By: Adam Eldaly & Thomas Lam

Project Part B: Tetress A.I Agent

Abstract

Tetress is a competitive, two-player adaptation of Tetris played on an 11x11 toroidal board, where players known as Red and Blue take turns placing tetrominoes. Pieces must be placed adjacent to another of the same colour, except on the first move, under conditions of perfect information where both players see the entire board. The goal is not just to fit pieces and clear lines but also to block the opponent's moves. The game has a 150-turn limit, with the winner being the player with the most tokens left or the one whose opponent can no longer make a valid move, emphasising strategic play and foresight.

A.I Agent

The debate on adversarial search techniques focused on Mini-Max with alpha-beta pruning versus Monte-Carlo Tree Search, each having distinct advantages and drawbacks. Considering the domain's large average branching factor of approximately 150 valid placements per turn, as aggregated from various simulations, Monte-Carlo was speculated to be more suitable with proper optimisations. To validate this hypothesis, both agents were developed and tested using empirical data to determine the most optimal approach.

MINIMAX ALPHA BETA PRUNING

EVALUATION/GAME STRATEGY

To effectively strategise in Tetress, especially when using the Mini-Max algorithm where reaching a terminal state within deep search limits is unlikely, the quantitative metric of Board Control becomes vital. This metric evaluates the difference in potential actions between the player and their opponent based on the current board state. A positive Board Control value signifies a strategic advantage, indicating more available moves and control of the board. Conversely, a negative value points to a disadvantage, stressing the importance of clearing lines, especially those with more opponent tiles, to open up the board for more moves and mitigate the Board Control deficit.

Strategically, the aim is to prevent the opponent from clearing lines or creating new opportunities for moves. This is done by avoiding placements that clear lines and prioritising those that create gaps. These gaps make it costly for the opponent to remove rows or columns, thereby pushing the game towards a condition where the opponent runs out of viable moves.

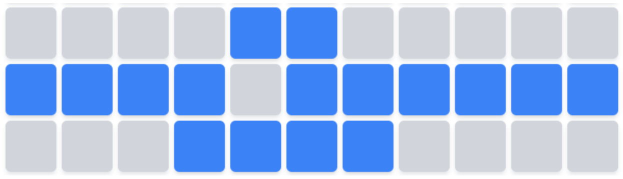
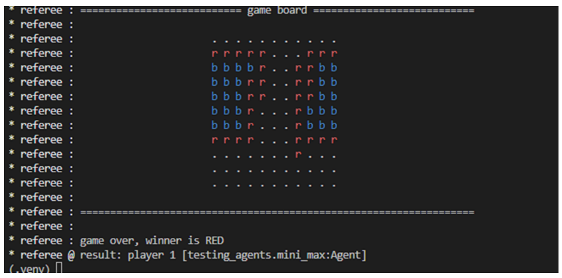


FIGURE1: GAP REFERENCE

Another key strategy in Tetress, particularly relevant in the context of Mini-Max evaluation, is to end the game with a greater number of tiles on the board than the opponent. As the game approaches its conclusion, monitoring and expanding the tile count disparity becomes critical. This can be effectively achieved by strategically clearing lines predominantly filled with the opponent's tiles. Such actions not only reduce the opponent's tile count but also disrupt their strategic setup. This tactic diminishes the opponent's future move options and bolsters the player's position by increasing their tile count lead. Each move is carefully calculated to extend this numerical advantage, ensuring a stronger position at the game's end.



```
* referee : ======================== game board ========================
* referee :
* referee :                       . . . . . . . . . . .
* referee :                       r r r r r . . . r r r
* referee :                       b b b b r . . r r b b
* referee :                       b b b r r . . r r b b
* referee :                       b b b r r . . r r b b
* referee :                       b b b r . . . r b b b
* referee :                       b b b r . . . r b b b
* referee :                       r r r r . . . r r r r
* referee :                       . . . . . . . r . . .
* referee :                       . . . . . . . . . . .
* referee :                       . . . . . . . . . . .
* referee :
* referee : ===========================================================
* referee :
* referee : game over, winner is RED
* referee @ result: player 1 [testing_agents.mini_max:Agent]
(.venv) 
```

