EAST WEST UNIVERSITY

CSE303: Statistics for Data Science

[Spring 2023]

Project Report

Course Code : CSE-366
Course Title : Artificial Intelligence
Section : 03
Project Name : Tic-Tac-Toe

Submitted by:

Student ID	Student Name
2019-2-60-048	Md Wahiduzzaman
2020-2-60-164	Rafia Akter Tuly
2018-3-60-109	Md Rabby Rayhan

Introduction:

We choose tic-tac-toe game in our mini project because virtually everyone knows the game and the rule are very simple enough to understand. We don't need an elaborate analysis of game configurations. Basically, we would like a computer player to 'look 'at the game board and decide where to move. There are many ways to do this but in our mini project we will a explore a common method of searching (Minimax Algorithms) for moves by building a game tree the represent the possible moves and selecting a move the best outcome.

Problem statement:

The problems statement of our mini project we faced is that –

01.The problem that the code solves is a single-player Tic-Tac-Toe game where the player plays against the computer.

02.The game is played on a square board of size BOARD_SIZE, where the player plays as 'PLAYER' and the computer plays as 'COMPUTER'.

03.The board is represented as a 2D array, where each element can be either EMPTY, PLAYER, or COMPUTER.

04.The Game class handles the game activities and interactions between the player and the computer.

05. The play function is the main function that starts the game. It creates an instance of the Game class, and then starts the game loop. In each iteration of the loop, the player is prompted to make a move. If the move is valid, the player's move is made on the game board. Then, the computer makes its move on the game board. If either the player or the computer wins or the game ends in a tie, the game loop ends, and the winner is announced. If the game does not end, the loop continues until a winner is declared or the game ends in a tie.

06.The getBestMove function uses the minimax algorithm with alpha-beta pruning to determine the best move for the computer.

07.The evaluateBoard function calculates a score for the current board state based on the number of player and computer pieces in a row, column, or diagonal.

08.Alpha-beta pruning is used to prune branches of the game tree that are guaranteed to be worse than previously evaluated branches. The minimax function recursively generates all possible moves and determines the best move for the current player using the minimax algorithm. The alphaBeta function uses alpha-beta pruning to optimize the minimax algorithm by pruning branches that are guaranteed to be worse than previously evaluated branches.

09. The checkGameState function checks the game board for a winner or tie. It checks each row, column, and diagonal for a win. If there is a win, the function returns the winner. If there is no win and the game board is full, the function returns a tie. If there is no win and the game board is not full, the function returns that the game is not over.

Formal algorithm:

Here are the algorithms for the code:

1.**NoneEmptyPosition** Exception Class:

• The **NoneEmptyPosition** class is a custom exception class that is raised when a player to make a move on a rubric that is already taken.

2. **OutOfRange** Exception Class:

• The **OutOfRange** class is another custom exception class that is raised when a tries to make a move outside the range of the board.

3. **GameState Enum** Class:

- The class has four possible states: "**tie**", "**notEnd**", "**o**", and "**x**".
- The "**tie**" state represents a tie game,
- the "**notEnd**" state represents a game that has not ended yet.
- the "**o**" state represents a win for the computer
- the "**x**" state represents a win for the player

4. **Board Class:**

- **init(self, size):** Initializes the Board object with a given size and creates an empty board.
- **print(self):** Prints the current state of the board.
- **getBoardPosition(self, position):** Given a position on the board, returns the row and column of that position.
- **getLastMove(self):** Returns the last move made on the board.
- **getRow(self, row):** Returns the rubrics in a given row of the board.
- **getColumn(self, column):** Returns the rubrics in a given column of the board.
- **getDiagonal(self):** Returns a list of the two diagonals of the board.
- **checkIfOnMainDiagonal(self, position):** Given a position on the board, returns True if the position is on the main diagonal, False otherwise.

