.slide 1

Hello everyone, welcome to another lecture on AI for computer games. Before I get into today's topic, let me tell you about your new short assignment.

(switch presentations)

.slide 1

.slide 2

This time it's implementing the first step of the MCTS algorithm – selection. Think about how you should traverse your search tree, when you should finish, which nodes you should select, etc. Again, make sure your solution compiles and then send it to me.

I should also mention the group assignment – today is the deadline for handing in your group project specifications, after which, you should start working on your project. You may remember that I said that there will be one round of consultations before the end of the semester which will also serve as replacements for labs that would normally be on 5.12. and 19.12. We've agreed that we'll hold those on 15.12. and 16.12., so Monday and Tuesday of the last week before winter break. We'll soon make available to you some way of signing up for these consultations It is highly preferable to have your whole team there, so, if you find you can't make it on those days, let us know, we'll try to give you a different date. Are there any questions about this?

One more thing that I want to mention, just as a quick side-note, because I stumbled upon it in my notes from last year – there exists something called the Summer School on Artificial Intelligence and Games, which is a multiday event organised every summer for people interested in learning more about game AI. Next year it's going to take place in the city of Leiden, in the Netherlands. So, if something like that seems interesting to you, feel free to give it a look.

Alright, with that out of the way, we can go back to talking about RTS games.

(switch back)

.slide 2

RTS, for those of you who don't know, stands for real-time strategy. In general, this can refer to any game that is played in real time and requires the players to strategize. However, these games have a sort of template – you build buildings and units and research tech upgrades, all in an effort to build an army with which you can destroy the enemy's army. Some classic games of this kind include Blizzard's Warcraft and Starcraft games, Age of Empires or the Command & Conquer games. These are the sorts of games that we're going to be talking about today.

.slide 3

.slide 4

As we're about to see, these games pose a complex and difficult problem from the point of view of AI, so, if any of you would ever like to try your skill at tackling something like this, there are various Starcraft competitions that you can try attending.

And then there's also Screeps, an online sandbox where you can implement your bot and just let it run for however long you like and compete with other players. The downside of this one, however, is that it only supports JavaScript for some reason.

.slide 5

To start with, I have an example here, so that everyone has some idea of what a game like this looks like. This is from Warcraft III. You have a map in this corner down here which shows a representation of the current game map. It's probably not visible so well, but you can actually only see what is going on in this corner, which is where your units and buildings currently are. This is the so-called fog of war – you can always only tell what is going on around entities that you control. In fact, it can be even worse than what you see here – this is a custom game, for which the player has to select a map, therefore they know the layout of the land as well as where important structures and potential starting positions are. In the campaign, you start with the map entirely black, so you don't even know that.

There are two important entities in games like these – buildings and units. Buildings mostly serve to create units or research tech upgrades and some can also work as defensive structures – these scout towers, in this example. Units have various functions. You usually start with some number of worker units – those are the ones you see here. These are capable of creating buildings. Other units mostly have various offensive capabilities and are used to make up your army – so, melee units, range units, flying units, etc.

To create either of these, or to research tech upgrades that can improve your buildings or units in various ways, you need resources, which, in this case, are gold and lumber – gathering these is another task of the worker units.

As I said, these games are a simulation of warfare, so the goal is to create an army and destroy your opponent. If I remember correctly, the winning condition here is destroying their base – which is this building you can see here.

.slide 6

So, what makes games like these difficult in the first place? Here is a – probably incomplete – list of problems that one has to address when creating AI for an RTS title.

First, there's decision-making under uncertainty. As most of the map is covered in the so-called 'fog-of-war' – meaning you can't see what the other player is doing, there's a lot of hidden information that you have to account for somehow.

Then there's the problem of constructing a good opponent model in such a complex game. And, ideally, you'd also like to be able to learn from your opponent in order to be able to beat them in the future.

Then you have to be able to do spatial reasoning, meaning reasoning over terrain – which parts of the map are more advantageous, which less – and temporal reasoning – so, things like deciding when the best time to attack is.

You have to figure out how to manage the AIs resources, since they can be used on various things – buildings, units and upgrades at least.

If your game allows for more than two players, you have to make the AI capable of cooperating with another AI or with the player.

And all of this, as the name suggests, has to be done in real time, meaning you usually have a very small amount of time to make a decision.

So, how do commercial RTS AI usually tackle all these complex problems?

.slide 7

They don't, really. Most of them use FSMs, Hierarchical FSMs or Behaviour Trees, so it's all designer-defined, no special planning or reasoning. These are simple techniques, but that does not necessarily mean that their implementation is simple.

.slide 8

Here is an example of a behaviour tree schema for an RTS game. It probably isn't the simplest diagram you've ever seen. Plus, I would expect that most of the red nodes have to contain some nontrivial logic, so this definitely isn't a walk in the park to code and maintain, if you want to do it well.

.slide 9

And, what's more, these usually don't provide a challenge to even moderately skilled players, because pre-scripted behaviours can usually be easily exploited. So, we are going to take a look at some more complex approaches and algorithms today.

.slide 10

Here you can see some examples of AI architectures used by notable bots that took part in some Starcraft AI competitions. These all break down the problem of playing the whole game into smaller parts. It is worth noting, however, that all of their architectures look very different, so there isn't even a consensus on how best to break the problem apart.

