COMP30024 Project Part B Report

Introduction

The main approach that is used for this project is the utilisation of the minimax adversarial search algorithm optimised using alpha beta pruning. It is an algorithm that allows the game playing agent to make optimised decisions based on the utility function that has been set. Alpha beta pruning is used to decrease the computation cost of the algorithm in order to fit with the given constraints.

Methodology

Minimax Algorithm

The minimax algorithm is commonly used for zero-sum, two-player perfect-information games that are deterministic. It is designed to determine the optimal moves for the player. The idea is to choose the move that has the highest minimax value or the best achievable payoff against best play. The structure of the Tetress game fits with this algorithm since it is sequential, two-player and deterministic. The following outlines how the algorithm works in the context of the Tetress game:

1. Represent the game as a tree, where the nodes represent a possible game state and the edges represent possible moves.
2. A utility function is used to compute the value of terminal nodes or game states where the game is over (in this case, where no possible moves can be made or when the depth limit is reached).
3. Starting from the current game state, the algorithm recursively explores the tree and considers all the possible moves by considering both the maximising player (current player) and the minimising player (opponent).

   A maximising player chooses moves that maximise their score. Thus at each level of the tree, the player selects the move that leads to the state with the highest utility score. The maximising player is set as the red player.

   A minimising player chooses moves that minimise the opponent's score. Thus at each level of the tree, the player selects the move that leads to the state with the lowest utility score. The maximising player is set as the blue player.

   Thus this algorithm assumes that the player wants to maximise their score while the opponent wants to minimise the player's score.

4. When the algorithm reaches the maximum depth of the tree, it backtracks to higher levels and propagates the utility score upwards to intermediate nodes. At each level, there is alternation between the maximising and minimising players because of the sequential nature of the game. Note that for every chosen game state in the tree, a function is called to check whether any filled rows or/and columns are present and eliminates the blocks that do form completed lines to accurately represent the state of the game.

5. Once the entire tree has been explored within the given max moves and max depth constraints, the algorithm selects the move at root level which leads to the highest utility score.

The minimax function essentially generates a tree containing the possible moves and game states made by those moves, traverses the tree by considering how the player (maximizer) and opponent (minimizer) make their moves and returns the move that leads to the game state which has the highest utility score. When put into the context of this game, this algorithm is complete since the game has a finite number of possible moves and therefore a finite number of game states. It is optimal only if it is against an optimal opponent since it assumes that the opponent is minimising the player's score. Since the algorithm uses depth-first search to traverse the game tree, its space complexity is O(bm). Without any optimization, the time complexity is O(b^m). Note that b = branching factor and m = max depth.

Utility Function

As mentioned above, the utility function determines the value of each terminal node in the minimax tree. The score is generated from various factors listed below given the chosen game state:

1. If the terminal node is at max depth, the utility score increases given the amount of possible moves left that can be made.
2. The utility score increases/decreases based on the difference in the number of occupied blocks between the player and opponent.
3. The utility score increases/decreases based on the negative of the difference in the number of blocks between this terminal game state and the present game state. Meaning that if the number of total occupied blocks in the present game state is higher than that of the terminal game state, the utility score increases and vice versa. This takes into account line elimination.
4. The number of eliminated blocks (difference between the red/blue blocks in the terminal and present game state) belonging to the player and opponent will be computed. The utility score will decrease by the number of eliminated blocks belonging to the player and increase by the number eliminated blocks belonging to the opponent.
5. Based on the total number of occupied blocks in the present game board, the weighting for these factors will be adjusted where, as the total number of blocks increases, the number of eliminated blocks belonging to the player/opponent and the total difference in the number of blocks will have more weighting since as the game progresses, decreasing the number of blocks that the opponent has becomes a higher priority. Given the same logic, the other 2 factors will also increase in weight but not to the same extent.
6. If there are no possible moves for the player at the chosen terminal game state, the score will be negative infinity since it is a game state where the player loses. If there are no possible moves for the opponent, the score will be positive infinity since it is a winning game state.

<u>Optimisation</u>

Alpha Beta Pruning

Alpha beta pruning was applied to the minimax algorithm to remove any nodes that don't affect the final move. It does this by eliminating branches of the tree that are guaranteed to be worse than previously examined branches.

