Introduction to Artificial Intelligence

Lab 10 (Week 12) - Adversarial Search

2023 - 2024

May 06th, 2024

Objectives

- Implementation of MiniMax algorithm for Tic-Tac-Toe game.

Overview:

Adversarial search problems, or games, refer to competitive environments in which the goals of the agents (players) are in conflict. The Tic tac toe game is one of such environment (also called zero sum games).

To solve games using AI, we introduce the game tree concept followed by the minimax algorithm. Game states are represented by nodes in the tree, similar to planning problems. Nodes are organized in levels corresponding to each player's turn, with the "root" node at the top depicting the game's starting position. In tic-tac-toe, this is the empty grid before any Xs or Os are played. The second level under the root contains nodes representing possible states resulting from the first player's moves (X or O), called "children" of the root node.
Each second-level node has children representing states reachable by the opposing player's moves. This process continues level by level until reaching end game states, such as a player winning with three in a row or the board being full, resulting in a tie.

Your mission

LAB 10 focuses on implementing two main classes: the `TicTacToe class`, representing the game environment, and the `MinimaxAgent` class, which implements an AI agent using the minimax algorithm to choose its next move. Patterns for both classes, `TicTacToe` and `MinimaxAgent`, have been provided in an attached Python file..

Task 1:

The `TicTacToe` class includes methods to manage the game board, evaluate game states, facilitate player moves, and determine the winner. It is defined by the following attributes :

- `board`: Represents the Tic-Tac-Toe game board as a 3x3 grid.

- `human` and `comp`: Constants representing player values (-1 for human and +1 for computer).
- *h_symbol* and *c_symbol*: Symbols ('X' or 'O') used to represent human and computer players on the board.
- `depth`: Represents the number of remaining moves in the game.
- `size`: Total number of cells on the board (default 9 for a 3x3 grid).

Additionally, the `TicTacToe` class is designed to implement the following methods:

- `goal_test(state, player)`: Checks if a given player has won the game based on the current board state.
- `game_over(state)`: Determines if the game has ended in a win or draw.
- `empty_cells(state)`: Retrieves a list of empty cells (coordinates) on the board.
- `valid_move(x, y)`: Checks if a specified move (x, y) is valid (i.e., the cell is empty).
- `set_move(x, y, player)`: Updates the board with a player's move if it's a valid move.
- `render(state)`: Displays the current state of the board in the console.
- `human_turn()`: Facilitates the human player's turn by accepting input and processing the move.
- `evaluate(state)`: Evaluates the state of the board and assigns a score (+1 for computer win, -1 for human win, 0 for draw).
- `run()`: A main game loop that controls the game flow between human and computer players.

Implementation Tasks:

- Implement the `'TicTacToe'` class according to the specified definition, ensuring all attributes and methods are correctly defined and implemented (use the provided notebook).

Task 2:

The `MiniMaxAgent` class represents an AI agent that utilizes the minimax algorithm to play optimally against a human player in a game of `Tic-Tac-Toe`. This class is responsible for evaluating potential game states, determining the best move to make, and executing that move on behalf of the computer player. The MiniMaxAgent class is defined with an attribute game, which refers to the current game environment and state.

Implementation Tasks:

- MiniMaxAgent Class Implementation:
  - Define the `MiniMaxAgent` class with the required `__init__` method to initialize the agent with the game environment (game).
  - Implement the `'minimax(state, depth, player)'` method to recursively compute the best move using the `minimax algorithm`, considering the current game state (state), depth of recursion (depth), and current player (player). This method returns a list containing the best row, best column, and corresponding score for the best move.

  - Implement the `play()` method to execute the `minimax algorithm` and make a move for the computer player (comp), while also handling the selection of a random move at the start of the game. If the game depth is at the maximum (indicating the start of the game), a random move is chosen instead.

- Testing the Implementation:
  - Create a game object and run it.
  - Can you beat the minimax agent ?