.slide 11

So, after drawing inspiration from some specific articles on this, here are the parts that I decided to break the problem down into. We are going to go over each of them and I'm going to mention some ways of approaching the problem that could be better than just fully scripted behaviours. I based the following slides on both info that I could get about actual games and research papers, so it is possible that some of these sound nice in theory, but aren't actually practical.

So, let's jump into the first topic-

.slide 12

Terrain analysis.

.slide 13

How can we represent the terrain in such a way as to be able to reason over it efficiently? Well, a good first step is to divide it into regions, as this is similar to the what we humans would do with a map like the one you can see here. This can be done either by hand or using an algorithm.

.slide 14

What can we actually use this representation for? Well, it can help us with things like recognizing which areas are dangerous, doing pathfinding more efficiently and overall help with tactical reasoning.

.slide 15

Some specific things we can look for are so-called cul-de-sacs, which are dead ends, and chokepoints, which are narrow places that form the only connections between some regions.

.slide 16

Cul-de-sacs are easy to find – they are regions that are only connected to one other region. With chokepoints, it's a bit more difficult.

.slide 17

You can start out by filtering out cul-de-sacs and regions that aren't narrow. Then, you can make a list of all the neighbouring regions for a given candidate, take each pair of those neighbours and start a BFS search with the aim of getting from one to the other without passing through the candidate region. This can be somewhat expensive, but unless the map can change during gameplay, you only have to do it once. Or you can even precompute and cache this data during production. And you can perhaps optimize the algorithm by cutting it off at a given depth. This wouldn't necessarily mean that the path you found was the only possible path, but it would mean that there is no comparatively short path between the two considered regions.

Ok, so, let's say that we've done all that, we have our cul-de-sacs and our chokepoints – what to do with them?

.slide 18

Well, cul-de-sacs are easy to guard, so they are good places for important buildings. Chokepoints are good for ambushes and they can also be efficiently guarded – if the only path to your territory leads through a chokepoint, it's easy to defend it.

.slide 19

Now that we have separated our map into regions, we can combine it with two data structures that can help guide our units through the map. They are called influence maps and potential fields.

.slide 20

In influence maps, each unit is assigned some influence, which is some measure of how dangerous that unit is. This influence propagates outwards from the unit's position. You can see an example from the game of Go – that was the best that I could find – on the right-hand side. But these can also easily be applied to regions, as you can see in the other image. Influence of your units and you allies is additive and the influence of your enemies is subtractive.

.slide 21

These data structures can be useful when deciding where to defend or attack, as you can easily lead your units to the enemy's weak points.

This can lead to some emergent behaviours. For example, in the game Kohan II: Kings of War, using this technique resulted in the AI attacking the player's weak spots, flanking manoeuvres and even seemingly luring the player away only to then attack them from the rear – none of which had been explicitly coded or designed.

And the authors also used this data structure to encourage cooperation between AIs that were supposed to be allies by combining their influence values and this supposedly achieved pretty good cooperation out of the box.

.slide 22

The other tool I mentioned, a potential field, is a data structure for navigation. I believe the idea is inspired by physics – you have a 2D array of attractive and repulsive forces that tell your units where they should move. These can easily be adjusted dynamically so they can be used to repel or attract your units to the enemy or anything else.

One unintuitive thing that these can also be used for is getting your units into formation. For example, by just adding some repulsive force between your own units, it will cause them to move with large spaces between them. On the other hand, if you make them attract one another, they will become all bunched up.

Alright, now that we have some way of abstracting the terrain and analysing it, we can move on to-

.slide 23

Unit tactics.

.slide 24

So, influence maps can be used to tell your units where to go and potential fields to tell them how to get there. But what about when they actually get into a fight?

.slide 25

As always, the simplest solution is just giving them scripted behaviours. And here, this can actually lead to some pretty ok results, especially if the behaviours are modelled on actual human player behaviours.

.slide 26

Here you have some examples of possible scripted behaviours – as you can see, it's pretty straight-forward stuff, like attacking the closest enemy or the weakest one. Kiting is one behaviour I want to point out, because this is most evidently inspired by human behaviour – it means that you have your unit attack another unit, but then immediately pull it back, so that it isn't retaliated against.

.slide 27

Ok, so, scripts are alright, but just having one scripted behaviour that a unit will carry out every time it is in a given situation will probably become predictable after a little while. To reduce predictability, you can

maybe try putting multiple scripted behaviours together into an FSM, or you can try using some more advanced script assignment or planning algorithms – so let's take a look at those.

.slide 28

Let's start with Portfolio Greedy Search or PGS for short. This algorithm is given a set of scripts – called a portfolio – and tries to find a good assignment of these scripts to its units.

.slide 29

Here we have a sketch of the algorithm. It starts out by assigning some default script to all the enemy units. Then, it searches for the best script to assign to *its units* – it determines which scripts are and aren't good using playouts, although just one per script, so this is a very coarse estimate. Then, it tries to find the best response from the enemy to *your* assignment. This is the initial setup. And then, as long as it is within budget – usually meaning until the time to make a decision runs out – it goes through all of its units one by one and tries to find a better script assignment for each of them separately – what is good is again judged using playouts. Then, once it finishes with its units, it starts going over the enemy units using the same process and then just repeats.

