COMP30024 AI Project 2 Report

Team: Algorithmic Insight

Introduction

This report outlines how we developed our final agent's strategy. We primarily utilised the Minimax algorithm, which we learned about in class, and created a unique heuristic evaluation function tailored specifically for Tetress. We will explain the optimisations and implementation details of the agent, which segues into an analysis of the time and space complexity of this agent. Furthermore, this report will examine the different adversarial game play strategies we trialled and a detailed explanation for why they fall short compared to the final chosen strategy. Lastly, we will discuss how we evaluated the performance of the agent as well as potential future improvements.

1. Approach Description

Our final agent is built upon an Iterative Deepening Minimax algorithm enhanced with alpha-beta pruning. The game of Tetress is deterministic and fully observable, which makes Minimax suitable. In this section, we'll delve into the evaluation function employed by our agent. Subsequent sections will provide further insights into the implementation intricacies and the optimisations we've applied.

We adapted our implementation so that the RED player is always the maximising player, and the BLUE player is always the minimising one. Consequently, boards with positive values are inherently more favourable to RED, while negative values are preferable for BLUE.

Heuristic Function

To assess the desirability of a non-terminal game state, our agent relies on a heuristic evaluation function. One of the victory conditions in our game occurs when turn 150 is reached, and the player has a greater number of tokens on the board. To guide the agent's decisions towards achieving this condition, our heuristic function strategically assigns scores to game boards.

We consider 2 factors when determining this heuristic score. Firstly, we consider the number of tokens of each colour that currently occupies the board. If there are more RED tokens then we believe the state will be more desirable for the RED player, and vice versa for blue. This directly links to ensuring the agent has more tokens than its opponent on the board. Secondly, we evaluate the number of rows and columns with six or more cells of each player. A higher count of such rows and columns reduces the desirability of the board state for the respective player, as filling them completely in the future would result in more loss of tokens due to the line removal mechanism, i.e. six or more tokens out of eleven. The evaluation function is described mathematically below, and can be found under the *eval()* function inside agent's program.py.

$$h(s) = rt - bt - (rl - bl)\text{, where}$$

rt = the number of red tokens on the state s
bt = the number of blue tokens on the state s
rl = the number of rows and columns where red occupies more than 6 of the cells
bl = the number of rows and columns where blue occupies more than 6 of the cells

Value for Terminal States

If the state is a winning state for the RED or BLUE player (terminal state), then the evaluation function will return a winning constant. We avoided using positive and negative infinity for winning state evaluation because it interferes with the values of alpha and beta used in alpha-beta pruning. Therefore, we selected two numbers whose value is either larger or smaller than any possible evaluation score using the heuristic formula. We selected the constants +999 and -999 for the RED and BLUE player win states respectively.

These values are crucial to the minimax algorithm. It ensures that if a win can be secured within the number of depth that have been looked ahead, then the agent will make sure to make that move. On the other hand, if a move can lead to a loss if the opponent plays optimally, then the agent will avoid playing that move.

2. Implementation Details and Optimisations

To improve and optimise our final agent, we primarily made use of Iterative Deepening Minimax and a custom Python class, *Board*, to represent the game states.

Iterative Deepening Minimax

Iterative deepening (ID) Minimax is a well-known modification of the Minimax algorithm that allows us to give the Minimax function-call a time restriction, and the algorithm will iteratively try to explore as deep as possible within the time limit by incrementing the depth starting from depth equalling to one. This is highly useful for two reasons: 1) it ensures that the program runs within the 180 seconds time limit imposed by the referee and 2) it allows Minimax to search into high depths when the branching factor reduces, particularly in the late game. Additionally, each time the *action()* method is called, we maintain the possible moves for the states explored through Minimax to prevent the regeneration of the same moves. This is done with Python dictionaries to keep track of all the moves we generate with the board's hash value as key, and a list of its valid moves as the value, allowing for quick lookup.

When determining the exact amount of time we should assign to each call of the ID Minimax, we conducted rigorous testing with a benchmark of ensuring that the average time taken to decide a move is roughly two seconds. We understand that the branching factor can vary from game to game drastically given the line-removal mechanism that can increase the branching factor unexpectedly. Hence we take a more **conservative approach** where we set a smaller time limit per turn to ensure the agent does not exceed the overall 180 seconds time restriction, rather than risking taking more time per turn to explore more depths.

After conducting the simulations, we decided to make use of ID Minimax with the following parameters:

- If the branching factor is larger than **200**, then we perform a Minimax search at a depth of one. This is a greedy approach, as it is safe to assume the game is not close to an end if the branching factor is this large.
- Otherwise, we set the time limit for ID Minimax to be **0.5 seconds**.

