.slide 1

Hello everyone, welcome back. Today, we are going to go over some adjustments of the Monte Carlo Tree Search algorithm that I introduced to you in our first lecture. As you can see, this lecture is called MCTS adjustments part 1, so next week, we are going to go over some more. I tried to split them in the following way – today, we are going to go over adjustments that are supposed to improve the algorithm but don't change what sort of games it is applicable to. And next week, we will take a look at some ways of altering the algorithm so that it is applicable to different types of games than the basic version.

By the way, the reason why I decided to pay so much attention to these adjustments is that, hopefully, they should teach you something about AI programming in general, so that, even when you'll be working with different algorithms, you'll be able to draw on the knowledge gained from this.

.slide 2

Here is a list of the adjustments we'll go over today. There's 15 in total, which may seem like a lot but don't worry, many of them are quite short and simple, it shouldn't even take more than an hour to go through all of them. If that still seems like a lot, then-

.slide 3

-here's a picture of a happy goat to console you.

.slide 4

Ok, now, let's start with a little refresher of what we went over in our last lecture.

.slide 5

We discussed Monte Carlo Tree Search, which is an algorithm for picking moves in games in a smart way. It consists of four-ish steps that you can see here. We start with the selection step, where we recursively pick moves using some strategy until we pick a move which we haven't tried yet. We then move on to the expansion step, where we perform the move that we selected, which gives us a new game state, we create a new node from that game state and append it to the tree. Then, in the simulation step, we run a simulation starting from the newly created node. That simulation ends in some final state, which we then evaluate to get a numeric score. Finally, in the backpropagation step, we backpropagate that score up the tree and use it to update the values in all the nodes.

.slide 6

We also mentioned that selecting a move in this algorithm can be considered an instance of the Multi-armed Bandit problem. And, since this problem has been around for a while and has a comparatively long history of research behind it, we said we can take advantage of that and use strategies devised for this problem in MCTS. Specifically, we discussed the UCB formula for picking moves.

.slide 7

Here it is. And we mentioned that it consists of two parts – the left part is just the average reward for a move, so that encourages the exploitations of moves that we already consider to be good and the right part encourages the exploration of new moves. Before we move on, are there any questions? Because if

you don't understand these things, you probably aren't going to get much out of the rest of what I'm about to say.

…

Alright. By the way, since I'm on this topic, back when I was in your shoes, I always tried to avoid wasting time by sitting at lectures that I knew I wouldn't understand, so, if you haven't yet familiarised yourself with the MCTS algorithm, trust me, you won't much of the rest of this lecture, so it's perfectly alright for you to leave, no hard feelings.

With that out of the way, let's move on.

.slide 8

So, you may remember that I started our first lecture by giving you a pitch about how versatile the algorithm is and how it's been successfully applied in many different games. It isn't a silver bullet however and the 'successful application' I talked about usually entailed adjusting the algorithm in some way. So, let's talk about what some possible complications of using this algorithm are. Some of the adjustments I will go through will be meant to help with some of these complications.

.slide 9

How about a high branching factor or severe time limit? That's a big one because our algorithm relies on sufficiently exploring all the moves, so if there are too many moves to do that in a reasonable amount of time, we can't expect it to give good results.

.slide 10

Another potential problem is when games are long. This means long simulations which a) take time and b) means that the final states have very low probabilities of actually occurring, so they aren't too useful.

.slide 11

Another one is that games may not simply end with a win or a loss – sometimes we care about some final score for example, so dealing just with zeros and ones is insufficient.

.slide 12

And, last but not least, this algorithm has a very hard time identifying trap states – that is, states that don't appear bad if one looks at the near-term future, but contain some trap that will become apparent later on. They can be commonly found in chess for example.

.slide 13

Ok, now that we have that out of the way, let's start with the adjustments. Starting with-

.slide 14

Adjustments that change the selection step. The first of which is-

.slide 15

UCB1-Tuned Tree Policy.

.slide 16

As you may have guessed from the name, this adjustment fine-tunes the UCB policy so that it performs even better. It does this by replacing the right part of the formula which, as you may remember, is responsible for encouraging exploration.