During the traversal of the minimax tree, the algorithm maintains two values, alpha and beta. Alpha represents the maximum score found so far along the path for the maximising player while beta represents the minimum score found so far along the path for the minimising player. Initially, these values are positive and negative infinity respectively. These are the scores of intermediate nodes which are based on the scores of the child nodes. For nodes belonging to the maximising player, the score of the node is the max of its children's score while for the nodes belonging to the minimising player, the score is min of its children's score.

When a maximising player lands on a game state with a utility score greater than or equal to beta, it knows that the minimising player won't choose this branch because the minimising player aims to minimise the score, and it already has a better option along another branch. Therefore, the current branch and all its sub branches can be pruned. Similarly, when a minimising player lands on a game state with a score less than or equal to alpha, it knows that the maximising player won't choose this branch because the maximising player aims to maximise the score, and it already has a better option along another branch. Therefore, the current branch and all its sub branches can be pruned.

Assuming perfect ordering, the time complexity for the minimax algorithm when integrated with alpha beta pruning becomes $O(b^{(m/2)})$.

Cutoff Test

A depth and move limit (cutoff test) was placed on the minimax tree to further decrease computation cost. This limits the amount of moves that the algorithm traverses through and stops searching when the limit is reached. If the algorithm traverses through a tree too deep and reaches the depth limit, it will also get cut off and the search stops. The terminal nodes located at the depth and move limit have their scores computed and the move which leads to the node with the highest score will be returned.

As the game progresses, it is possible to afford higher depth and move limits due to the decreasing number of possible moves and game states. As such, the cutoff test which was initially constant becomes dynamic, where, as the number of total blocks in the current game board increases, the move and depth limit follows. This results in lower time complexity because at the start of the game, there are so many possible moves and game states that it would be too computational intensive for the algorithm to find a winning or losing game state at this point. The moves made at this point are also relatively inconsequential. That is why the first move made by the algorithm is completely random.

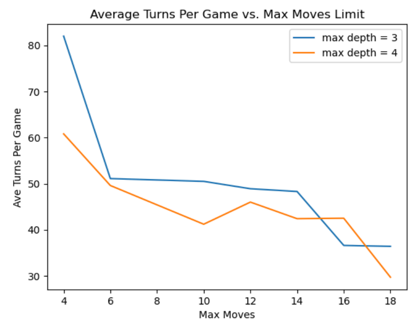
Performance Evaluation

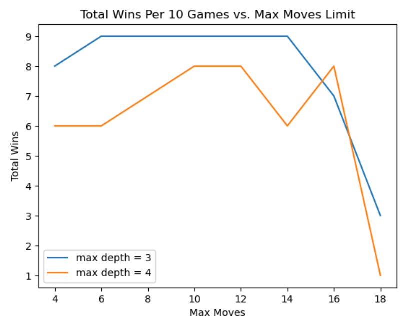
Throughout the project, the minimax agent is tested against itself, previous versions of itself and a random agent. Initially, the minimax agent was unoptimised, where before receiving the data from the random agent, the depth and move limits were static. Thus, the performance of the final minimax function was determined by testing it with the random agent and the unoptimised version of the minimax agent.

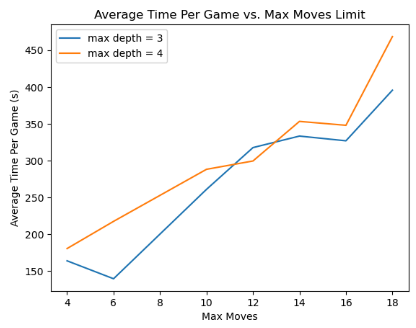
Random Agent

The first agent that was made as a performance benchmark for the minimax agent makes moves randomly by generating a random coordinate in the game board that has not been occupied and is adjacent to the other blocks of the same colour and generating 3 more random coordinates which connect to each other and the first coordinate while also not being in an occupied spot (if the 3 coordinates cannot be found after an arbitrary 500 times of generating random coordinates, the first coordinate will be removed and the whole process repeats again until a full 4 coordinates are found). These 4 coordinates make up a random move.

