2024
Sem 1

Game Playing Agent

COMP30024 ARTIFICIAL INTELLIGENCE: PROJECT PART B

WILLIAM SPONGBERG & GAOYONGLE ZHANG

Introduction

Upon learning of the game Tetress, it seemed to our team that the best possible game algorithm had to be Min-Max. It looks at all possible moves, finds the best ones, and executes them. However, after playing the game manually and researching further, we soon realised the immense branching factor that impeded this algorithm, and the restrictions within find the "best" move.

There are 19 pieces in the game. Each of these pieces has 4 coordinates, therefore each piece can be placed down on the board in 4 different ways.

$$Branching\ Factor = 4 \times 19 \times no.adjacent\ coordinates$$

In the worst-case scenario, this branching factor could be in the thousands (assuming most piece placements are valid). While Min-Max in coordination with Alpha-Beta pruning would effectively reduce this branching factor somewhat, it seemed far too computationally expensive for effective game-playing, and it seemed that Min-Max would run out of time well before the end of the game.

Furthermore, the heuristic used to calculate Min-Max could realistically not be optimal and getting it close to optimal seemed an arduous and almost impossible task. Even after all this extra time given to Min-Max, the move may not be the best one. Depth seemed to be the one thing that was within control, and we knew which algorithm took advantage of this best.

Monte-Carlo Tree Search took the large branching factor of this game and turned it into an advantage. By randomly selecting several of the available moves of a state and simulating a random v random game from each move to a terminal state, MCTS can think far ahead and have a general idea of what moves to avoid/prioritise. It can also use current states and compare the possible future states against previous simulations to determine if they were beneficial or not. The more time you give it, the better it becomes, and the more it adapts and learns how to play the game, even giving the future possibility of training it by using machine learning, an exciting task. With MCTS, the need for all branches to be explored, the unbearable task of creating all-knowing heuristics and the need to win the king's game of Tetress appeared to be conquered.

MCTS Implementation

MCTS's general premise is to "learn on the job" – it sends off random paths to terminal states and reports back for every move made whether it was a good or bad move based on the result. It can then use this to influence the moves it makes in the future, especially if a possible move is one it has already simulated for. This allows measuring of the average utility of a state rather than just loosely guessing it based on a heuristic.

MCTS consists of four main methods:

1. **Selection**
   This consists of using the UCB1 formula to select the successor node based on a balance of expansion and exploration using its number of playouts and wins. This means the same nodes won't be played repeatedly, and to explore other states, while still allowing for beneficial nodes to be explored further and their results to be used again in future.
2. **Expansion**
   Add a new child from the selected leaf to the tree. This is randomly picked.
3. **Simulation**

Do a random playout from the new child to a terminal state for x number of times. The number of times this simulation is done varies greatly in our agent, as detailed later. This simulation must be optimised as much as possible to give the agent more time.

4. **Backpropagation**
   Update the child node and each subsequent parent on its number of playouts and number of wins. This is simply represented and executed using a defaultdict.

As each MCTS node is created, all possible actions are calculated by getting all valid_moves for every valid_coords of the current state. Then, an action is chosen based on UCB1. This action has a new MCTS node created for it, which is then played out to a terminal state – with it and its parents' number of playouts and wins updated accordingly, all the way back to the initial root node.

Initial Optimisations

Within MCTS, unfortunately, lay a different and not entirely easier set of issues compared to Min-Max. While MCTS did play better, it required an enormous amount of time to simulate a whole game, e.g., 100 times, per turn. However, this was uncertain – it was simply a number we gave the algorithm because it seemed like a good one.

Our next steps were then to optimise the underlying methods used and identify bottlenecks before we eventually optimised the number of simulations. Thanks to my tutor, we realised the usage of cProfile and snakeviz, which were immensely helpful in finding such functions. It had also been made obvious that the Board class was inefficient, but it was relatively unclear how inefficient and where until it was made obvious by cProfile.

