AI-Assignment-4

Sidhant Gumber

Part 1: Heuristic

The heuristic I developed consists of 3 parameters to score a board state, a higher heuristic score implies a more favourable state for the agent. The following are the three parts of the heuristic:

- Advanced Line Scoring: This method evaluates each row, column, and diagonal of the board for the current player's token. It looks for patterns of tokens and assigns scores based on the potential of these patterns to lead to a win. For instance, a sequence of four tokens ('wwww.') is scored higher than a sequence of two ('ww..') because it's closer to a win. This method considers both the length of the sequence and its openness (whether it can be extended on the next move). The scoring is given as such:
    - 1 point for a single w or b token in a row
    - 10 points for 2 w or b tokens in a row
    - 100 points for 3 tokens in a row
    - 1000 points for 4 tokens in a row
    - 10000 points for 5 tokens in a row: I.e terminal state.

- Twist Potential: After placing a token, consider the score change for each possible twist. The best score from these potential twists can be added to the heuristic score. This method assesses how the potential to twist quadrants of the board could improve the player's position. It does this by simulating twists for each quadrant and using the advanced_line_scoring function to evaluate the resulting board state. The highest score from these simulations is taken as the twist potential and added to the overall heuristic score.
- Mobility: Count the number of moves that lead to immediate threats, such as creating an opportunity for four in a row that can't be immediately blocked. Mobility is a measure of the number of options available to a player that would lead to immediate threats or significantly improve the board state. In this method, you simulate each possible move, apply it to the board, and use the advanced_line_scoring method to evaluate how good the move is. The more high-scoring moves available, the higher the mobility score.

Heuristic Score Evaluation :

- Board State Analysis: The current state of the game board is analyzed, looking at the arrangement of tokens for both the player and the opponent.

- Line Evaluation: Every row, column, and diagonal is passed to the score_line method to get a basic score based on consecutive tokens. The results are summed up in advanced_line_scoring.

- Twist Simulation: For each of the four quadrants, the board is virtually twisted left and right, creating new board states which are then evaluated for line scoring. The evaluate_twist_potential method captures the best possible improvement in score due to a twist.

- Move Simulation: All possible legal moves are considered. For each move, the evaluate_mobility method simulates the move, applies it to the board, and then evaluates the resulting board state for its line score.

- Heuristic Score Calculation: The scores from line evaluation, twist potential, and mobility are aggregated to form a heuristic score for the board. This score is a measure of how favorable the board is for the player: higher scores indicate a board state that is closer to a win, while lower scores indicate a state that is either neutral or unfavorable.

- Decision Making: In the context of a game tree search like Minimax, this heuristic score is used to make decisions at leaf nodes (where the search depth limit is reached). The Minimax algorithm, enhanced by the heuristic, selects the move that leads to the highest heuristic score when it's the AI's turn to move and assumes the opponent will do the same for their moves.

Part 2: Random

Implemented the random agent successfullly and can be tested using the command:

```
python main.py random random
```

Part 3: Minimax

The minimax algorithm is implemented as follows:

- Base Case: The recursion stops when it reaches a predefined depth limit or when a win condition is detected. At this point, the board is evaluated using a heuristic function.

- Maximizing Player: When simulating the AI agent's turn (the maximizing player), the algorithm explores all possible moves, applies them to simulate new board states, and chooses the move that results in the highest heuristic score.

- Minimizing Player: When simulating the opponent's turn (the minimizing player), the algorithm again explores all possible moves but chooses the one that results in the lowest heuristic score for the AI agent.

- Move Application: Moves are applied to the board using the apply_move method, which returns a new board state with the move executed.

- Heuristic Evaluation: The sg3824_h heuristic function assesses the board states to provide a score indicating how favorable a board state is for the AI agent.

IMPORTANT NOTE: In my implementation of minimax without alpha beta pruning, the agent is producing near optimal results by defeating a random agent in minimum number of moves possible, HOWEVER, if you increase the depth limit to 2 the computation becomes significantly more expensive and moves take a lot of time. This in my opinion is expected behaviour as pentago has an extremely large branching factor. To test the only the minimax algorithm without pruning kindly stick to a depth limit of 1 by giving the command:

```
” python main.py minimax random -d 1 ”
```

Testing on my system typically produced a search time of 3 to 7 seconds for each move.