FIGURE2: WRAPPING OPPONENT

For both agents in Tetress, the optimal strategy is to encircle the opponent's pieces early on. This tactic limits their ability to expand and escape, effectively reducing their viable move options and steering the game towards a favourable outcome

By: Adam Eldaly & Thomas Lam

ALGORITHM ANALYSIS

The Minimax algorithm, with its combinatorial nature, faces a high computational load, exhibiting a theoretical worst-case time complexity of $O(b^{150}))$, where b is the branching factor representing the maximum number of moves. This complexity makes it impractical for the tight time constraint of 180 seconds. Although its space complexity is more manageable at (O(b* m) using depth-first search, it doesn't mitigate the significant time burden. Nonetheless, the bitboard implementation ensures the 250MB memory limit is never exceeded.

To enhance efficiency, Alpha-Beta Pruning is integrated to potentially reduce the best-case time complexity to $O\left(\frac{b \cdot m}{2}\right)$ and average-case to $O(\frac{3b \cdot m}{4})$ by cutting off irrelevant search branches early. However, Tetress's game mechanics, involving child generation and valid piece placements across a fixed grid, complicate effective move ordering. In Tetress, optimal moves can appear anywhere on the board without an inherent order that consistently aids in early pruning.

To accommodate the strict time constraints, a strategic depth cutoff at a smaller value was established. The accompanying graph indicates the most optimal depth to be of 2. This strategy allows the agent to explore the most advantageous moves while avoiding premature termination of potentially winning paths.

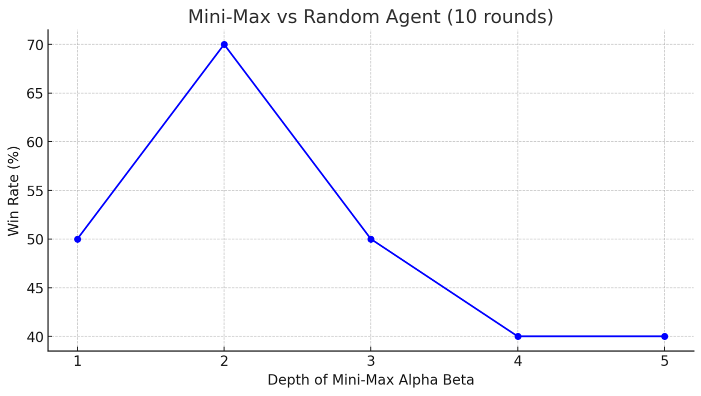


FIGURE3: OPTIMAL DEPTH

MONTE-CARLO TREE SEARCH

Monte Carlo Tree Search (MCTS) is an adaptable algorithm for optimising decisions in complex games like Tetress, where traditional methods like Minimax are less effective due to high branching factors and limited computation time. Unlike Minimax, MCTS doesn't rely on a complete search tree; it uses random simulations to explore a broad range of possible game states efficiently.

Theoretical Framework of MCTS for Tetress

MCTS involves four key steps: selection, expansion, simulation, and backpropagation.

Selection: In Tetress, reusing the Monte Carlo Tree Search (MCTS) tree across multiple turns proved impractical; on average, only 1.5 out of 150 turns benefited from tree reuse due to the overhead of maintaining and validating the tree.

Consequently, the strategy was shifted to restart the root node with each new action, simplifying the process and maintaining relevance to the game's current state. With a branching factor b and total tree nodes t, the tree's depth is $(\log_b(t))$, and the selection phase, involving traversing b children at each level to find the highest UCB1 value, has an estimated time complexity of $(O(b \cdot \log_b(t)))$

Expansion: Upon reaching a leaf node in the Monte Carlo Tree Search (MCTS), all valid placements based on the preceding player's colours are generated as potential children of the node, expanding the tree with future game states. This expansion is crucial for exploring viable strategic options and operates with a constant time complexity if an array stores the node's children. However, when considering all possible placements, the algorithm was only exploring to a depth of one due to prioritising unvisited nodes, rendering the UCB1 score redundant if a node isn't explored. This limitation was identified through our visualisation program and indicated that merely adjusting the exploration constant in MCTS was insufficient to resolve this issue. Consequently, we strategically limited the children's generation to a branching factor of 25, a necessary tweak to enhance exploration and ensure more effective decision-making in the tree.