.slide 30

Here we have the pseudocode for the algorithm, but we don't need to go into that here, it's just there in case the short explanation wasn't clear.

.slides 31-33

.slide 34

This algorithm has a weakness from a theoretical point of view in that it isn't guaranteed to converge to a Nash Equilibrium, meaning it is not guaranteed to find the optimal strategy. In the example shown here, it will keep switching between assigning *a* and *b* to player 1 and *e* and *f* to player 2 and never stumble upon the optimal assignment of *c* for player 1 and *e* or *f* for player 2.

However, from a more practical point of view, I'd say the weakness of this algorithm is that it isn't even guaranteed to produce a very *good* assignment of actions, let alone the theoretically optimal one. This can happen because it simply isn't equipped with a good-enough portfolio of scripts, or because a single playout isn't enough to judge the quality of a script assignment reliably, or because of its greedy nature.

Still, it may produce results that are good-enough for our purposes.

If we'd like to improve it though, one obvious way of doing that would be to nest it. Which gives us-

.slide 35

Nested Greedy Search, or NGS for short.

.slide 36

This algorithm basically just adds one level of recursion to the process used in PGS. It starts out similar, though not the same. First, it assigns some default script to the enemy units, then to its own units – this is different from PGS, it doesn't jump straight into looking for a good response. After that, it goes to the

main loop where it goes through all the units and for each tries to find a better script. But it does this using one level of recursion, so, for every action that it considers, it first nests itself and tries to find the best response from the enemy and only if it doesn't find a good response from them does it assign this script.

We could keep doing this, of course – adding more and more levels of recursion to get better results. The problem is that we'd be facing a combinatorial explosion – to me, even this sounds sound like something way too computationally expensive to be actually usable in most RTS games.

.slide 37

Again, some pseudocode, we can skip that.

.slide 38-40

.slide 41

Ok, so, these were some greedy algorithms that, while possibly useful, were in no way *guaranteed* to find a good assignment of actions. So how about trying some proper planning algorithms instead, ones based on searching game trees for example? To do that, we first need to address how to even apply them to a setting like this, one which a) is real-time and b) allows for simultaneous moves.

We discussed possible approaches to addressing this in MCTS last time, however, here, we are going to solve it a bit differently.

.slide 42

The first planning algorithm that we're going to look at here is an adaptation of Alpha Beta Search called Alpha Beta Considering Durations, or ABCD for short – a very pleasing acronym. It handles the fact that actions have durations by simply performing them and then rolling the game state forward until at least one player can do something again.

As for simultaneous actions, it separates nodes where both players can make moves into two – a FIRST node and a SECOND node. A move for one player is picked in the FIRST node, but it isn't applied until the opponent's move is also chosen and applied in the SECOND node.

I need to emphasize here that-

.slide 43

This is not a theoretically sound model for simultaneous move games! The player who gets to pick second has the advantage here of basically knowing the first player's moves, even if they aren't applied. It's the same problem as we discussed with rock-paper-scissors.

.slide 44

There can be different strategies applied to picking which player goes first to try and mitigate the advantage – they can alternate, be chosen at random, or alternate such that first one player goes first, then the other, then the other again, then the first – so-called 1-2-2-1 alternation.

.slide 45

As you know, Alpha Beta Search requires a way of evaluating game states which, unless we can construct the entire game tree, has to be done heuristically. So, what possible heuristics can we use here? There are several possbilities. The first formula just takes into account the hit points and damage per frame of each unit.

The second one applies a square root function to the hps of the units. This is done to differentiate between the cases when you have, say one unit with 100 hit points or two units with 50 hit points. The latter is seen as better and applying square roots to them, takes that into account, because, in the first case, the square root of 100 gives you 10, whereas 2 times the square root of 50 is something around 14.14.

Then, of course, you can use playouts, like in MCTS and PGS and NGS which we just discussed. Or, there are other possibilities. I've seen a paper where someone proposed using Lanchester's Attrition Laws, which are formulas developed for estimating the strength of armies and the outcomes of their encounters in actual real-world wars, to estimate who has the advantage in RTS games, so those could be applicable here as well. And, instead of just using these statically to evaluate states, you could actually also use those instead of playouts to estimate the outcomes of battles.

.slide 46, 47

.slide 48

We can also take move ordering into account. Since we are under sever time constraints, we have to run this algorithm in iterative deepening mode, meaning that we first construct the tree up to a given depth, then increase that depth if we have more time in available. And, since Alpha Beta Search prunes parts of the tree that are provably suboptimal, so, in every iteration, we can store information about which moves we consider good and pick those moves first in the next iteration. If this is done right, it can greatly increase the speed of our algorithm.

.slide 49

So that was an adjustment of Alpha Beta Search, but, of course, I couldn't close this section without talking about our old friend, Monte Carlo Tree Search. An application of this algorithm to the problem of assigning actions to units is called UCT Considering Durations, or UCTCD for short. The base algorithm is adjusted using the same techniques we saw in the Alpha Beta example, so rolling the game state forward until at least one of the players can move and splitting simultaneous move nodes in two.

.slide 50

Now, the problem with applying these algorithms to a setting like this is that they are probably way too computationally costly to be usable. Generating all the possible actions takes time, as does the act of simulating the game, even if that is only done in some simplified way, and the branching factor is most likely very high. This is probably the reason why PGS and NGS outperformed UCTCD in experiments.