At first, we played the game using minimax with a consistent depth of two, which is the highest depth that provided reasonable runtime across every turn in our implementation. This is where we realised the evaluation scores of each state are very similar, hovering between -1 and +1, when the branching factor is larger than roughly 200. This suggests that when most of the game board is empty, a depth of two for Minimax does not yield meaningful evaluation scores. Furthermore, logically thinking about relatively-empty boards, it is difficult to determine how desirable a move is. Therefore, to conserve time usage in these cases where the desirability is difficult to gauge, we decided to only search with a depth of one for them, in other words, take a greedy approach.

In addition, the 0.5 seconds is a limit for the minimum time the ID Minimax runs for. We do not want to stop a run of Minimax for a certain depth in the middle and waste the computation already done for that depth. This approach guards against wasting computational effort already invested in exploring a particular depth. Consequently, certain turns may take a lot more time for the algorithm to determine the best move, especially considering the exponential increase in branching factor. However, given our conservative approach mentioned earlier, and after testing the algorithm with the 0.5 seconds limit, the final agent does meet the benchmark requirement.

Custom Board Class

We implemented a custom Board class to optimise the representation of a given state. This class contains two Python sets of *Coords*, one contains all the coordinates occupied by the RED player and the other contains all the coordinates occupied by the BLUE player. This ensures that no duplicate *Coords* are stored and allows for faster lookup than Python lists.

Attempted Optimal Ordering

Since optimal ordering is known to reduce the runtime of the Minimax algorithm as alpha-beta pruning is likely to be more efficient, we attempted to reorder the list of children states based on their evaluation score in previous turns. But we did not observe any substantial improvement in runtime. We hypothesise that this is due to the nature of the game as evaluation scores can change quite fast and unexpectedly given the line-removal rule. Additionally, performing this sorting still does not ensure perfect ordering.

Without the mentioned ordering, the final agent is already performing within the time limit for all the games during our testing. Sorting takes up the time resource without guaranteeing saving more time overall. Therefore, we decided not to sort, to give time for the agent to explore more depths.

3. Time and Space Complexity

3.1 Time complexity

This agent runs in $O(b^m)$ for each turn where *b* is the branching factor and *m* is the maximum depth. The maximum depth is variable given the ID Minimax implementation.

3.2 Space complexity

For each call to the *action()* method, the agent will store all the board states expanded using generate_all_moves() within the call in a dictionary. This dictionary is discarded when the *action()* method returns. This means the space required is $O(bm)$ for each turn.

4. Other Approaches

Throughout the project, we built five different agents including the final agent, each taking a different approach to game play. The other four agents implemented are: 1) Random agent, 2) Monte Carlo Tree Search (MCTS) agent, 3) Greedy agent, and 4) Predetermined-depth agent.

Random agent (under folder 'randy'): this is the first agent we implemented, which randomly plays a valid move at each turn it plays. This is used as a benchmark to assess the 'more intelligent' agents we build later.

MCTS agent (under folder 'monte'): in this agent, we attempted to implement the MCTS algorithm discussed in the lectures. We ultimately decided against using this agent because we noticed that the branching factor is very high in the early and mid game stages. Given our implementation of MCTS and other helper functions such as move generation, the MCTS agent usually does not have enough time to perform a sufficient number of rollouts for all of the children states for a given game (parent) state, let alone the children states of the children states. Therefore, we observed the choice of *PlaceAction* to be almost random in the early and mid game. In comparison, our final agent can yield better results with minimax.

Greedy agent (under folder 'greedy'): this is a modification of our final Minimax agent, it always runs the Minimax algorithm at a depth of only 1 for each given state, hence greedy. This was used to further test the final agent.

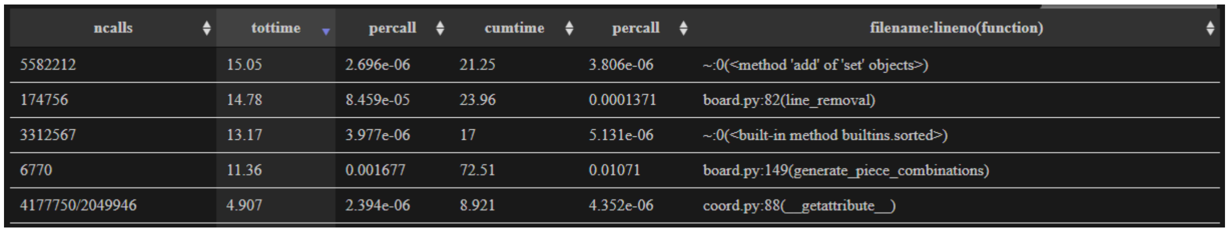
Predetermined-depth agent (under folder 'notiter'): this is the previous version of our final Minimax agent that determines the depth of a Minimax call using the branching factor at a given state. Instead of giving the agent a time restriction like in Iterative Deepening Minimax, we determine the depth from the start of the call based on the branching factor of the current board. After testing between the final agent and this predetermined-depth agent, we noticed that the final agent performed better overall.

