this.

.slide 80

.slide 81

.slide 82

.slide 83

I know that all this stuff on the left probably seems very complicated, but the idea is quite simple. The μ_{d_1}, p_{d_2} and n_{d_3} values again stand just for mean reward, the number of parent visits and the number of node visits respectively. The difference to regular UCB is that, instead of being computed from just one

node, they are computed from a number of nodes that are the descendants of the node from which we are starting the computation. And the level up to which the nodes are taken into account is given by the indices d 1 through 3.

So, imagine we just have a tree instead of a graph again. We have a root node, it has 3 children and each of those again has 3 children. Setting all these indices to 0 would just mean normal UCB, we just compute it using the data stored in this node. Setting them to 1 would instead mean combining the values in these three nodes and averaging them. And setting them to 2 would mean taking the values of these 9 "grandchildren", summing them up and averaging them. And, in a tree, these three approaches should give almost identical results – the only difference is that the parent node always has one more simulation then all its children combined, the one that started from that node when it was visited for the first time. But, when we are dealing with graphs, that's no longer the case. And if we take a look at our previous example, we can see that-

.slide 82, 81

using this approach with just a one-level lookahead, we can get a better picture of how good node b and c are.

That probably still sounds convoluted, but once you'll take a look at the formulas, it should start to make sense.

Ok, so that's it for transposition tables and DAGs.

.slide 82-84

.slide 85

And for the whole lecture. Thank you for coming, tomorrow we are going to take a look at the complexities of AI for real-time strategy games followed by a lecture on the same topic next week. See you then.