.slide 51

We can try to combat the last problem using two ways, one is ditching basic unit actions and instead searching over script assignments – this seems to give UCTCD the edge over PGS and NGS again.

The other is putting the units into groups and assigning each group one script or action. This again makes sense because that's what humans do when playing these games – they issue commands to multiple units at the same time.

.slide 52

This grouping can be done based on various criteria, such as proximity, unit type, or unit characteristics, like their attack range, distance from the enemy and current hp. So, for example, you can group together all the units that have a low hp and have them retreat, or group together all the units that can attack from a certain range and command them to do so together.

But, to be honest, even with all that, I'm not too sure about the viability of using these approaches in commercial products.

.slide 53

One thing that has been used in commercial products though, to tackle the problem of how units should fight, is machine learning – specifically neural networks.

These have been used at least in two titles – Supreme Commander 2 and Age of Empires IV. In Supreme Commander, specifically, they were used as part of an FSM – if the machine transitioned to a specific state, the unit commands would be issued using a neural network.

There were different neural networks constructed for different types of units, such as land units, naval units, etc. They had a bunch of inputs, like the number of units, unit health, damage per second and so on, and their outputs were the utilities of various actions, such as attack weakest, attack closest, stuff like that. Run away wasn't among the actions, instead, if all the resulting utilities were under 0.5, the units would retreat.

.slide 54

The obvious downside to using these is the fact that they may be difficult to design properly and the results difficult to fine-tune. However, the behaviours you get out of using them can actually be adjusted. For example, in Supreme Commander, the authors tried changing the inputs they gave to the neural network to produce different behaviour. Specifically, they found that if they added some positive bonus, it would make the AI more confident and therefore more aggressive.

Another possible issue though is the fact that these networks can produce behaviours that are hard for humans to understand. I've heard that Creative Assembly, the company behind Total War, has had someone do research on using neural networks to command whole armies and the AI could then play pretty well, but its behaviour was just so strange that they found they couldn't explain what it was doing to players, so they scrapped it.

Alright, that's it for unit tactics, but of course, fighting isn't the only use of units in RTS games. Another important thing is-

.slide 55

Scouting.

.slide 56

This entails answering two questions – how to gather information and how to use it once we have it.

.slide 57

Let's start with the first one, how to gather information?

We start by creating one or more scouting units and sending them out to explore the map. In some games, like Starcraft, there are some predefined possible starting positions for players, so the unit can just explore them one by one. Or, if a more thorough map search is required, it is possible to generate candidate positions based on utility measures such as distance, new area that would become visible from that point – because, in some games, you can have these vantage points from which more of the map is visible – etc.

.slide 58

Ideally, the unit should encounter the enemy sooner or later. Once that happens, we can use potential fields to both attract the scout to the enemy to get a better look at what they're up to and to repel it in order to avoid being caught.

.slide 59

Ok, let's say we've found the enemy and uncovered their evil plans. Now what?

There has been some research done on using Bayesian inference, which is a technique from the area of probability theory, to estimate the enemy army's strength based on their observed units. However, this is pretty complicated stuff, which is why I decided to skip discussing it in detail – just know that it exists and you can take a look at it if you feel so inclined.

There are simpler approaches to this however. Specifically, you can try to estimate the opponent's strategy, and therefore also their unit composition, just from their buildings, using hardcoded rules.

.slide 60

This was used in the Starcraft bot ICEBot, which scored very well in some Starcraft competitions and it supposedly outperformed Bayesian approaches.

.slide 61

This can then be used to inform your overall strategy and unit composition, as well as to decide when to attack the enemy. For example, ICEBot launches a large attack when the size of its army is at least *c* times the estimate enemy army size, where *c* is a tunable parameter.

Ok, so, that's scouting. Now I want to move away from discussing units and instead move on to buildings. Specifically-

.slide 62

Where to place them.

.slide 63

So, this problem has some straight-forward constraints. Buildings shouldn't be placed too far from the base building, because the larger your settlement, the harder it is to defend. They shouldn't block access to resources, shouldn't block unit movement and generally not be in the way. They should be placed in locations in which they are easy to defend. And, of course, the decision process has to be fast.

.slide 64

Again, the simplest approach is to just predefine some building spaces, but this would probably be quite time consuming, as it would have to be done by hand for every map – unless you constructed some algorithm to automate it.

For an example of an approach that's a little more complex, I'll tell you what the designers did for Starcraft. First, a boundary of the settlement is defined – I'm not sure if this is predetermined or done somehow algorithmically, but I would guess the latter. Then, spaces are defined where buildings shouldn't be placed – around the edge of the settlement, around resources and around other buildings, as well as on some straight paths that allow for efficient unit movement. This leaves a bunch of usable places. I don't know how the AI then picks from among these – I would assume it's just done based on proximity to the base building.

.slide 65

Here you can see a visualization of what that's like. So, the hashtags are the boundary, the S cells are the base building, M squares are minerals and G is probably gold, I think, or some other resource. First, you shouldn't block the path between the base and your resources, so the red squares are off limits.

.slide 66

Then, you shouldn't place them right next to the boundary either, I assume it's because they would be difficult to defend there and also difficult for units to get around.

