Adversarial Search for Gomoku

Introduction

Gomoku, a deterministic strategy game, presents a huge challenge for adversarial search algorithms due to its **vast state space (approximately n^2! possible states, making it infeasible to expand all nodes)**. 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 `MinMax` , pruning branches that can be confidently excluded based on the opponent's potential decisions.
3. **`AlphaBetaHeuristic`** : Enhances `AlphaBeta` by using heuristic evaluations to prioritize strategic moves.
4. **`MCTS` (Monte Carlo Tree Search)**: Leverages simulations for probabilistic decision-making, balancing real win probabilities with exploration.

Methodology

Game Setup

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

Algorithm Implementations

1. **`MinMax`** :
   - **Overview**: Explores all possible moves to determine the optimal outcome for the current player through exhaustive search.
   - **Key Functions**:
     - **`_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: `alpha` and `beta` parameters in `MinMax` are placeholders and are not used for pruning in this implementation.
  - **Limitations**: Computationally expensive due to exhaustive search, making it impractical for larger boards.
  - **Optimization**: Introduces a `max_depth` 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`** :
  - **Overview**: Improves `MinMax` by eliminating branches that cannot influence the final decision.
  - **Key Functions**:
    - **`_apply_max_pruning(alpha, max_score, beta)`** : Implements pruning for maximizing player.
    - **`_apply_min_pruning(alpha, min_score, beta)`** : Implements pruning for minimizing player.
  - **Impact**: Reduces the search space significantly, enabling deeper exploration within the same time constraints.
3. **`AlphaBetaHeuristic`** :
  - **Overview**: Extends `AlphaBeta` by integrating a heuristic evaluation function to estimate the desirability of intermediate board states.
  - **Heuristic Design**:
    - 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 `HeuristicScore` class.
  - **Key Functions**:
    - **`_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.

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

4. **`MCTS` (Monte Carlo Tree Search)**:
   - **Overview**: Leverages simulations to estimate the probability of winning from a given state. The algorithm follows four main steps:
     a. **Selection** (via **`_Node.select()`**): Traverses the tree using UCB1 to identify the most promising nodes for exploration.
     b. **Expansion** (via **`_Node.expand()`**): Adds a new child node to the tree.
     c. **Simulation** (via **`_Node.simulate()`**): Randomly simulates a game from the newly expanded node.
     d. **Backpropagation** (via **`_Node.back_propagate(winner)`**): Updates the tree with simulation results.
   - **Customization**: A configurable time limit (default: 15 seconds per move) ensures the algorithm responds efficiently.
   - **Final Move Selection**:
     - **`_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.

Results Analysis

Experiment Setup

Experiments were conducted on a `9×9 board with varying` max_depth` and simulation counts to evaluate:

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

1. **`MinMax`**:
   - Depth-limited to `max_depth=3` due to exponential growth in game states.
   - Struggles with longer simulations, making it easy for the user to win.
   - On my Mac (`max_depth=3`), it expands approximately **500,000 nodes in 11 seconds**, but the AI consistently loses due to lack of strategic depth.
2. **`AlphaBeta`**:
   - Runtime reduced by **30%-50%** compared to `MinMax` through branch pruning.
   - At `max_depth=3`, results are produced in **0.2 seconds with 6,000 nodes expanded**. Increasing `max_depth` to 5 results in **11 seconds runtime and 500,000 nodes expanded**, matching `MinMax`'s runtime but with slightly better strategic decisions. However, AI still struggles to win.

3. **`AlphaBetaHeuristic`** :
    - Demonstrated improvements in AI' intelligent, effectively blocking user moves, prioritizing the center, and constructing consecutive pieces.
    - On my Mac (`max_depth=3`), the first move takes **6 seconds for ~100,000 nodes**, and subsequent moves take **2 seconds for ~30,000 nodes** 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 `UCB1` .
    - Fixed to **15 seconds per move** (configurable). On my Mac, it performs approximately **30,000 simulations in this timeframe**. The AI is significantly smarter, consistently blocking user moves and forming its own 5-consecutive pieces.

Performance Comparison

Algorithm	Time per Move (sec)	Nodes Explored	Key Strength
`MinMax` (`max_depth=3`)	11	~500,000	Baseline for comparison
`AlphaBeta` (`max_depth=3`)	0.2	~6,000	Efficient pruning for deeper exploration
`AlphaBetaHeuristic`	2 (average after 1st)	~30,000 (average)	Strategic decisions, blocks, and center focus
`MCTS`	15	~30,000 simulations	Adaptable probabilistic analysis, strategic wins

Note: 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.

Contributions

- The Python package `pygame` 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 `MinMax` , `AlphaBeta` , `AlphaBetaHeuristic` , and `MCTS` , **were entirely developed by me** (single person in a group).
- This report was reviewed using ChatGPT to ensure proper English grammar and clarity.