Part 4- Alpha Beta Pruning

Alpha-Beta Search Algorithm

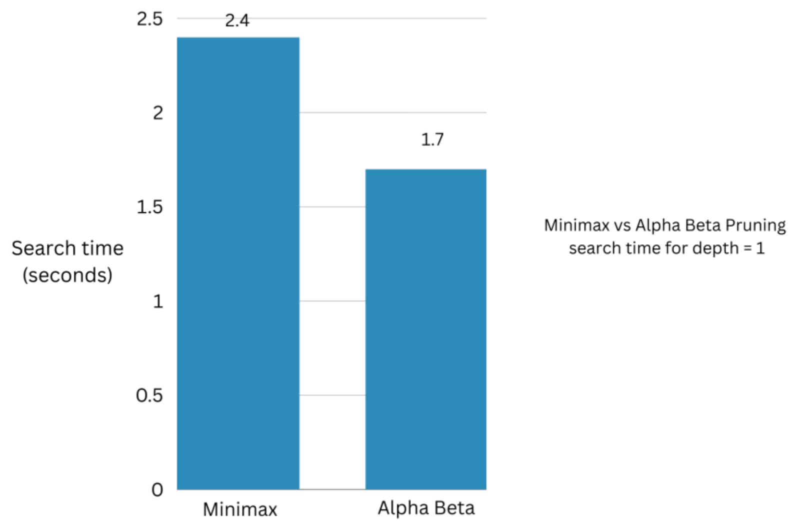
The alphabeta method is a recursive function that explores the game tree, applying alpha-beta pruning. It alternates between maximizing and minimizing turns, tracking the best possible outcomes for both players with the alpha and beta parameters. Pruning occurs when the condition alpha >= beta is met, eliminating the need to explore further down that branch of the tree.

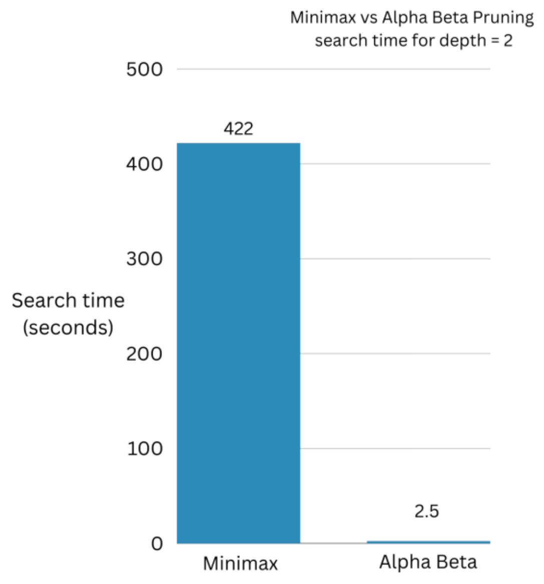
NOTE: Even after pruning, the decrease in runtime was not as much as I expected it to be, although I could see significant improvements at depth 2, to further improve the algorithm, I implemented dynamic depth adjustment which basically adjusts the depth based on the number of empty cells on the board, which serves as an indicator of the game phase:

- Early Game: When more than half of the board is empty, a shallower search is performed. This reflects the high branching factor and the relative insignificance of deep searches due to the many possibilities.
- Midgame: As the board fills up beyond a certain threshold, the search depth is increased moderately.
- Endgame: With fewer empty cells and more potential for game-deciding moves, the algorithm searches to the maximum specified depth.

Search time: Typical search time for a move for a depth of 2 lied between 1.5-3.5 seconds with the search time being reduced with every subsequent move as the tree search became a lot more streamlined as number of empty cells kept reducing.

SEARCH TIME COMPARISONS:





Part 5 - Minimax + MCTS

Overview

The Minimax_mcts class implements a sophisticated agent for playing the board game Pentago, combining a two-move lookahead using Minimax or Alpha-beta pruning with Monte Carlo Tree Search (MCTS). This hybrid approach aims to leverage the deterministic strength of Minimax to evaluate immediate moves and the stochastic assessment of MCTS for deeper game states.

Design Considerations:

- Minimax Lookahead: The agent begins by assessing the potential moves using a heuristic evaluation. This step simplifies the move selection by considering a limited number of promising moves, which are then explored further with MCTS.

- Depth Limitation: The depth for the Minimax portion is limited to two moves—one for the AI agent and one for the opponent. This depth was chosen to balance the computational cost with the benefit of strategic foresight.

- Move Evaluation: For each candidate move from the Minimax lookahead, the algorithm performs an Alpha-beta search to evaluate the opponent's potential responses, further refining the selection of moves for the MCTS phase.

- Monte Carlo Tree Search: MCTS is applied after the Minimax lookahead. For each selected move, the agent simulates numerous random playouts to completion. The move that results in the most favorable outcomes (e.g., wins for the AI agent) is chosen.

- Performance Optimization: Each move's computation is timed to monitor performance. This information can be used to adjust the number of playouts or tweak the depth of the lookahead.

- Number of Playouts: The number of playouts per move in the MCTS phase is a key parameter that affects both the accuracy of the move assessment and the computation time. This parameter should be set high enough to ensure a reliable evaluation but not so high as to cause excessive delays. For the purposes of this assignment I have set it to be 100.

Implementation

The minimax_for_mcts method initially uses the heuristic to rank all possible moves and selects the top 'n'. It then applies Alpha-beta pruning to evaluate the moves from the opponent's perspective.

The get_move method orchestrates the MCTS phase. It times each move's evaluation and prints these times, providing insight into the computational demand of each move's analysis.

The playout function simulates random games from a given board state, alternating moves between the player and opponent until a terminal state is reached.

Time Measurements

The timing measurements for each move provide valuable data for analyzing the performance. A typical move takes between 20-35 seconds on my system.