.slide 67

Then, some additional straight paths are defined.

.slide 68

And to top it off, additional banned spaces are defined around resources and existing buildings.

.slide 69

And what's left are viable spaces for placing buildings.

Alright, now that we are capable of placing buildings, the question is, which buildings can we even afford to build? And that brings us to-

.slide 70

Resource management.

.slide 71

We need resources to do various things – build buildings, create units, research tech upgrades and possibly more, depending on the specific game. So, how do we allocate resources to these different tasks?

.slide 72

This is a problem that most often just isn't addressed very thoroughly – or at least I assume that based on the fact that I found very few articles, research or otherwise, about this topic. There hasn't been any widescale survey to confirm this though, as far as I know.

The simplest techniques to use here are first come first served, or having some predefined partitioning. It is theoretically possible to do some planning based on predefined priorities, but I didn't find any material on how to do that.

Ok, so with that, we've gone over the most important building blocks of RTS games – from terrain to units to buildings. Now the question is, how to put it all together?

.slide 73

What is our overall strategy?

.slide 74

We have a lot of decisions to make at the highest level, some of which we've touched on already – which buildings to build, which units to create, which tech upgrades to research, how to make use of scouting data, where to send our units, when to attack the enemy.

.slide 75

At the beginning, I showed you this slide with the architectures of different prominent Starcraft bots.

.slide 76

All of these are somehow scripted at the highest level. They are either just hardcoded or choose from some predefined strategies. So, that's just to say that, even in competitions, nothing more complex than that is usually used. Or, I mean, these results are already some years old, so maybe that's changed, I can't be sure.

.slide 77

So, the easiest – and probably most practical – thing to do here, is using designer defined strategies. These can actually yield pretty good results, if, like the scripts we discussed in the unit tactics section, they are based on strategies used by expert human players – and provided that they are supported by capable systems at the lower levels of course.

A notable example of this is Halo Wars 2 – one of the designers of this game was a pro RTS player and he helped create good strategies at both the high level and the lower levels which were based on how humans play.

Alright, but let's say that you still want to try to do something more complex, what are the possible approaches you could use?

.slide 78

To be clear, all of these are just speculation, I haven't seen most of these used to govern the high-level decision-making in RTS games.

Goal-oriented action planning is a reasonable candidate.

.slide 79

I won't go into the details as this technique was already discussed during the lecture on F.E.A.R. AI.

.slide 80

You can use some tree search algorithms, which we've also talked about during this course. If I tried using something like MCTS here, I would probably ditch modelling the opponent and just focus on my actions while modelling the opponent's behaviour as random environmental effects. Also, it is possible to make the tree search hierarchical – you first search at the highest level of abstraction, choose some actions, then you progressively refine your plan – so maybe that could be useful in this context.

.slide 81

You can use constraint satisfaction and optimization algorithms as well as scheduling algorithms. These are useful for things like deciding build orders or which specific units to build in order to counter the enemy. These weren't covered during this course, but if you want to know more, there are courses on planning and scheduling and on constraint planning taught by Roman Barták. These go into considerable depth on the topic of course, more than would be necessary for our purposes.

.slide 82

Machine learning – I haven't seen anyone besides Google successfully use machine learning for high-level strategizing, but I suppose it could be useful. A more straightforward use case, however, would be using it in conjunction with some other approaches. For example, I've seen a research paper where the authors tried to come up with a neural network for judging the quality of the current game state in Starcraft, I believe, so that could be used in conjunction with tree search algorithms.

.slide 83

And, last but not least, we have Hierarchical Task Networks, or HTNs for short. These were also talked about in the lecture on F.E.A.R. AI, but only briefly, if I'm not mistaken, so let me quickly reiterate what they are about.

.slide 84

They are based on a designer-defined hierarchy of tasks. These can be either primitive or compound – primitive tasks can be translated straight into actions, so they are things like 'go here' or 'attack here'. Compound tasks need to be further decomposed. So, as a really simple example, you could have a compound task such as 'destroy the enemy's base' and then you could decompose that into 'navigate to the enemy's base' and 'attack enemy' and then you would continue decomposing until you had arrived at primitive tasks issuable to every unit.

.slide 85

This decomposition can be done in different ways. So, you can imagine a compound task being some broad objective, like 'attack the enemy' – there are various ways you can attack, so that can be decomposed in different ways. These ways also have to be predefined.

Of course, whether a given decomposition is actually applicable depends on the given state of the game, which is why each task has preconditions that determine whether it is applicable to a given state. The primitive tasks also have effects. The task decomposition is carried out until a plan consisting only of primitive actions is created. And the planning process of course involves backtracking – if you find that some decomposition you chose doesn't work, you undo all of its effects and choose a different one.

Now, you may notice that these HTNs aren't really suited for modelling adversaries, they only model *your* actions. If you wanted to change that, you can use-

.slide 86

Adversarial Hierarchical Task Networks, or AHTNs.

This is basically a combination of HTN and minimax. Each player has a task hierarchy and you start by decomposing tasks for one of the players until a primitive action is reached on the left-most side – this will be the action taken by that player. The reason it has to be the left-most side is that that's the order in which HTNs are traversed – you start from the left and progress through all the leaves, which are primitive actions. So, one level in the minimax tree corresponds to different decompositions of a task, until a primitive action is found. Then you move to the next level, where you decompose tasks for the opponent. This is once again done until a primitive action is encountered on the left-most side. Then, you switch back to decomposing your tasks, etc.

