Gomoku AI Implementation Report

This project was developed by our team, consisting of Mohammed Yasin (BSSE-1406), Sejuti Sharmin (BSSE-1420), and Habibur Rahman (BSSE-1422). The following report details the implementation of the Gomoku AI in `Logic.py`, focusing on the evaluation criteria, search algorithms, and optimization techniques used to fulfill the project requirements. The AI is designed to play a competitive game of Gomoku on a 10×10 board against a human or AI opponent, adhering to standard rules where the goal is to form a line of five stones.

Evaluation Function

The evaluation function, implemented in `evaluate_board` and `evaluate_position`, assesses the board's state by assigning scores to patterns of consecutive stones for both the AI (player 2) and the human (player 1). Key patterns include:

- Five or more stones: 1,000,000 points (winning position).
- Open four (four stones with both ends open): 100,000 points.
- Closed four (one end blocked): 10,000 points.
- Open three: 1,000 points.
- Closed three: 100 points.
- Open two: 10 points.

The function evaluates only positions with stones from the move history to optimize performance. Immediate threats (winning moves) add 500,000 points, prioritizing critical positions. The final score is the AI's score minus the player's, guiding the AI toward favorable states.

Search Algorithm and Minimax with Alpha-Beta Pruning

The AI employs the minimax algorithm with alpha-beta pruning, implemented in `minimax`, to search the game tree and select optimal moves. The algorithm explores a depth-limited tree (default depth of 3, adjusted to 4 for fewer moves), alternating between maximizing (AI) and minimizing (player) scores. Alpha-beta pruning reduces the search space by cutting off branches where the outcome cannot affect the final decision, improving efficiency. A transposition table caches board state evaluations using a hash of the board, avoiding redundant computations.

Move Prioritization and Early Stopping

The `find_best_move` function prioritizes moves to enhance decision-making:

1. Immediate AI win (`check_immediate_threat`).
2. Blocking player's immediate win.
3. Creating an open four (`find_open_four_move`).
4. Blocking player's open three (`find_block_open_three_move`).
5. Minimax for remaining moves.

This prioritization serves as an early stopping mechanism, allowing the AI to select high-priority moves (e.g., winning or blocking moves) without deep minimax searches. The `get_relevant_moves` function optimizes by limiting the search to empty positions within a 2-cell radius of existing stones, sorted by a heuristic score from `quick_evaluate_move`. The search terminates when a terminal condition (win, full board, or depth limit) is reached, returning the evaluation function's score.

Optimization Techniques

To enhance performance, the AI uses a pattern cache in `pattern_cache` to store results of expensive computations, such as win checks and threat detections, keyed by board state or position. The move history tracks played positions, reducing the need to scan the entire board. The `quick_evaluate_move` function provides a lightweight heuristic for move sorting, prioritizing promising moves early in the minimax search.

Conclusion

This project provided a valuable learning experience, deepening our understanding of game tree search, evaluation functions, and optimization techniques. The Gomoku AI demonstrates efficient and competitive gameplay through a robust evaluation system, strategic move prioritization, and performance optimizations like caching and transposition tables.