COMP30024 Part B report
Liam Anthian 1269187
Anthony Hill 1305284

Terminology:
Branching factor - BF
Greedy / alpha-beta pruning minimax hybrid - Gr-aB
Variables:
b - branching factor
s - depth of search (SET/FIXED)
d - max depth of game
h - heuristic time complexity
k - number of tokens on board

Approach

Action Selection throughout the game:
After discussion we deduced that there are generally two types of classification for the behaviours of our agents:

- Aggressive: Attempt to end the game on the opponent's turn before turn 150 by forcing there to be no available moves on their turn.
- Passive: Attempt to win the game on turn 150 by maintaining strong board presence throughout, focussing on building up our tokens and clearing our opponents'.

This aggressive approach can be altered to also include a factor that focuses on building up our agents own possible moves, allowing a customisable offensive-defensive play as we deem necessary. A mix of aggressive and passive win-seeking behaviour is determined best for maximising our chances of winning against bots of all levels of intelligence.

Our decided game-playing program adopts a dynamic approach for PlaceAction selection, adapting its strategy based on the stage of the game. We approached the game as being separated into early-game and late-game, with late-game being when at any one point a player has at most 15 possible PlaceActions given the board state; and early game being all points in the game beforehand. This was done because the early-game movements were found to be arbitrary and extremely memory-intensive, and deeper searches here provided no value. Additionally, the late-game is where the game-deciding moves are made, and therefore more resources of the game were allocated here.

This tipping point of 15 moves for late-game is not final, in that our Agent switches back and forth between strategies should an action (e.g. clearing tiles) open up more possible moves and re-enter early/mid-game. This approach prevents the use of costly searches suited for small problem spaces from being applied to broad problem spaces. The decision of 15 moves is partially arbitrary, guided by the need to be small since this move count is effectively the BF of the game.

- for a-B pruned minimax of searching depth 's' = 3 using H2 it becomes ~15^4 (~50625) moves checked per turn, if erroneously assuming 15 stays consistently. It may potentially be even higher.
    - $O(b^s) * O(h) = b^s * b = 15^4$.

We also considered accounting for other game states and have listed some unimplemented ideas below:

- End-game Heuristic Weighting: Nearing game end by turn count but still plenty of moves remaining - switch to greater passive heuristic weighting to maximise our tokens on the board
- Midgame: at for example 60 (arbitrary/untested cutoff) remaining possible moves, switch to a-B but with a shallow depth e.g. 2; aiming to balance resources so slightly more resources are used at mid-game where decisions are semi-important.

Initial move

The first_move function strategically determines the initial placement in a game. When playing first, our agent selects a random coordinate when the board is empty to maintain impartiality in placement - this is due to the lack of strategic advantage at initial game state due to the toroidal game domain. For the second player, a random coordinate of the first player's piece is chosen to place off of, in order to minimise opponents expansion in one direction.

First placement also avoids I-shaped tetrominoes to prevent extending the play area excessively in one direction, balancing flexibility in future movements and risk management as less tokens populate a particular axis.

(Our first move selection remained consistent across all agents produced.)

Search Algorithms:

We implement three main search algorithms:

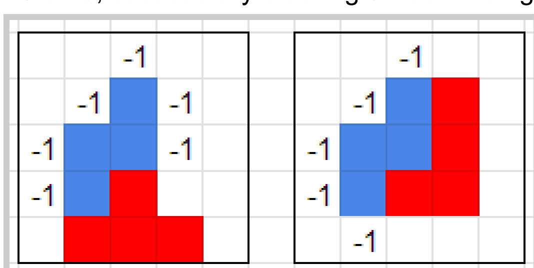
- Greedy search: At times when more complex searches branch too severely, we greedy-search our way to a more computable state with a combination of heuristics. It is with different weightings of these heuristics involved that our agent performs with different goals / behaviour - discussed more in the Heuristic Selection section below.
  - $O(b * h)$
- Minimax with Alpha-Beta Pruning: This classic search algorithm is utilised to explore possible moves and their outcomes. By traversing through the game tree to a specified depth, our agent evaluates potential moves via a heuristic and selects the one that maximises its chances of winning/places it most advantageously. Alpha-beta pruning enhances the efficiency of this process by eliminating unnecessary branches, which is crucial given the BF of this game domain. We found minimax search without pruning timed out more often than it succeeded, especially at greater depths.
  - $O(b^d)$ – cannot assume pruning will be successful; if it were ideal ordering, $O(b^{d/2})$
- Monte Carlo Tree Search (MCTS): MCTS is employed as a complementary approach to further explore the game space. We found this method is *theoretically* particularly effective in this game context where the branching gamestate space is too large to traverse exhaustively, and would allow our agent to make informed decisions based on statistical outcomes. The emphasis on *theoretically* here is important – as it currently stands, generating a singular random child of a gamestate requires generating all moves leading to all possible children, and thus, each training of the tree isn't actually $O(d)$ but $O(b * d)$ (where d is the max depth of the game).
  - To train the tree simply 10 times is up to 10*b*150 calculations - MCTS becomes only plausible in late-game given game constraints. Performs better than the default minimax in terms of calculation time still, but still substandard.

Strategic Motivations and Heuristic Selection:

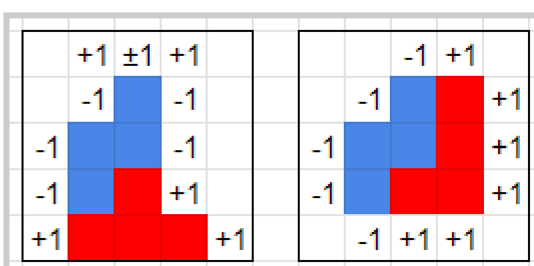
Our agent employs various heuristics to guide its decision-making process. These heuristics work around minimising the opponent's possible moves, tracking opponent token count, and assessing the protection level of its own tokens. These metrics influence the strategic decisions of our agent.

Most of our heuristics, although mathematically different, have been ultimately managed in a way such that they focus on minimising the number of moves an opponent has to choose from:

- <u>H1</u>: By directing the bot to maximise its own token count while minimising opponent token count, it prepares for a 150 turn game outcome - the player with the most tokens wins in this scenario. Additionally, the less tokens our opponent has, the less options for moves they typically have as there are less starting coords from which they can centre and place a piece. In turn, we have more power to direct the game towards more advantageous board states leading to winning outcomes.
    - $O(k)$ where *k* is the number of tokens on the board
        - (k < 121 at all times, so technically $O(1)$)
- <u>H2</u>: This heuristic looks 1 step ahead and calculates the total number of possible moves each player has from a given state, finding the difference between these figures. It aims to directly maximise our possible moves while minimising our opponents moves. However, we found that this heuristic was often too computationally expensive for even simple greedy implementations. We ended up with a complexity of $O(b)$ each turn due to the method by which we calculate possible moves, and as a result of the large BF '*b*' of the Tetress game, we deemed this calculation more often than not too slow to utilise. Only really usable in late-game, when *b* decreases.
- <u>H3</u>: An attempt at simplifying H2 to $O(k)$ where *k* is the number of tokens on the board, this heuristic aims to minimise the free spaces surrounding our opponents tokens. Instead of calculating all possible moves it equates a free connecting space to a move. When using this heuristic, our bot tends to try to 'suffocate' our opponent's agent - surround as many of their tokens as possible. It is also possible to supply H3 a weighting to assign to our own free spaces, balancing suffocating our opponent with leaving ourself room for more moves.
    - For example, viewing from Red's perspective with a 0 weighting for red free-spaces, the left state would have an H3 reading of -6, successfully blocking 2 possible tiles, and the right state an H3 of -5, successfully blocking 3 tiles. The right situation is preferred here.



    - Alternatively with wrapping and red weighting of +1; Left $(5 - 6 =- 1)$ Right $(6 - 5 = 1)$



It should be noted that for each heuristic employed here, there is a time complexity of $O(b^h)$ in greedy search, as when called they are tested on each possible child state.