A key difference from standard HTNs is that you don't need to do any backtracking here, because the different tree nodes represent the different decompositions.

.slide 87

You can observe, however, that this algorithm necessitates that the game be turn-based and perfect information. In order to adjust it to work with durative actions and simultaneous moves, we can use the same techniques that we applied to ABCD and UCTCD – namely rolling the game state forward until at least one of the players can make a move and splitting simultaneous moves into two nodes. As for imperfect information, I know of no research on the topic – I assume it should be possible to handle this by combining AHTNs with some inference techniques like what I talked about with respect to scouting. There is one more solution to that, though, which ties into the next topic.

So, thus far, we've covered all the subtopics that I laid out at the beginning. But. There is one thing that I've been avoiding talking about this whole time. And that is-

.slide 88

Cheating.

We are the ones designing the game, so, of course, we can allow the AI to cheat. In a game as difficult as this, it may even be desirable to do so in order to give the AI at least a fighting chance against a somewhat skilled human player. The question is, when and how?

.slide 89

Here are the topics we've covered thus far. Out of these, I'd say you can cheat in-

.slide 90

These four. Scouting is sort of obvious – the AI can be allowed access to all information on the map, so it doesn't need to do any scouting at all. This ties into high-level strategy, because, if you know exactly what the other player is doing, you can try to counter them, for example by building units that are strong against their units.

With resource management, you can just give the AI more resources.

And as for unit tactics – you can even allow the AI to create more units out of nowhere.

The question is always – will this improve the player experience? That depends, first of all, on how obvious it is. For example, I heard that, in the first Warcraft game, the AI would spawn enemies just outside your field of vision and keep doing so until your army was almost depleted. You would then feel like you've just won a hard-fought battle, which is a good feeling. Unless, of course, you somehow found out that was what the AI was doing, in which case, you'd most likely just be terribly annoyed.

On a similar note, I remember when I was playing Warcraft III as a kid, how I was really impressed by how fast the AI could get a decent army going – as we saw in last week's lab – and how I thought 'wow, I wish I could play that well'. And now, knowing that it just has a resource advantage definitely takes that feeling away.

Another example of this is scouting – as I said, the AI doesn't really need to scout, it can just have access to the entire map. However, if it then immediately knows where you are, then that's kind of obvious. So, in some games, the designers have the AI send some scouting unit your way to pretend like the AI actually scouted you and only then start attacking – that makes it somewhat more believable.

And another thing, this time with respect to unit tactics, is actions per minute, or APM. It is somewhat questionable whether to consider this cheating, because the computer can simply pay attention to the entire map and control units faster – that's just an inherent property of the machine. To some, this might be ok, to others, it might not.

If you have a dilemma about this, then one thing you can do is actually expose the AIs parameters to the player.

.slide 91

This is a screen from Supreme Commander 2. As you can see, the player can set the values for various advantages that the AI can have. Doing this can make them feel like they are more in control and therefore can, I would assume, improve their experience, even if the AI is cheating.

So, that's what I wanted to cover with respect to RTS AI in general. To close this topic, I would like to take a look at a single real-world example.

.slide 92

The Total War series. This is a well-known series of real-time strategy games that are even more complex than the likes of Starcraft. And, since it's a long-running series – the first entry came out back in the year 2000 – it's AI has evolved over time. So, we are going to look at some of its underlying AI architecture as well as how the designers' approaches evolved over time. This will not be some in-depth analysis – there isn't enough info out there for that – we'll mostly just go over some basic ideas and approaches that the designers used.

.slide 93

First, some overview of what these games are like. Every Total War game has two modes. You start in the campaign mode, where you have a large map with territories, some of which belong to you. You then try to create armies, move them around, conduct diplomacy and do a bunch of other things in order to get control of as much land as possible.

When armies from two enemy factions come to the same territory, that's when the real-time battle ensues. These battles are different from those in classic real-time games however, because the armies of the players are fixed, so there is no building creation and management of resources – all that happens in the first mode.

From this, you can see that this game differs somewhat from the classic template. However, there is no other game of this kind that I know of with this many entries and with enough information about its AI out there for me to use as an example, so this is what we're going to go with.

First, let's take a look at the underlying AI architecture.

.slide 94

This should be what they are using today, though I doubt it's been this way since the first entry.

As you can see, just like the game itself, the AI is split into Campaign AI and Battle AI. These are two completely separate systems.

The campaign AI is further separated into parts that deal with diplomacy – that's another big topic that we haven't touched on at all today, by the way, but I will talk about it a little bit when describing the AI for one of the entries in the series – construction of buildings, management of armies, etc.

.slide 95

These AI systems make decisions in a specific order, which reduces the complexity of the search space and makes the individual components more independent.

.slide 96

The battle AI is further divided into two systems which run in parallel. I'm going to call the one on the left the strategic system, for the lack of a better name.

.slide 97

So, the strategic system decides whether the army should attack or defend and assigns objectives to detachments – these are groups of squads of units. Every squad then gets assigned a Tactic.