By: Adam Eldaly & Thomas Lam

Simulation: In the MCTS framework for Tetress, simulations start from each expanded node to estimate potential game outcomes, randomly proceeding until a terminal state—either 150 turns or no valid placements are possible—is reached. On average, 8 simulations are conducted per turn. This limited number can affect the statistical reliability of outcomes, as more simulations typically yield a more accurate expected value of a move. However, increasing the number of simulations might not be feasible due to computational or time constraints.

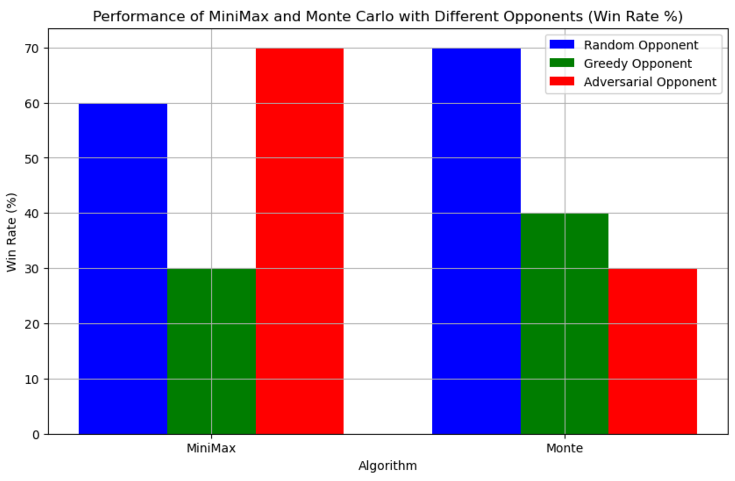
Simulating involves finding and executing the first available move for a given board state, a process with constant time complexity. This repeats until the terminal state is reached. Early in the game, or when simulations do not end before 150 moves, the worst average time complexity is $(O(150))$. However, since the depth of the current node is $(\log_b(t))$, the deeper the node, the closer it is to the terminal state, reducing the average time complexity, which can be generalised as $(O(1/\log_b(t)))$.

Backpropagation: The outcomes from the simulations in the MCTS framework for Tetress are propagated back up the tree, updating nodes with statistical information about win probabilities. In this zero-sum game setup, updates are straightforward: a win for our player results in marking all traversed node paths as wins; a loss does not mark them as wins. This simplifies the update process as the Upper Confidence Bound 1 (UCB1) value used during selection does not differentiate between our player's nodes and the opponent's. The time complexity of this backpropagation is determined by the depth from which it starts, $(O(\log_b(t)))$.

Summing the complexities of all phases—selection $(O(b \cdot \log_b(t)))$, simulation $(O(1/\log_b(t)))$, and backpropagation $(O(\log_b(t)))$—the overall time complexity becomes $(O(\log_b(t^{b+1} + 1/\log_b(t))))$.

PERFORMANCE ANALYSIS

Having implemented both Monte Carlo and MiniMax AI algorithms, extensive research was conducted to determine the superior AI agent for submission. The effectiveness of each AI was assessed by simulating 10 games against different algorithms: one that places



random moves and a greedy algorithm utilising the same strategy aforementioned for Mini-Max. Additionally, 10 direct games were conducted between the MiniMax and Monte Carlo agents to further evaluate their performance.

Based on the graphs, MiniMax has a 60% win rate against a Random AI, while Monte Carlo slightly outperforms it with a 70% win rate. Against a greedy algorithm, Monte Carlo also leads with a 40% win rate compared to MiniMax's 30%. Intriguingly, despite these statistics suggesting Monte Carlo's overall superiority, MiniMax significantly outperformed Monte Carlo in direct matchups, achieving a surprising 70% win rate.

The unexpected result may stem from Monte Carlo's generalistic strategies formed through multiple simulations, compared to MiniMax's clearer, focused strategy prioritising placements that enhance Board Control and Domination value. This generality in Monte Carlo's approach might also explain its higher win rates against the Random and Greedy agents, as it adapts more effectively to the nature of its opponents than MiniMax.