5. Performance Evaluation

To evaluate the performance of our agents, we used the Python *time* and *datetime* module to time the amount of time used by our agents. We pitted them against each other numerous times and recorded the outcome to identify trends.

For testing, debugging, and code profiling purposes, we implemented a simple custom referee (see ref.py) that facilitates a game of Tetress. We utilised cProfile to profile all the function calls that occur during an entire gameplay and analysed the runtime of each function and method.

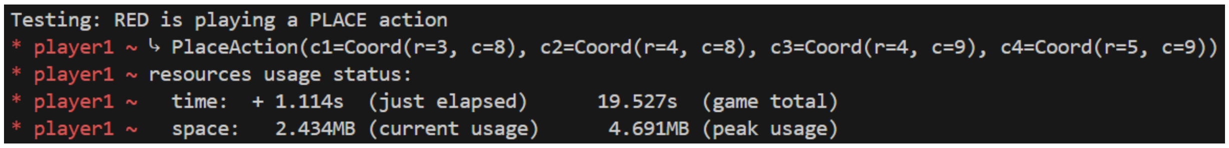
Later, we visualised the profiling with *snakeviz* to identify the most time-consuming functions and methods. This largely guided us through the optimising process. For example, *Figure 5.1* displays the result of a profiling for the final agent in an early stage of development. The functions and methods that used the largest amount of time (tottime) are clearly outlined and we focused our efforts on improving their efficiency.



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
5582212	15.05	2.696e-06	21.25	3.806e-06	~:0(<method 'add' of 'set' objects>)
174756	14.78	8.459e-05	23.96	0.0001371	board.py:82(line_removal)
3312567	13.17	3.977e-06	17	5.131e-06	~:0(<built-in method builtins.sorted>)
6770	11.36	0.001677	72.51	0.01071	board.py:149(generate_piece_combinations)
4177750/2049946	4.907	2.394e-06	8.921	4.352e-06	coord.py:88(__getattribute__)

Figure 5.1: a cProfile output visualised with snakeviz, displaying the top 5 function calls that take up the most total time in one game of Tetress.

Once we settled on the algorithm and built a basic version of the agent, we ran simulations using the command **`'python -m referee [agent name] [agent name] -v 3'`** in the terminal. This displays the time and space used for one turn, and also the total time used over the whole game *(Figure 5.2)*. Based on these numbers, we tweaked the cutoff for a large branching factor and the time limit allocated to deciding a move.



```
Testing: RED is playing a PLACE action
* player1 ~ ↳ PlaceAction(c1=Coord(r=3, c=8), c2=Coord(r=4, c=8), c3=Coord(r=4, c=9), c4=Coord(r=5, c=9))
* player1 ~ resources usage status:
* player1 ~   time:  + 1.114s  (just elapsed)      19.527s  (game total)
* player1 ~   space:   2.434MB (current usage)      4.691MB (peak usage)
```

Figure 5.2: terminal output displaying time and space usage at each turn (player 1 is the final agent). This is an example where the time elapsed is larger than the 0.5 limit.

The final agent's performance was assessed against all other implemented agents. It engaged in 30 games as the RED player and 30 games as the BLUE player against each opponent. (See Table 5.3).

Player 1 Agent (RED)	Player 1 (RED) Wins	Player 2 (BLUE) Wins	Player 2 Agent (BLUE)
Agent	30	0	**Random**
Random	1	29	**Agent**
Agent	29	1	**MCTS**
MCTS	2	28	**Agent**
Agent	30	0	**Greedy**
Greedy	2	28	**Agent**
Agent	17	13	**Predetermined-depth**
Predetermined-depth	14	16	**Agent**

Table 5.3: match results between the final agent ('Agent') and all other agents implemented throughout the project

6. Future Improvements

To improve the agent in the future, we hope to attempt a combination of Minimax and MCTS in one agent. When the branching factor is above a certain threshold, the agent should use Iterative Deepening Minimax because MCTS is highly likely to yield near-random results. Conversely, when the branching factor falls below a certain threshold, MCTS will be deployed, leveraging its heuristic-independent nature to explore numerous rollouts within the time constraints. To determine the threshold, we believe it would be ideal to incorporate machine learning algorithms to run a large amount of tests, rather than performing manual testing as we did for this project.