.slide 17

It replaces it with this unpleasant looking thing. I tried to find out how the equation for this was derived, but was sadly unable to do so, so we're just going to have to believe that it works. But what this is supposed to do is provide a tighter upper bound on the value of a move and therefore make our algorithm more efficiently balance between exploration and exploitation.

.slide 18

.slide 19

.slide 20

Next up, we have SR+CR MCTS.

.slide 21

To explain this adjustment, we first have to go cover the notion of expected regret. To put it simply, expected regret is the difference between the expected reward of the best possible choice you could have made and of the choice you actually made. We can concern ourselves with two types of regret – simple regret, which is computed over a single action, and cumulative regret, which is computed over multiple actions. That is what the SR and CR in the name of this adjustment stand for.

.slide 22

Regular UCB is meant for the Multi-armed Bandit setting where we want to minimize the cumulative regret over all the arm pulls. In MCTS however, we only care about picking the best move once, at the end of the algorithm. Minimizing cumulative regret at the root can therefore cause us to focus on moves we already consider to be good too much, so we should minimize simple regret instead.

.slide 23

There a few policies for this, the epsilon-greedy policy for example, which chooses the action with the highest mean observed reward with probability ε and chooses any other action with probability $\frac{1}{n-1}$ where n is the number of possible actions. Another such policy is UCBsqrt, which is the same as UCB, except the logarithm on the right side is replaced with the square root function, which puts more emphasis on exploration. It is important note here however, that this policy is *only used when picking a move at the root*. In all other nodes, normal UCB, or some other cumulative-regret-minimizing policy is used. This is because the rewards accumulated in all nodes except for the root affect decision making in the nodes above them, so focusing on exploration and not caring about the rewards much would throw off the whole algorithm.

.slide 24

Ok, next up, we have First Play Urgency.

.slide 25

The first two adjustments I mentioned were a bit more complex, now we're moving to the less complicated ones. First Play Urgency is a simple little adjustment meant to combat the problem of UCB and high branching factors. Instead of having the algorithm try to play every move at least once, it sets the initial evaluations of all moves to some hand-picked constant x, allowing for early exploitation.

Moving on, we have-

.slide 26

Progressive bias.

.slide 27

This is a technique for combining UCB with a domain-specific heuristic, so, let's say you are trying to apply MCTS to chess and you have a heuristic for evaluating how good individual states are. You could use this technique to combine that heuristic with UCB and the effect would be basically that, while the number of simulations containing a given move would be low, it's score would mostly be given by the heuristic and the more times the move would be tried, the lesser it's value would be.

The way Progressive bias does this is very straightforward – it adds the heuristic value to UCB while multiplying it by a discount factor.

So, that's Progressive bias and the last selection adjustment that we'll be talking about today is-

.slide 28

.slide 29

Move groups.

.slide 30

To explain what this adjustment is about, I first need to introduce a game called The Amazons. This is a game for two players, each of whom has 4 pieces at their disposal. These pieces move just like queens in chess – so, they can move to the right, left, up, down and diagonally and they can move any number of squares but can't jump over other pieces. A single move consists of two parts. The first is moving one of the pieces. The second is shooting an imaginary arrow from that piece. This arrow can move in the same way that the pieces can and the square on which it lands gets blocked until the end of the game, so neither pieces nor arrows can pass over that square from then on. The game ends when one of the players can no longer move any piece.

.slide 31

So, when applying MCTS to this game, the first idea might be to just model the two parts – moving a piece and shooting an arrow – as a single move. But, as we'll see, that may not be the wisest choice.

.slide 32

To see why, we have to do some quick computations. Let's say we have *n* legal positions to move to and *m* legal positions to shoot to. The upper bound on the number of possible moves at a given node is then *n* * *m*.

.slide 33

Now, let's consider a different approach. Let's say that we split the move into its two constituent steps, so the algorithm will first pick where to move a piece, then where to shoot an arrow, instead of picking both at the same time. In this case, we first choose from among *n* nodes, then *m* nodes. Therefore, instead of picking the best action out of *n* * *m* options, we are basically picking it out of *n* + *m*.