MiniMax's 30% win rate against the Greedy algorithm can be attributed to its time-intensive strategy of looking ahead, which contrasts with the Greedy algorithm's focus on immediate gains without forward planning. In simulations, the Greedy agent often won by reaching the 150 action limit and maintaining a higher tile count. While both algorithms prioritise high Board Control and Domination values, MiniMax's forward-looking approach is hindered by strict time constraints, leading it to potentially overlook the most optimal current moves. This disadvantage becomes particularly significant in

Tetress's final stages, where immediate actions are crucial.

After analysing the data, the Monte Carlo AI will be chosen for submission. Despite its generalistic strategies potentially underperforming against AIs with clearer tactics, it demonstrates higher win rates against certain opponents compared to MiniMax. This shows its advantage in adapting beyond a limited set of strategies.

OPTIMISATIONS

BITBOARD

For the Agent in Tetress, selecting an optimal data structure for internal game representation was key. The bitboard representation, fitting the game's fully observable and deterministic nature on an 11x11 toroidal grid, provides faster processing compared to the 'Board' class. The latter uses a hashmap linking the 'Coord' class to 'PlayerColour' and is better for game monitoring and interaction but incurs more overhead with its management of elements like 'CellMutation' and 'CellState'. While useful for certain tasks, this increases abstraction and reduces efficiency. The Bitboard approach, in contrast, is vital for the AI's computational needs, enabling quicker state evaluations and updates necessary for exploring more nodes in time-limited adversarial searches.

Implementation Details

Each agent within the game utilises an instance of the BitBoard class. This class includes:

- A separate bitboard for each player's piece placement on the board.

- A third bitboard that superimposes both players' pieces to provide a comprehensive view of the game state.

In this representation, each bit of a bitboard denotes the state of a cell on the board, indicating whether a cell is **occupied (1)** or **empty (0)**.

Key Operations

- **Horizontal Movement:** Shift bits by multiples of 11 to move pieces left or right, reflecting the contiguous row representation in the bitboard.
- **Vertical Movement:** Use divmod with the divisor as 11 (number of columns) for vertical shifts. This adjusts the row index and wraps movements around the board edges, where the remainder determines the column.
- **OR Operator:** Combines tiles by ORing the bit pattern of a new piece with the player's bitboard, updating the board state to reflect newly occupied cells.
- **AND Operator:** Checks for overlaps by ANDing the two players' bitboards. A nonzero result indicates a clash, showing that a move is invalid due to overlapping pieces

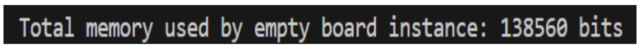
Memory Efficiency

The BitBoard class demonstrates exceptional memory efficiency:

- Each BitBoard instance contains 121 set bits. (11*11 Board Size)

- Each agent manages 3 bitboards, leading to a total of 363 bits per agent.

This configuration results in a significant reduction in memory usage approximately ***3,817%*** when compared to the memory consumption of the traditional 'Board' class

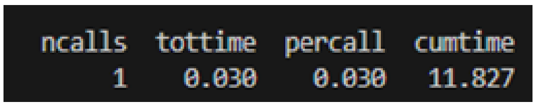


```
Total memory used by empty board instance: 138560 bits
```

FIGURE4: 'BoardClass' Memory

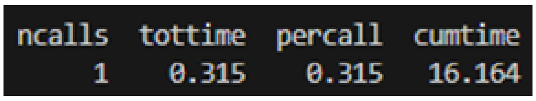
Time Efficiency:

For benchmarking, we compared the time taken to generate and place all valid pieces using the same board state. Due to the terminal's three-decimal place display limitation, we scaled and repeated the process: 1,000 turns for the Board class and 100,000 cycles for the Bitboard. The Board class required 0.01183 seconds per piece, while the Bitboard needed only 0.000162 seconds, marking a 98.63% reduction in time.



```
ncalls  tottime  percall  cumtime
     1    0.030    0.030   11.827
```

FIGURE5: 'BoardClass' 1,000 Cycles



```
ncalls  tottime  percall  cumtime
     1    0.315    0.315   16.164
```

By: Adam Eldaly & Thomas Lam

FIGURE6: BitBoard 100,000 Cycles

According to *(Figure 7,8)* shows that without optimisation techniques, adversarial search methods become impractical. For example, the Monte-Carlo approach conducted only 1.8 simulations on average, rendering piece selection nearly random. This contrasts with the Bitboard's 131 simulations, demonstrating a significant 98.63% improvement. Meanwhile, the Mini-Max algorithm with alpha-beta pruning explored an average of 34.90 nodes, about one-third of the typical first-level depth, leading to suboptimal performance and making it an ineffective strategy for greedy searches, similar to a depth-1 Mini-Max.