To initiate mass testing in order to save time, another python script was written to handle running multiple instances of games running all at once using multithreading. Each instance has the referee called using the '-s' and '-t' flags to apply the 180s time limit and 250mb peak memory usage during each game. The '-l' flag is also used to generate a log file for each game which contains the number of turns, total time and winner. These are all read by the testing script which outputs the total wins, losses, average time and turns per n number of games. For testing against the random agent, the move and depth limits of the minimax agent were changed and for each datapoint, 10 games were played. The results of these were plotted using matplotlib and jupyter notebook. Note that when testing on a different machine, the performance for the agent changes even with the same moves and depth limit applied. Losses due to the 180s time limit being surpassed were more prevalent in weaker machines. This could be due to the amount of CPUs available for each machine when running docker which hampers performance. Thus, for the sake of consistency, all of the data presented was taken using the same machine.

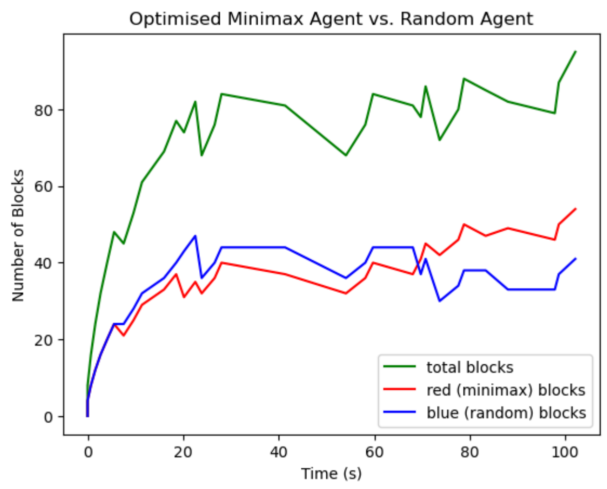






The graphs above indicate that the agent performs better at the lower depth limit of 3 overall. The move limit that allows for a greater win rate is from 10-12 max moves at both depth limits. The plot also shows how there is a decrease in the number of turns per game as the move limit increases. This is to the minimax agent surpassing the 180s time limit where it takes a longer amount of time to make moves, resulting in less moves being made. On top of this, the average time for each game increases as the move limit increases, which follows from the previous logic of the agent taking so much time to make a move. Overall, this shows that there needs to be further optimization to improve the minimax agent, particularly with the cutoff method.

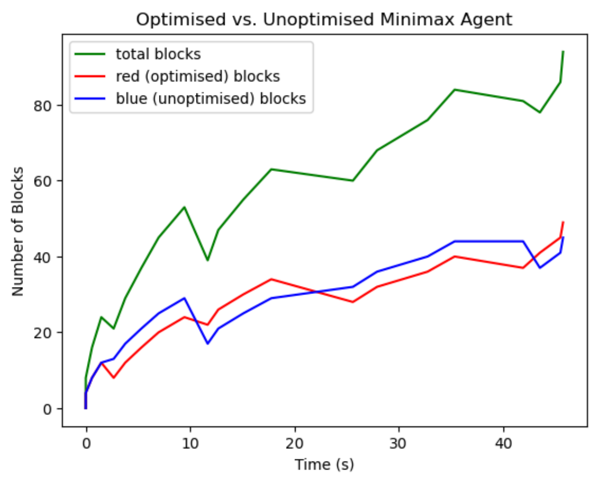
After implementing dynamic move and depth limits on the minimax algorithm, the agent is tested again with the random agent. This time, the agent takes the 'time_remaining' data from the referee right before it returns the next move. This data along with the current number of red, blue and total blocks present in the game board are then appended into a csv file which is then presented as the following plot using the pandas library.



This graph shows the performance of the optimised agent against the random agent in one game. The optimised agent was able to win the game in under the 180s time limit comfortably (only took around 100s). Note that the time taken for each turn increases as the number of total blocks increases, where the plot becomes flatter as the game progresses. This is exactly what was expected since it is the dynamic move and depth limits at work. The random agent was beaten in 32 turns (not including the turns made by the random agent).

