Adversarial Search for Gomoku

Gomoku, a deterministic strategy game, presents a huge challenge for adversarial search algorithms due to its
. The objective is to align five consecutive marks on an $n \times n$ grid (set to 9×9 in this report).

To address these challenges, four adversarial search algorithms were implemented:

1. **`MinMax`** : A baseline algorithm that explores all possible game states.
2. **`AlphaBeta`** : An optimized version of , pruning branches that can be confidently excluded based on the opponent's potential decisions.
3. **`AlphaBetaHeuristic`** : Enhances by using heuristic evaluations to prioritize strategic moves.
4. **`MCTS`** : Leverages simulations for probabilistic decision-making, balancing real win probabilities with exploration.

The game board is a 9×9 grid implemented with a Python-based GUI using . is denoted for User, for computer.

1. **`MinMax`** :
   - : Explores all possible moves to determine the optimal outcome for the current player through exhaustive search.
   - :
     - **`_maximize(current_depth, alpha, beta)`** : Recursive function to maximize the score for the current player.

    - `_minimize(current_depth, alpha, beta)` : Recursive function to minimize the score for the opponent.
    - Note: and parameters in are placeholders and are not used for pruning in this implementation.
  - : Computationally expensive due to exhaustive search, making it impractical for larger boards.
  - : Introduces a parameter to limit search depth and ensure reasonable response times. Without this, it would be infeasible to compute results, as the state space grows astronomically (e.g., 80! ≈ 7.1569E+118).
2. **`AlphaBeta`** :
   - : Improves by eliminating branches that cannot influence the final decision.
   - :
     - `_apply_max_pruning(alpha, max_score, beta)` : Implements pruning for maximizing player.
     - `_apply_min_pruning(alpha, min_score, beta)` : Implements pruning for minimizing player.
   - : Reduces the search space significantly, enabling deeper exploration within the same time constraints.
3. **`AlphaBetaHeuristic`** :
   - : Extends by integrating a heuristic evaluation function to estimate the desirability of intermediate board states.
   - :
     - Scores are assigned based on consecutive pieces and open ends (e.g., 4 consecutive pieces with 2 open ends are highly prioritized, followed by 4 consecutive pieces with 1 open end).
     - Penalizes the opponent's advantageous positions by blocking critical moves.
     - For detailed scoring logic, refer to the well-documented class.
   - :
     - `_calculate_heuristic_score(row, col)` : Computes the overall heuristic score for the current board state.
     - `_calculate_current_score(row, col, player)` : Assigns scores for the current player by evaluating their consecutives on the board.
     - `_calculate_opponent_score(row, col, opponent)` : Penalizes the opponent's consecutives by reducing the score for threatening positions.
     - `_evaluate_center(row, col)` : Adds a positional bonus for moves closer to the center of the board.

- : Enabling the AI to make more informed decisions within practical time constraints.

4. **`MCTS`** :
   - : Leverages simulations to estimate the probability of winning from a given state. The algorithm follows four main steps:
     a. (via **`_Node.select()`**): Traverses the tree using UCB1 to identify the most promising nodes for exploration.
     b. (via **`_Node.expand()`**): Adds a new child node to the tree.
     c. (via **`_Node.simulate()`**): Randomly simulates a game from the newly expanded node.
     d. (via **`_Node.back_propagate(winner)`**): Updates the tree with simulation results.
   - : A configurable time limit (default: 15 seconds per move) ensures the algorithm responds efficiently.
   - :
     - **`_Node.best_final_move`** : Prioritizes moves that guarantee a win. If no such move exists, selects the most visited node to maximize the likelihood of success.

Experiments were conducted on a max_depth` and simulation counts to evaluate:

> Following tests were conducted on a MacBook Pro M1 Pro chip (16GB).

1. **`MinMax`** :
   - Depth-limited to due to exponential growth in game states.
   - Struggles with longer simulations, making it easy for the user to win.
   - On my Mac ( ), it expands approximately , but the AI consistently loses due to lack of strategic depth.
2. **`AlphaBeta`** :
   - Runtime reduced by compared to rmthrough branch pruning.
   - At , results are produced in .Mla%esults Increasing

3. `AlphaBetaHeuristic` :
   - Demonstrated improvements in AI' intelligent, effectively blocking user moves, prioritizing the center, and constructing consecutive pieces.
   - On my Mac ( ), the first move takes , and subsequent moves take due to reduced board complexity. Performance improves significantly as the game progresses.
4. `MCTS` :
   - Faces challenges in deterministic winning moves during endgame scenarios without fine-tuning .
   - Fixed to (configurable). On my Mac, it performs approximately . The AI is significantly smarter, consistently blocking user moves and forming its own 5-consecutive pieces.

( )	11	~500,000	Baseline for comparison
( )	0.2	~6,000	Efficient pruning for deeper exploration
	2 (average after 1st)	~30,000 (average)	Strategic decisions, blocks, and center focus
	15	~30,000 simulations	Adaptable probabilistic analysis, strategic wins

: Win Rate (%) is influenced by the user's skill level. However, it is evident that the AI improves significantly in strategic decision-making with each algorithm upgrade.

- The Python package was used to handle the GUI. The initial GUI code was generated using ChatGPT-4 and subsequently modified and improved by me.
- All algorithm implementations, including , , , and , (single person in a group).
- This report was reviewed using ChatGPT to ensure proper English grammar and clarity.