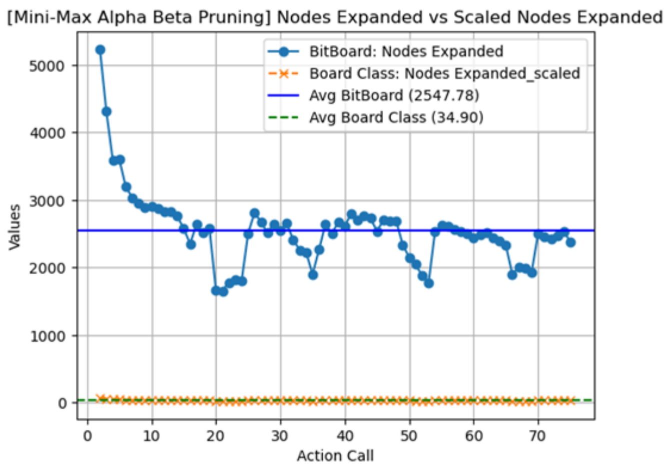


FIGURE7: Mini-Max BitBoard vs Board Class

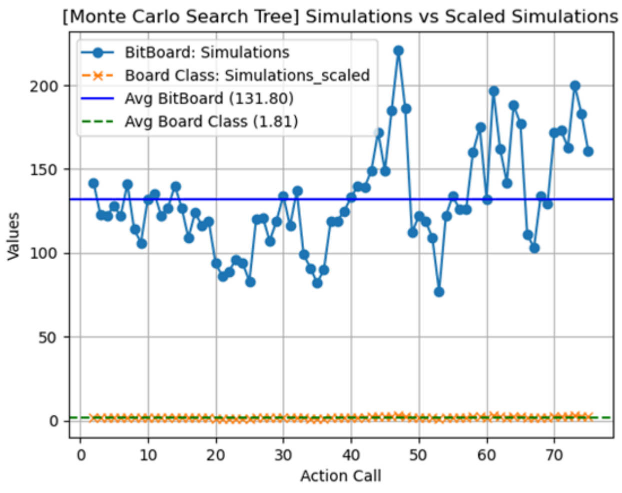


FIGURE8: Monte-Carlo BitBoard vs Board Class

SCRIPTS

- We utilised the cProfiler module to analyse the runtime of our 'action' and 'update' methods within the game agents. This analysis indicated that the runtime of the Mini-Max algorithm increases monotonically, often surpassing the predetermined limit of 2.4 seconds per turn due to its recursive approach in iterative deepening. To mitigate this, we implemented a Timeout Exception mechanism. This mechanism effectively interrupts the recursion stack, returning control to the top-level initial invocation, thereby stabilising the runtime at approximately 2.4 seconds per turn. This adjustment is crucial as it prevents the agent from exceeding time constraints and ensures consistent performance throughout the game.

- To facilitate the creation of graphical visualisations, the Pandas library and Matplotlib were employed for processing various CSV files. Data extraction was conducted using several externally developed custom profiler classes, each tailored to collect specific data that assists in decision-making and insight generation. However, integrating these processes directly into the game agents introduced additional computational overhead due to the demands of simulating and managing game instances. Thus, they were directly embedded within the agents to optimise performance during gameplay.

- Additionally, a function ‘print_traverse_tree’ was implemented to visualise the Monte Carlo tree after each action. This revealed that due to the high branching factor, the tree was typically only exploring to a depth of one level, as Monte Carlo prioritises unvisited nodes regardless of their UCB1 scores. To optimise performance, we adjusted our approach to limit the children's generation to a branching factor of 30, which proved most effective.

- Finally, we developed scripts to pre-compute various bitboards. These scripts generate bitboards for multiple purposes:
  1. Identify all pieces that could potentially occupy a given cell, aiding in the determination of valid placements

2. Track all bitboards that represent a full row or column, streamlining the line clearance process
3. Map out adjacent tiles for any given tile. This third set of bitboards facilitates efficient lookups for adjacent tiles in bitboard representation, compensating for the absence of a directional class.