Performance Evaluation

Evaluation Metrics and Data

- We evaluate our program's performance based on its win ratio against various opponent types. This includes comparing its performance when starting first (Red) versus when starting second (Blue). We have displayed this as Wins / Losses from a Red perspective below. Can be converted to Blue's perspective by inverting the fractions. All 'draw' games are exempted - treated as neither a win nor a loss.

  To achieve this evaluation, we conducted extensive testing between different bot types constructed from the above heuristics and search algorithms. This comprehensive testing provided insight that cemented theoretical predictions in some cases and changed in others.

The bots present in test 1:

1. RDM: Random move selection
2. Greedy H1: Greedy best move selection via H1
3. Greedy a-B: A hybrid minimax / greedy approach, where the alpha-beta pruned minimax relies on H2 for state evaluation at a depth of 3, and the greedy move selection relies on a heuristic combination of H3 and H1 (with more emphasis on H3).

R wins / losses	RDM	Greedy H1	Greedy a-B	Performance	Total
RDM	72/78	81/69	123/27	204/106	276/174
Greedy H1	70/80	79/71	107/43	177/123	256/194
Greedy a-B	37/113	52/98	75/74	89/211	164/285
Performance	107/193	133/167	230/70	470/430	
Total	179/271	212/238	305/144		696/653

Table 1. Wins / Losses for 1st to move (Red) agents - 3 agents

The bots present in test 2:

1. RDM: Same as test 1
2. Greedy H3: Greedy best move selection via H3
3. Greedy a-B: Same as test 1
4. a-B: Alpha-beta pruned minimax relying on *H3* for state evaluation at a depth of *2*
5. MCTS: Monte Carlo Tree Search algorithm, trained over 10 iterations each turn

R wins / losses	RDM	Greedy H3	Greedy a-B	a-B	MCTS	Performance	Total
RDM	23/27	32/18	37/13	40/10	22/28	131/69	154/96
Greedy H3	13/37	25/25	31/19	32/18	18/32	94/106	119/131
Greedy a-B	8/42	20/30	19/31	29/21	4/46	61/139	80/170
a-B	9/41	19/31	28/22	24/26	6/44	62/138	86/164
MCTS	34/16	43/7	37/13	44/6	15/35	158/42	173/77
Performance	64/136	114/86	133/67	145/55	50/150	506/494	
Total	87/163	139/111	152/98	169/81	65/185		612/638

Table 2. Wins / Losses for 1st to move (Red) agents - 5 agents

- Table 1 data was generated by playing out **150** games for each bot, versus each of 3 bots, from both starting (1st, 2nd) perspectives, totalling to **1350** games. Note that the tally is off by 1 due to a singular draw outcome occurrence that was dropped (a Greedy a-B v Greedy a-B match).
- Table 2 data from **50** games per bot, each versus 5 bots, both starting perspectives, totalling to **1250** games.
  Both processed via our handler.py program.

Key Observations from data:

- First-move advantage: When observed in the terminal, Red was observed to win at a frequency unmatched by Blue. To test this potential insight, the above data was produced. Red *seems* to win slightly more games than blue (696:653) in the grand scheme of things in test 1, which could hint at this starting bias for the first player to move.
  - Based on current data, using a one-sided binomial test with n = 1350, x = 696, p = 0.5, at CI = 0.05, p-value is 0.13223 which is >= CI and therefore there is insufficient evidence to posit there is no first-move advantage. The data is however quite small, and further testing will be needed to definitively substantiate this hypothesis.
- Passive play: While hypothetically 'Greedy' should be more informed and thus outright better than 'RDM', the experimental data shows no observed improvement with heuristic choice of H1. It does however show an advantage with H3. In turn, it is possible that passive play focussed heuristics (like H1) do little to sway match outcome by themselves, and need to be used alongside other aggressive heuristics for optimal play.
- Greedy a-B dominance: Our Greedy a-B minimax at the moment is majorly outperforming opponents, approximately winning up to 4 times for every 1 loss against the benchmark RDM bot and also triumphing over the Greedy bots. It combats default a-B well without the expensive computation that comes from running a-B from early-game.
- Self-play consistency: Agents played against self indicate consistent self-play outcomes, demonstrating algorithm stability and reliability and serves as an ideal benchmark for outcomes against other agents.
- MCTS Failure: Our MCTS agent severely underperformed any expectations we had for it, surprisingly playing even worse than RDM. This is more than likely an artefact of the bare-bones learning iterations allowed (only 10), or alternatively, possibly due to a bug or flaw causing the agent to lose earlier than actual game end. Efficiency would need to be improved to train further.