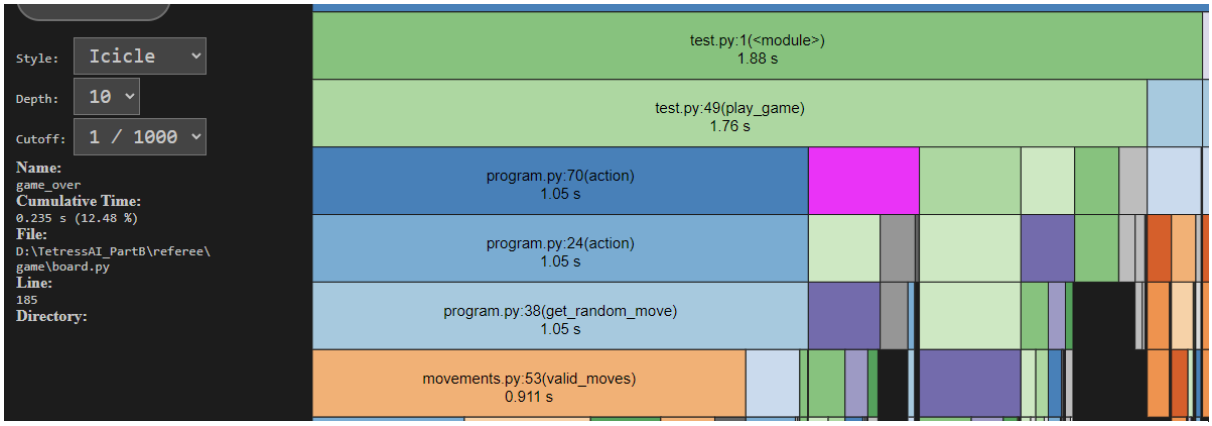

Figure 1

To use cProfile, a test environment was created in test.py where the actions of the agents were looped over and their states updated accordingly, in effect simulating the game without the usage of the referee. This allowed the methods of the agents themselves to be focused upon by cProfile, and the extra time taken by the referee to be removed.

As you can see by the snakeviz diagram in Fig. 1, the game_over method is taking up a significant amount of time (.2 seconds of a total 1.8 seconds, as shown on the left). After examining this method's code, we can see that it is rather inefficiently looping over all empty coordinates and trying to place down every possible piece – regardless of whether the empty coordinates were adjacent to a piece. There is also time being taken up by _parse_place_action and _resolve_place_action, which do a lot of extra work in asserting the moves are valid, and if so, saving

the state to the board's history. This is again unnecessary, as all moves we are passing to the board have already gone through these checks.

To fix these problems and to increase efficiency, SimBoard was created. The extra checks and unnecessary extra functionalities were removed and through this, the time taken for checking game_over and placing moves was reduced significantly (see Results). The copy() method was also created to avoid the surprisingly expensive use of deepcopy(), since MCTS regularly uses this method when expanding and rolling out from current nodes. All this combined allowed MCTS to reduce the time taken for simulations and to use this extra time to its advantage.

It was also clear that the Coord class was used a very large number of times in adding and subtracting coordinates to find valid moves, etc, and seemed like it had the potential to be a bottleneck if not implemented efficiently. Its methods were being called by far the most out of any class. However, to our dismay, after experimenting with the code and realising the logic required that it already implemented, it did not seem feasible nor worthwhile to create our own version of Coord.

With the underlying code of the game satisfactorily optimised for now, we turned our attention back to MCTS agent itself.

MCTS Heuristics and Searching

The main issues with MCTS were clearly the number of simulations per turn, and time taken for each of these simulations. We soon realised that it wasn't necessary to play out the entire simulation every time, as the large branching factor meant the game could go several different ways past a certain depth, and it wasn't worth playing out entirely every time. Instead, we changed the playout to only go to certain depth, which after reaching estimated if we were winning or not by basing it on a heuristic that determines the number of moves available to us and to the opponent (we also tried a more aggressive heuristics that only based on opponents' available moves, but it was not suitable for certain cases). This significantly reduced the time simulations took whilst still giving the MCTS a good idea of what moves were good and bad.

Continuing this idea of not bothering with exploring relatively useless branches, we realised the time spent on the first couple turns of the game by MCTS was pretty much useless.

$$Possible\ moves = 11^2 \times 19 \times 4 = 9196$$

There are almost 10,000 different possible moves to play for the first turn and trying to look at only a few of these moves and see if they're beneficial or not is effectively useless. This time was saved by instead randomly selecting actions for the first 6 turns (which still have incredibly high branching factors), to allow more time for later moves where this time can be better used.

Even after some turns have been skipped, it is still not easy to run the required amount of MCTS simulations to make an informed decision. So, we decided to use dynamic depth and simulation number to alter the MCTS search each turn based on number of available moves. If there is a large amount available moves (more than 100), the search is only allowed to rollout 4 steps ahead. And if there are far fewer available moves, the depth is extended to 8 steps for better reliability. Since the branching factor highly relies on the number of following available moves, the depth change will not significantly affect the time cost and it is worth it if predicts the route to win.