.slide 98

These tactics are implemented using FSMs and usually represent tactical manoeuvres such as flanking. All this is meant to get the units near to the enemy in a hopefully smart way. Once the units actually make contact, the Attack Manager takes over.

.slide 99

It temporarily steals the control of a unit away from a tactic and then gives it back once the battle is done.

And that is the gist. So, let's now take a look at how the AI of Total War evolved, starting with the very first entry –

.slide 100

Shogun: Total War, which takes place in 16th century Japan. The behaviours of individual units in this game are – supposedly – controlled by neural networks. I say supposedly because I got this from a video on the AI and Games youtube channel and Tommy Thompson, the author, is usually a pretty reliable source of information on these things – plus, he knows people who work at Creative Assembly – but someone who claimed to be an ex-Creative Assembly employee wrote a comment there, claiming that they never used neural networks for anything. So, I don't know.

The overall Battle AI was inspired by the famous book The Art of War and used rules inspired by it.

The diplomacy AI was based on simple FSMs, but genetic algorithms were used to give each AI some personality.

.slide 101

After Shogun, the overall AI system remained mostly the same for a couple of years, until Empire: Total War in 2009. This game takes place at the beginning of the 18th century, so a major change of setting which brought about some corresponding changes in gameplay, like naval battles and an emphasis on guns.

The AI got a complete overhaul for this game. From this title onward, the campaign AI has been completely separate from the game logic, so no cheating, and both it and the battle AI were implemented using Goal Oriented Action Planning.

The AI was very ambitious and showed some promise, but unfortunately, it fell short of expectations. The authors claimed that it was because, while it could plan well for the long term, it couldn't foresee the conflicts between some of the actions it wanted to take and therefore frequently self-sabotaged, basically.

.slide 102

After this, the next entry to bring some major AI changes was Total War: Rome II, in 2013. This was quite possibly the first game of its kind to integrate MCTS into its AI. Specifically, the authors use it as part of the campaign AI for two purposes. First, to allocate resources to tasks that are generated by various generators – these generate tasks for different aspects of the gameplay, so, there's a generator for

armies and diplomacy and all the other major tasks – and then, MCTS is called a second time to give commands to units. So, it is first used to decide what to do and then how exactly to do it.

The version of MCTS used here was of course heavily optimized, but even with that, the algorithm could reportedly usually only search the first level of the game tree properly and even that took longer than the developers would have liked.

.slide 103

After Rome II came Total War: Atilla, in 2015. The diplomacy AI for this game was supposedly a notable improvement, with more detailed deal evaluation generation and each faction being equipped with a set of data- driven personalities. The reason it's a set instead of just a single personality is that these change over time. They do so in relation to how the game is going for the given faction as well as to reflect changes in the faction's leadership.

The MCTS implementation also received a number of improvements, such as-

.slide 104

Using influence maps to determine the level of threat from enemies in various regions. This saves on path-finding queries – you can use this estimate instead of actually computing whether an army can reach some region. However, it is just an estimate, so it can happen that you mark a region which actually isn't reachable by the enemy as in danger, so, as with everything, there is a trade-off.

Stricter pruning of moves was introduced – that's always a good strategy for dealing with large state spaces, just do your best to heuristically prune as much of it as possible. Namely battles where the enemy was clearly stronger than the AI and moves where the likelihood of success was deemed to be low were all pruned.

The implementation of MCTS used some higher-level state abstraction, which could involve moves that weren't actually doable in the game – something like the example with influence maps I just mentioned. Before committing to a move, it therefore needs to be validated, which, I imagine, isn't computationally cheap. In order to do this as little as possible, the algorithm only does this once a new best sequence of moves is found – and, if it is found to not be legal, the moves are discarded.

Some plans involved actions that have non-deterministic results and only if they come out a certain way can the plan proceed. The designers therefore post-processed all such plans to carry out the non-deterministic actions as soon as possible, to determine if replanning was necessary.

And they also introduced an arbitrary move ordering in order to avoid duplicate action sequences, which are sequences which have the same outcome and consist of the same moves, just in different order.

All of this reportedly made MCTS perform a lot better, but it still usually couldn't search more than one level of the tree.

.slide 105

Atilla was followed by Total War: Warhammer. This entry received some improvements to the specific AI subsystem responsible for siege battles. These are battles where one of the players is walled off in some fortress and the other is trying to break in.

In older entries, there was an emphasis on the historical accuracy of these battles. So, the AI would start by bombarding the walls and, once it established some entry points, it would try to storm the city. This was implemented by FSMs.

In Warhammer, the design was a bit more ambitious, which was partially necessitated by the change in settings – Warhammer is the first Total War game that takes place in a fantasy setting, so you have things like flying units, weird monsters and hero units with special abilities.

.slide 106

The siege AI uses designer-defined lanes to determine points of attack. So, these are just lines that stretch from the position of the AI's army towards the city that they are laying siege to and the points where they intersect with the walls are the points that the AI will attack.

Once these points are determined, their surroundings are scanned to determine which tactic is best for creating an entry point there. There are 3 possible tactics – storming the gates, breaching the walls or climbing them. All attacking units are assigned one of these tactics. If the tactic is successful, an entry point is created. If not, the remaining units are moved to the reserves, which is a new feature of Total War AI. The reserves stay in the back and can then be deployed when necessary.