Selection Criteria and Chosen Agent:

- Our selection of our final agent is based on a comprehensive analysis of its performance compared to competitors, regarding both game outcome and observing the actual game playing itself. We balanced factors such as win ratio, time complexity, memory efficiency, and adaptability of each approach to different opposition strategies.
- By comparing the performance of our agent against alternative approaches, including different search algorithms and heuristic evaluation functions, we eventually decided on our Greedy / alpha-beta pruning minimax hybrid. Our Gr-aB strategy has the time complexity benefits of dynamic move selection discussed previously in "*Approach*", while also demonstrating the higher intelligence play of looking ahead numerous moves to fend off late-game losses when necessary. It saves processing time and memory earlier in the game, allowing it more time to think later and direct the game end towards victory.
  - It significantly outperforms adversaries that approach the game only looking one move ahead - see this domination in Table 1.
  - It holds its own against other intelligent algorithms like a-B (depth 2) without leaching both memory and time - see this in Table 2.

Other Aspects

Notes on Optimisation:
Managing the effectiveness versus the time complexity of the algorithms employed in this project was a challenge. Many times, trying to develop a smarter agent resulted in exponentially increasing computation of states, and trying to minimise time complexity by storing precalculated states would result in massively increasing memory usage. Every hurdle appeared to be some kind of hidden complex 'catch-22'. Some optimisations and tests made have been listed below:

- <u>Data Structure Hashing:</u> The significance of the benefits of hashing (dictionaries and python sets) can not be understated, and were utilised at every possible opportunity for quick lookup; Gamestate, MCTS, PriorityDict, Agenthandler - almost every Class we created stored its data in some variation of hashing.
- <u>State Caching:</u> Where possible, any calculations in algorithms (*especially* in MCTS) that would be noticeably repeated were stored in one of the custom data structures above, e.g. in MCTS, the list returned from possible_moves() to find children states is stored to be reused in a complementary Node class.
- <u>Value Wrapping:</u> Instead of performing weighty and repetitive comparisons between states themselves when choosing a best move, we often utilised a custom value wrapping class ValWrap to store states with their coupled heuristic values - thus only evaluating their value once.

Supporting work

- <u>Handler Script (handler.py):</u> The most notable script we produced to assist with the testing of our agents was `handler.py`. An entry point to simulate games between agent implementations via a slightly modified referee module, we collected data and compared agents over numerous games played out. We used this script for both quickly running tests while designing our algorithms and agents, and also automating the production of the game data seen in Table 1 of *"Performance Evaluation"*.

Future Improvements

Given more resources and time, some changes we can incorporate to improve performance of our game-playing agent are as follows:

- <u>Enhanced Learning Algorithms:</u> Developing adaptive learning techniques such as *reinforcement learning* could significantly improve our agent's performance. This approach would allow the agent to learn from past games and adjust its strategies dynamically based on the opponent's behaviour, potentially uncovering more effective game-winning behaviours.
  - For example, linear regression with the weightings of combined heuristics would help balance the aggression and defensiveness of our agent, leading to a well rounded bot for all adversaries.
- <u>Algorithm Optimization:</u> Further refinement of our search algorithms, particularly Monte Carlo Tree Search (MCTS), could enhance efficiency and effectiveness. Exploring optimization techniques like reducing the branching factor through better move ordering could lead to memory, time and efficacy improvements. Furthermore, heuristic inclusion in simulation selection could reduce memory usage to fit within game constraints.