Unoptimised Minimax Agent

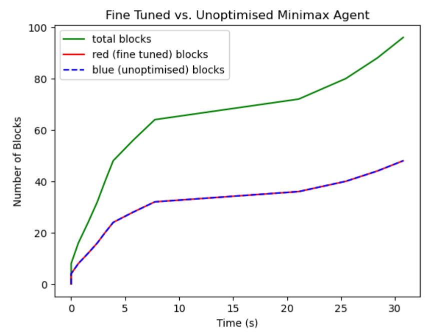
Another test was done by letting the optimised and unoptimised minimax agents compete with each other. This is to determine whether the dynamic move and depth limits were effective. Here, the move and depth limits of the unoptimised agent were set to 12 and 3 respectively, following the results above. Using the same measurement technique as before, the following plot is generated.

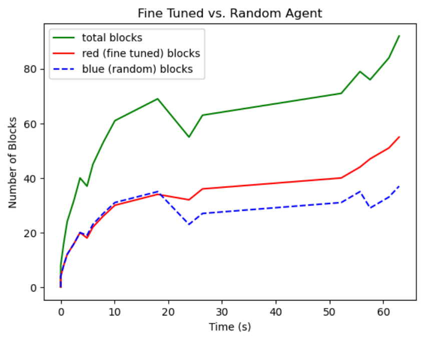


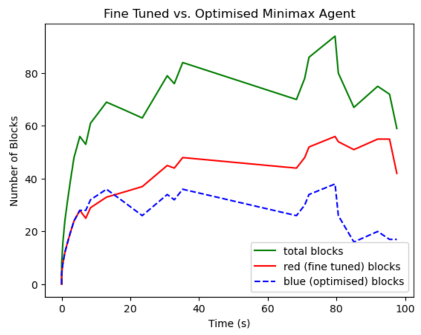
The game was much closer here, where the optimised agent edges out a win. This time, the agent was able to win in about 50s. The same feature of the curve becoming flatter as the game progresses is still present here. Since both agents use minimax, it is expected for the race to be tighter in this case. Note that the total wins for the optimised agent is 51 after 100 total games. The unoptimised agent was beaten in 21 turns.

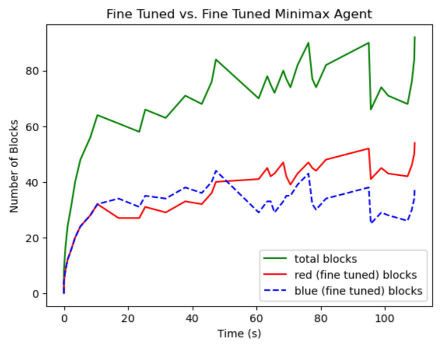
Fine Tuning

Using the performance metrics above, the optimised minimax agent was then fine tuned, where the weighting for each factor in the utility function and the number of blocks at which the move and depth limit changes are tuned to further optimise the time complexity of the agent. It was found that through multiple trial and error testing, when the difference between the number of blocks the player has compared to the opponent is weighted much higher than the rest (about two orders of magnitude higher), the performance of the algorithm increases. But this weighting drastically changes as the number of blocks in the game board increases, where more weighting is given towards the number of possible moves for the gamestate as the game progresses. The move limit also increases from 2 to 256 as the game progresses. Here are the plots generated from testing with the random agent, unoptimised agent and the optimised agent that hasn't been fine tuned.









From the plots above, there seems to be a high improvement over the optimised minimax agent, where the algorithm was able to beat it quite handily (note that this game ended due to the optimised agent surpassing the 180s time limit). In terms of turns, the algorithm was able to beat the random, unoptimised and optimised agents in 18, 13 and 26 turns respectively, which is a great improvement over the metrics for the optimised agent. Moreover, when testing with 100 games, the fine tuned agent was able to beat the unoptimised agent 77 times, which is also a sizable improvement.

The graph on top shows how the fine tuned agent performs against itself where the red player took 37 turns to beat the opponent. Because this algorithm was the most effective when testing it with the other agents, it was chosen to be the final submission.

<u>Conclusion</u>

Overall, the Tetress game playing agent was successfully implemented using the adversarial game playing algorithm Minimax, along with α-β pruning and cutoff tests to improve time complexity. Through testing the agent with itself, its previous iterations and a random agent, fine tuning of the utility function, move and depth limits were possible using trial and error. Possible extensions include the implementation of Monte Carlo Tree Search and using machine learning to retrieve the relevant parameters to improve and fine tune the adversarial search algorithm.