After several competitions against random, we then found that some "clever" random agents happen to survive intensive situations and play the long game, which makes our agent run out of time. To avoid such cases, we added the time management system to approximate the time cost of each turn to make sure we have enough time for future turns. Based on *Time Management for Monte Carlo Tree Search* by Hendrik Baier and Mark H. M. Winands, we found that we can limit the searching time by predicting how many turns left till the end of the game, and then just divide the remaining time by the number of estimated turns remaining. This number of estimated turns is found by approximating the number of turns taken by simulations – we chose to run three simulations for this estimation as it seemed the most cost-efficient whilst still giving a good estimate. Through this, it knows to save time if it is far away from finishing the game, and it knows to spend more time thinking in more complex situations.

However, it is still possible to run out of time. A backup time system is introduced to stop estimating time and return random moves if there are only few seconds left. Though it is almost a lost since the agent is unable to think now, this might help to waste opponent's time and create chances to win.

Further Optimisations

After reading one of the tutors on Ed talk about more efficiently representing the state of the game using bits, we were intrigued. It seemed rather ingenious to cut out the middleman of representing the game as a dictionary of PlayerColour and Coord, which as stated earlier have heavily used functions. Bitwise operations, while more complex to write, are obviously far more efficient when computing due to all computers at their cores being sets of bitwise operations.

There were a lot of methods to convert to use these new bitwise operations however, so it seemed an arduous task. The game state of the board was first split up into red and blue states represented by bits[1]. This allowed the state to be more efficiently represented as it meant placing down pieces was now a simple bitwise OR with bit shifting by the computed index of each coordinate, and similarly removing cells when clearing lines used bitwise AND with bit shifting. The same general logic as sim board for adding pieces, checking if game over, etc was thankfully allowed to be used, and it was not ultimately too difficult to convert the necessary code. Unfortunately, it did not ultimately make a meaningful impact to the performance of the game.

Conclusion and Results

Performance of MCTS vs Random

The MCTS agent easily won against the Random agent, winning 24/25 times in our testing. It should be noted that games lost against the Random agent are likely due to the estimated number of turns being incorrect, and causing the MCTS agent to either spend too long on a move that does not meaningfully contribute to the game – or rushes through the end of the game, incorrectly thinking there are plenty of moves remaining.

Performances of Random v Random (Simulation Performances)

Number of games: 1000, Depth: unlimited

	Total time	*Average time per game*	*Games per second*
Board	331.4879	0.3315	4.0167
SimBoard	328.0228	0.3280	3.0486
BitBoard	329.8957	0.3299	3.0313

Number of games: 1000, Depth: 8

	Total time	*Average time per game*	*Games per second*
Board	16.9651	0.0169	58.9445
SimBoard	8.3267	0.0083	120.0956
BitBoard	10.0509	0.0101	99.4932

SimBoard doubles the efficiency of Board in the more useful data shown by the second table, and as you can unfortunately see here, was also 20% more efficient than BitBoard. This is likely due to its methods not taking full advantage of its bitwise operations – for example mostly using the costly "state" property to convert BitBoard to work with the existing methods designed for the other boards. With more time, we are sure BitBoard could have been made more efficient.

Machine Learning

There are several things we could have done better if we had more time to apply machine learning:

The estimated turns are sometimes not reliable since the simulation cannot always predict the future moves. The number of simulations to determine the average number of turns is limited, and too many simulations can waste valuable time, so we aimed to balance this.

Also, the c_param is another constant hard to balance. It is not only essential for MCTS to control the exploration and pick the best moves, but also can ruin the further search if it is in an improper number and instead always prioritises e.g., expanding the same nodes over and over rather than exploration.

The constants to control when to use different search states are important, too. We cannot find the best numbers to define if a branching factor is too high, or if the depth is always suitable for some cases without further research. All these constants could have been balanced well using machine learning, and if we had been given time this agent would be far more powerful.

References:

1. Baier, H., & Winands, M. H. M. (2016). Time Management for Monte Carlo Tree Search. IEEE Transactions on Computational Intelligence and AI in Games, 8(3), 301–314. https://doi.org/10.1109/tciaig.2015.2443123