PROJECT REPORT

Tetress Game Playing Agent

By Long Nguyen & Jenny Mai

MCTS Agent Approach

MCTS Agent employs the Monte Carlo Tree Search (MCTS) algorithm to select actions throughout the game. We believe MCTS is a suitable choice for Tetris due to its ability to efficiently explore the vast game tree, balancing exploration and exploitation. We attempted this approach although MCTS can be challenging to apply successfully.

MCTS Implementation

Our MCTS implementation consists of the following key components:

Node Selection: We use the UCT (Upper Confidence bound applied to Trees) formula to select the child node with the highest UCT value, balancing exploration and exploitation.

Expansion: We create new child nodes for each possible move, simulating the game tree expansion. As this results in an extremely large branching factor and significant runtime, we decided to only keep the 'best' 10 child nodes, chosen by the highest UCT scores estimated by our heuristic evaluation function (HEF).

Simulation: We play a random playout from the new node to the end of the game, estimating the value of each possible move.

Backpropagation: We update the statistics of the nodes from the new node to the root, propagating the results of the playout using the UCB1 formula + the HEF value of the new node weighted by 0.5.

Heuristic Evaluation

Our evaluation function, `heuristic_evaluation`, assesses the quality of each possible move based on various strategic considerations. The evaluation function consists of two main components:

Cell Count Ratio: We calculate the ratio of the player's cell count to their opponent's cell count. This metric rewards moves that increase the player's cell count and penalizes moves that reduce it. We implemented taking a weighted sum of these counts that prioritizes line clearing. However, we later decided that our strategy would be to play and end the game fast to avoid runtime issues rather than keep clearing lines and playing the game longer.

Holes Penalty: We count the number of holes (disjoint sets of empty blocks of >4 cells) on the board using the `count_holes` function. We encourage nodes with even numbers of holes and penalize those with odd numbers of holes. As we tested playing the game, the winner is usually known a few turns before the game ends. With the nodes being our possible moves, we want an even number of holes on the board (most desirably 2). In that case, as the opponent takes 1 turn, if no lines are cleared, it will be *more likely* our agent makes the final move and wins.

The final evaluation value is a sum of these two components. We used to implement other components such as board control and blocking ability as well. However, those are removed because of adding significant runtime without increasing much effectiveness in heuristics. The `heuristic_evaluation` function returns a floating-point value, with higher values indicating more favorable actions.

We also initially implemented encouraging moves where the opponent will have the least possible moves on their turn. However, our `generate_possible_move` function is the most expensive utility function with high runtime and memory, which always takes roughly 85% runtime, evaluated locally using cProfile. Therefore, it is not ideal to call `generate_possible_move` in the HEF.

Ending: This `Ending` class chooses actions when the game board has > 85 filled cells (estimated to have 3 turns left for each player before the game ends). This class returns the action where the opponent will have the least possible moves (ideally 0) on their turn. This strategy is realistic to use closer to game_over because our `generate_move` function attempts to fit pieces into empty cells, which are quite few as the game ends, making its runtime fast and manageable to call multiple times. Moreover, our MCTS considering the 10 best moves evaluated by HEF with several weighted components might not guarantee choosing the one to win the game. The `Ending` class guarantees choosing the one to win if any.

UCT Formula

We use a variant of the UCT formula, incorporating the exploration_constant parameter to control the trade-off between exploration and exploitation. We also incorporated the HEF value weighted by 0.4 into calculating the UCT scores.

At first, we initiate the UCT scores of all new BoardNodes to infinity as per the existing algorithm. However, due to runtime constraints, we can only conduct 25-40 iterations per turn. Thus, our MCT might still have unvisited children with infinity UCT scores. This unvisited child will be chosen as the best child with the highest UCT. In this case, we changed to initialize UCT scores to HEF values initially so that the MCTS Agent avoids choosing unexplored children. But even when it does, it behaves like a greedy agent. However, we have tuned the weights of different heuristic components so that the initial UCT scores are lower than those of explored nodes nearly all the time, preventing choosing unexplored children.