.slide 107

When the AI successfully creates some entry points, its units try to storm the city. Once inside, they are taken over by a different AI subsystem, specifically intended for this task. This system uses multiple influence maps that contain information such as enemy influence, strategic values, exposure to missile fire, etc., to tell the units where to go.

.slide 108

The last entry I'm going to talk about is 2019's Total War: Three Kingdoms. This game received some notable improvements to its diplomacy AI and I'd like to spend a little time on how this system works, since its something we haven't touched on yet in this lecture.

The designers had three design principles with regards to their deal generating system – the deals had to be reasonable, diverse (the AI shouldn't keep proposing the same deal over and over again) and fast.

.slide 109

Here is their overall system architecture. The deal generation part I don't know much about, but I assume this can be just some combinatorial generator – as in, you have various things you can offer the other side, so you generate all possibilities. These are then filtered, using some heuristics and the resulting deals are further heuristically assigned priorities. That gets passed to the deal evaluation system.

.slide 110

This system has 5 categories of criteria, or consideration groups, that it considers – all of these involve a lot of hard-coded rules and computations.

.slide 111

The stance consideration group, meaning ‘how do I feel about the other faction’? Do I like them or dislike them?

.slide 112

The respect group – how reliable do I find the other faction? How has it behaved towards its allies up until now?

.slide 113

The economic group – how economically sound is the given deal, am I offering a reasonable amount of compensation, or conversely, am I asking for a reasonable amount?

.slide 114

The diplomatic group – how might other factions react to this deal?

.slide 115

And the strategic consideration group – how dangerous is the faction I’m dealing with, how much power do they have?

.slide 116

These considerations were all combined together to create a score for all considered deals and one would be chosen. This ensures reasonableness of the deals and speed is ensured through proper heuristic pruning and prioritizing before the deals are passed to the evaluation system, but what about diversity?

.slide 117

To ensure the AI doesn’t keep offering the same deals again and again, it simply remembers the deals that have been rejected for n rounds and doesn’t offer them again until after.

.slide 118

So, that was the broad outline. One thing that the designers tried to focus on specifically was to create these organically shifting alliances involving intrigue and betrayal. To do this, they created a formula for evaluating the threat level of every faction. With this, each faction can identify its main threat and try to adjust its alliance based on that – that can mean allying itself with other factions that have the same main threat, but also stabbing their current allies in the back, in case their newest main thread is among them.

.slide 119

Lastly, I’d like to show you a bit of how this works with an example. Here we have Cao Cao, a leader of one of the factions in the game. Here you can see the values of some of his personality parameters and their comparison with those of a default AI. First, his main_threat_value is high – this means that, once he identifies a main threat, it becomes his top priority. Then, down here, we have his respect_value_own value – that indicates how much a character cares about how others perceive him. In Cao Cao’s case, it’s zero, so he is willing to do anything irrespective of what others will think of him. And then we have his

diplomatic_stance – this indicates how much a character cares about how others have behaved in the past. So, putting all this together, you get the personality of a paranoid, cunning diplomat, who will go after whomever they consider a threat and won't hesitate to stab their allies in the back.

And, in the talk that I'm basing this on, the creators actually showed a bit of gameplay, where Cao Cao first allied with another faction with whom they shared a main threat, and they quickly dispatched it, but his ally became to powerful in the process, so he immediately turned around and started attacking him, even though their initial enemy wasn't fully destroyed yet.

So, that's how combining data-driven personalities, where you define a myriad of values that the designers can then tweak to achieve different behaviour, with this threat-evaluation system can result in interesting in-game behaviour.

Alright, that's all the info I managed to get about Total War. Note how many different techniques were used here – FSMs, neural networks (supposedly), genetic algorithms, GOAP, MCTS, influence maps and maybe I'm forgetting something. And, of course, pathfinding is also necessary. So, that's basically all that you've been taught during this course in just one game series. As far as game AI goes, this is really as complex as it gets.

There is one thing I should note now in relation to this. I watched a couple of talks about the AI of Total War from the designers when creating this part of the lecture. And in them, they always talk confidently about how interesting and intricate their design is. And then, you look at the comments, and it's all stuff like this:

.slide 131

"Weird, the games with the worst AI ever made, the creator is talking about what they did right with the AI??? Shouldn't it be an analysis of how they've failed to make any AI improvements in over 20 years?"

And just looking through these and through some other videos on the topic, it seems that many Total War players really don't like the AI. I'm not sure whether this is actually a prevalent opinion in the community or just a loud minority, but, more importantly, I'm not sure they know what they are asking. Because, as we've seen, this is a hard problem and the designers are evidently constantly trying to improve, so it might be like being mad at doctors for not having yet developed a cure for cancer. However, I haven't played the games yet, so it may be that it could be done better.

That concludes today's lecture. Tomorrow we are going to have a lab that's going to be quite different from ones I've led up to now, which have all been focused on brainstorming solutions to various game AI problems. Tomorrow, I'm instead going to show you some new game development tools based on generative AI that you can use to craft novel experiences. If you're going to come, please bring your notebooks because there should also be time for you to try these out yourselves.

After that, Adam is going to take over again for two weeks when he'll talk to you about reinforcement learning and then I will close off the semester with a lecture on how machine learning more broadly is being utilized in games these days. See you then.