Of course, we aren't picking actions randomly, so it isn't certain that this will improve our algorithm, but tests show that, in this case, it does.

.slide 34

At this point you might be thinking that this is an extremely game-specific adjustment that you can't apply to most games. However, this is not true, it just requires a bit more thought. For example, some researchers have applied this variant to the game Go by coming up with three categories for separating the moves. Their version managed to beat the basic version of MCTS in 836 games out of a hundred. So, as long as you can come up with some relevant categories for dividing the available moves, you can use this adjustment.

.slide 35

Alright, we've successfully cleared the selection adjustments and we are moving on the simulation adjustments. Starting with-

.slide 36

Decisive moves.

.slide 37

This is a pretty straightforward adjustment that can actually also be applied to the selection step, but I consider it to be more of a simulation adjustment, which is why I put it here.

So, decisive moves are moves which immediately end the game. For example, if you are playing chess and have an opportunity to checkmate your opponent, that's a decisive move. And this adjustment simply says that you first check if there is such a move, and if there is, you perform it. And you can also take this a step further by taking so-called anti-decisive moves into account – these are moves that prevent the opponent from making a decisive move.

Ok, moving on, we have-

.slide 38

Last Good Reply.

.slide 39

This is another really simple adjustment. When performing simulations, you consider every move to be a response to the previous move. If the simulation ends in a win, you cache the responses you've made to the opponents moves. Then, if you encounter the same move in another simulation, you use the cached move, instead of a random one.

You can of course cache more replies this way, in case the first one isn't legal anymore for some reason. And you can also implement forgetting, where, if a simulation ends in a loss, all the responses used in that simulation that were cached are forgotten. Pretty straightforward.

Next up, we have-

.slide 40

MAST, which stands for-

.slide 41

Move Average Sampling Technique. In this version of the algorithm, you keep a table with average reward for every move. You then use these rewards to bias the selection of moves in a simulation using something called the Gibbs distribution, which you can see here. It may look a bit complex because of all the symbols it uses, but it's really quite simple.

.slide 42

It simply says that the way you compute the probability of choosing an action is-

.slide 43

you first compute e to the power of the average reward of a move divided by-

.slide 44

a tuning parameter tau – you do this for every move, and then the probability for move *a* is that value for move *a*, divided by the sum of those values for all moves. And this gives us a simple way of performing smarter moves in simulations without much computational overhead.

.slide 45

Ok, we've made it through the Simulation adjustments, next up, we have evaluation adjustments.

.slide 46

Starting off, we have Relative Bonus MCTS.

.slide 47

The idea behind this one is that the shorter the simulation is, the more information it gives us. So, in order to take this into account, we compute a bonus that we add to the reward during evaluation. This bonus is used as a control variate-

.slide 48

which is the name for a variance reduction technique which takes advantage of correlation between two random variables, provided that one of them has a known mean, in order to create an unbiased estimator with a reduced variance.

So, if we take a look at the equation on the slide,-

.slide 49

X is the original estimator of some value – so, in our case, that would be the original reward –

.slide 50

Y is a correlated variable with a known mean – that is the length of the playout, although we can't actually know the mean length of a playout in advance, so an online-computed estimate is used –

.slide 51

and Z is the new estimator of E[X]. This is the case because, if we apply the expected value operator to this, then the Y here will become E[Y], which means that this will be 0.

.slide 52

And it can be shown that, if X and Y are correlated, setting the alpha parameter to $-\text{Cov}(X, Y)/\text{Var}(Y)$ minimizes the variance of Z. And this is in our interest because that means we should be able to get a tighter bound on how good the different moves are and therefore our algorithm should be more efficient at identifying good moves.

Alright, moving on, we have-

.slide 53

.slide 54

FAP MCTS.

.slide 55

FAP here stands for Feedback Adjustment Policy, though I suppose that doesn't tell you much. The basic idea behind this one is that simulations that are performed later give us more valuable info because they take advantage of the information present from the previous iterations of the algorithm. To take this into account, this adjustment divides the simulations into groups. Each group is then assigned a multiplicative factor *n* which makes the playouts in the group worth *n* normal playouts. The later the playouts are performed, the larger the factor.