We chose an exploration constant (C) of 0.2, which controls the trade-off between exploration and exploitation. A higher value of C encourages exploration, while a lower value encourages exploitation. With that value, our agent encourages exploitation to get as much information about children nodes as possible, balanced with some level of exploration. In testing, it normally exploits up to depth 6th of a few best child nodes from depth 1.

Minimax Agent Approach

Motivation

Our initial approach for this game-playing agent involved the Minimax algorithm integrated with α-β pruning. This decision is based on the nature of the algorithm, which is the ability to play the most optimal under the assumption that the opponent is also optimal.

We hypothesize that against a random opponent, in the game of Tetress which is a perfect-information, deterministic game, our Minimax agent could outperform the opponent as playing randomly in a deterministic game can be considered suboptimal play.

Implementation

Node structure:

The node represents a node in the search tree of the Minimax algorithm. It records the board state, the move that leads to that board state and player's color.

MinimaxAgent class:

This class is the main class for our Minimax implementation. It has the following key methods

1. __init__
- This method is responsible for initializing the agent with the color and keeping the board state.
2. select_move
- This method is responsible for selecting the best move based on the current board state. It iterates over all available moves from the current board, and applies each move to the current state in order to evaluate that move. The move that leads to the highest scoring board state is selected as the best move
3. minimax
- This method implements the Minimax algorithm with α-β pruning. It explores the game tree to a certain depth and uses the heuristic_evaluation function to estimate the value of each game state. α-β pruning technique is used to eliminate branches that have values lower than the current min/max value. By applying α-β pruning, the exploration speed is increased tremendously, taking into account the complexity of the board state.
4. evaluate
- We call the function heuristic_evaluation as described above to get the heuristic value of a board state.

Performance Evaluation

To evaluate the effectiveness of our game-playing program, we conducted a series of experiments comparing the performance of three agents: an MCTS Agent, a Minimax Agent (with α-β pruning), and a Random Agent.

Experimental Setup & Performance Metrics

We designed a tournament-style experiment, where each agent played 100 games against every other agent. We used the following metrics to evaluate the performance of each agent:

- Win Rate: The percentage of games won by each agent.
- Board Control: The average cell percentage at the game end of each agent.
- Game Length: The average number of turns taken by each agent to *win* a game.

Results

Agent	Win Rate	Board Control	Game Length
MCTS Agent	76%	59%	38
Minimax Agent	63%	52%	36
Random Agent	23%	43%	53

Discussion

Our results show that the MCTS Agent and the Minimax Agent are quite comparable, and both outperformed the Random Agent regarding win rate and board control. The MCTS Agent's ability to adapt to changing game conditions and make informed decisions under uncertainty contributed to its slightly higher performance.

The MCTS Agent achieved a win rate of 76%, higher than the Minimax Agent's 63% and the Random Agent's 23%. This indicates that the MCTS and Minimax agents could make more effective moves and capitalize on their opponent's mistakes.

Interestingly, the Game Length metric reveals that the MCTS Agent took an average of 38 turns, and the Minimax Agent took 36 turns, compared to the Random Agent's 53 turns. This can likely be the effect of our `Ending` class pushing the game towards ending faster.

Other Aspects & Supporting Work

We developed the count_holes function to count the number of blocks with more than 4 continuous empty cells using disjoint set data structures. This optimization enabled us to improve the performance of our evaluation function and make more informed decisions during gameplay. We utilized disjoint set data structures to efficiently manage and union sets of coordinates, allowing us to quickly identify connected components and count holes on the board.

We wrote a custom simulation script that allows us to simulate games between two agents without relying on the referee module. This enabled us to more accurately examine the runtime of our functions and identify areas for improvement.

We employed the cProfile library to profile our code and identify performance bottlenecks. This helped us to refine our algorithms and data structures, leading to significant improvements in runtime efficiency.

We thought of and tested out creating an Agent using Minimax in the first half of the game and MCTS for the second half of the game. However, in testing, that agent did not perform well and is just a bit better than the Random Agent. Therefore, we did not choose to go with that idea.