CS365 Lab B – Adversarial Games: Minimax Implementation

This repository contains a Python implementation for Part 2-A of the CS365 Lab B assignment. The script implements a game-playing AI for the board game Breakthrough using a minimax search algorithm. Two AI players use the Evasive heuristic to evaluate board states.

Files Included

-main.py

Contains the complete Python implementation of the game, including board setup, move generation, terminal state testing, minimax search with the Evasive heuristic, and the game simulation.

- README.md

This file, which explains how to run the code and describes the expected input and output.

- Part2-ADesignfile.pdf

Explains how to execute scripts and example input and outputs.

Requirements

- Python 3

No additional libraries are required; the script uses only standard Python modules such as `copy` and `random`.

Code Overview

The script is organized into several sections:

- Board Setup and Display:

  - initial_state(rows, cols, piece_rows): Generates the starting board configuration. For example, for a 5×5 board with 1 row of pieces per side, the top row is filled with white pieces (`X`) and the bottom row with black pieces (`O`), while empty cells are represented by `.`.

  - display_state(state): Prints the board in a human-readable format.

- Move Generation and State Transition:

  - get_possible_moves(state, player): Generates all legal moves for a given player. In Breakthrough, pieces move one step forward (with diagonal moves capturing or moving into empty cells).

  - apply_move(state, move, player): Applies a move (a tuple indicating the start and end coordinates) to the board, returning a new state without modifying the original state.

- Terminal State Test:

  - is_terminal(state): Checks if the game has ended. A terminal state is reached if:

    - A white piece (`X`) reaches the bottom row.

    - A black piece (`O`) reaches the top row.

    - One player loses all of their pieces.

- Minimax Search and Utility Function:

- minimax(state, depth, current_player, maximizing_player, heuristic): Implements the recursive minimax algorithm to select the best move for the current player. At terminal states, a very high positive or negative score is returned.

- heuristic_evasive(state, player): Implements the Evasive heuristic, which returns the number of pieces for the given player plus a random value (in [0, 1)) to break ties.

- Game Simulation:

- play_game(heuristic_white, heuristic_black, board_state, depth): Simulates a full game between two AI players using minimax search. It prints the board state at each move and, upon game termination, prints the winner, total moves, and capture counts.

Sample Input and Expected Output

Input

The script automatically sets up the initial board configuration. For example, it may create a 5×5 board with 1 row of pieces per side:

- White (`X`): Pieces are placed on the top row.

- Black (`O`): Pieces are placed on the bottom row.

- Empty Cells: Represented by `.`.

Expected Output

A sample run of the program might produce output similar to the following:

```
Initial board:
XXXXX
.....
.....
.....
OOOOO

Starting game: Evasive (White) vs. Evasive (Black)
Move 0: Player X
XXXXX
.....
.....
.....
OOOOO

Move 1: Player O
XXXXX
.....
.....
.....
OOOOO

...
Game Over!
Winner: X
Total moves: 15
Captures - White: 2, Black: 1
Final board state:
.....
..X..
.....
.O...
OOOOO
```

Note: Due to the use of `random.random()` for tie-breaking in the heuristic, the exact moves and final board state may vary between runs. The script sets a fixed random seed (`random.seed(42)`) to improve reproducibility.

Conclusion

This implementation provides a foundational framework for playing the Breakthrough game using a minimax search algorithm with the Evasive heuristic. It is intended as a starting point for further exploration into adversarial game strategies. Feel free to extend and modify the code to test other heuristics, enhance move generation, or optimize the search process.