What is interesting about this adjustment is that it seems to handle large numbers of units better than other adjustments that I have tried, so it might be more suitable for RTS games and the like.

It does have one major disadvantage, namely that it needs to know the number of playouts beforehand because it uses that to figure out how many playouts should go into each group, but I suppose it shouldn't be too problematic to alter that.

After this, we have-

.slide 56

WP MCTS.

.slide 57

The WP here stands for weighted propagation and the idea is that, provided that one has a heuristic for judging the quality of a game state, they should be able to get more information out of a single playout than just who wins. Specifically, it tries to take into account the whole trajectory by computing the heuristic value for every encountered state and computing their weighted sum and using that as the result of evaluation.

.slide 58

Alright, we've successfully made it to the last category on today's list – adjustments that don't fall into any of the previous categories.

.slide 59

To start off, we have MCTS_HP.

.slide 60

This is primarily applicable to games where the player fights using multiple units, so mostly RTS games, but I decided to include it because I find it quite interesting. It changes two parts of the original algorithm – evaluation and backpropagation. Evaluation is changed so that, instead of returning 1 or 0 based on who won or lost, it returns the difference between the sum of remaining hit points of the two teams. And backpropagation is changed so that the backpropagated score is always normalized by the sum of hit points of the team that won in the playout. This is to reflect the fact that battles in RTS games aren't just about winning, since you then get to reuse your remaining units in the next battle. So winning with 1 unit remaining and with half your army remaining is a big difference.

After this we have-

.slide 61

All Moves As First, or AMAF for short. Ok, this is a good one.

.slide 62

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 63

Here we have an example. Let's say that the simulation start with move C2, then A1, then performs 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.

.slide 64

Moving on, we have the MCTS Solver.

.slide 65

At the beginning of this lecture, I mentioned that one of the weak points of MCTS is that it has trouble identifying traps. This adjustment is meant to improve that. It does this by trying to prove which nodes are sure wins and which are sure losses. To that end, it backpropagates the values positive infinity and negative infinity, as well as regular ones and zeros. A terminal node – meaning, a node which corresponds to a final state of a game – always gets assigned a value of positive or negative infinity based on whether the given player won or lost in that game. Non-terminal nodes get assigned a value of positive infinity, if at least one of their children leads to a sure win and negative infinity if all of their children lead to sure losses.

.slide 66

Now, this might seem fine at first glance, but it does beg the question – how should we treat the minus infinity nodes during simulations? The positive infinity nodes are fine, whenever we encounter them, we perform the necessary move to get to them because they are sure wins. But what about the negative infinity nodes? The problem is that,-

.slide 67

if we take them into account in the example shown here, then we will conclude that node A is worse than node I, which is clearly not the case, since, in node A, we have a better move at our disposal than any of the moves available in node I. But if we ignore them-

.slide 68

we will conclude that A is better than E, which is also not true.

.slide 69

This is an open question, I don't have one correct answer for you. The authors of the MCTS Solver perform selection using their default policy for the first N times a node is visited, then start using UCB and disregard the nodes which are proven losses.

.slide 70

This points to a broader question with respect to MCTS though-

.slide 71

How can we take into account the number of available moves?

.slide 72

Because, there are games where a low number of available moves is indicative of a bad position, like chess, for example. How can we take this into account? This is again an open question.

Moving on, the last adjustment that we'll discuss today is-

.slide 73

Score bounded MCTS.

.slide 74

This is a further refinement of the MCTS Solver that we just discussed. The idea is that some games can end in a more nuanced result than just win/lose. In such games, it is possible to backpropagate optimistic and pessimistic scores for moves alongside their rewards, which bound real score from both sides.

.slide 75

.slide 76

.slide 77

This allows us to make alpha-beta-like cuts where, if a node's upper bound is lower than the lower bound of some other node, it can be safely pruned.

And that, I believe-

.slide 78

it it.