- **checkIfOnSecondaryDiagonal(self, position):** Given a position on the board, returns True if the position is on the secondary diagonal, False otherwise.
- **drawX(self, position):** Given a position on the board, draws an X on that position.
- **drawEmpty(self, position):** Given a position on the board, draws an empty space on that position.
- **drawO(self, position):** Given a position on the board, draws an O on that position.
- **checkIfRubricEmpty(self, position):** Given a position on the board, returns True if the position is empty, False otherwise.
- **all_same(self, listToBeChecked, char):** Given a list of rubrics and a character ('X' or 'O'), returns True if all the rubrics in the list are the same as the character, False otherwise.

5. Game Class:

- **init(self, numberOfPlayers, boardSize):** Initializes the Game object with a given number of players and board size, creates a Board object, and sets the turn to either the player or the computer.
- **coinFlip(self):** Randomly decides which player goes first.
- **getPlayersNames(self):** Gets the names of the players from the user.
- **getPlayerMove(self):** Gets a move from the player.
- **checkForWin(self, turn):** Given the current turn, checks if the player has won the game.
- **checkForTie(self):** Checks if the game is a tie.
- **genrate(self):** Generates all possible moves on the board.
- **checkGameState(self):** Checks the current state of the game (win, tie, or not ended).
- **start(self):** Starts the game and manages the turns until the game is over.
- **iterativeDeepSearch(self):** Performs the Minimax algorithm with alpha-beta pruning to find the best move for the computer.
- **minmax2(self, depth, isMax, alpha, beta, startTime, timeLimit):** Recursive function that performs the Minimax algorithm with alpha-beta pruning.
- **calculateLine(self, line):** Given a line of rubrics on the board, calculates the number of X's, O's, and empty spaces in the line.
- **getScoreLine(self, line):** Given a line of rubrics on the board, calculates the score of that line.
- **evaluate(self):** Evaluates the current state of the board and returns the score.

Implementation of Code:

In our Tic-tac-toe game, we implemented the Minimax algorithm with alpha-beta pruning which is a well-known algorithm for finding the best move in a game with perfect information for our opponent AI player.

The code is divided into two classes: Board and Game.

The Board class represents the game board and contains functions for printing the board, getting the position of a move on the board, getting the last move made, getting a row or column of the board, getting the diagonals of the board, checking if a position is on the main or secondary diagonal, drawing X, O, or an empty space on the board, and checking if a position on the board is empty.

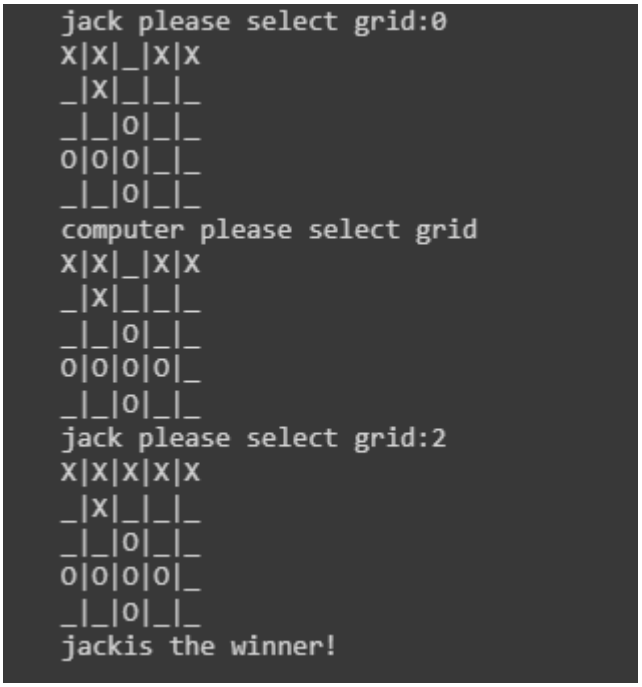
The Game class represents the game itself and contains functions for getting the names of the players, getting a move from the player or the AI opponent, checking for a win or tie, generating all possible moves, checking the game state, starting the game, and performing the Minimax algorithm with alpha-beta pruning to find the best move for the AI opponent.

The start() function in the Game class is the main function that runs the game. It first gets the names of the players, then alternates between getting a move from the player and the AI opponent until the game is over. It checks the game state after each move to see if there is a winner or a tie. If the game is not over, it switches the turn to the other player.

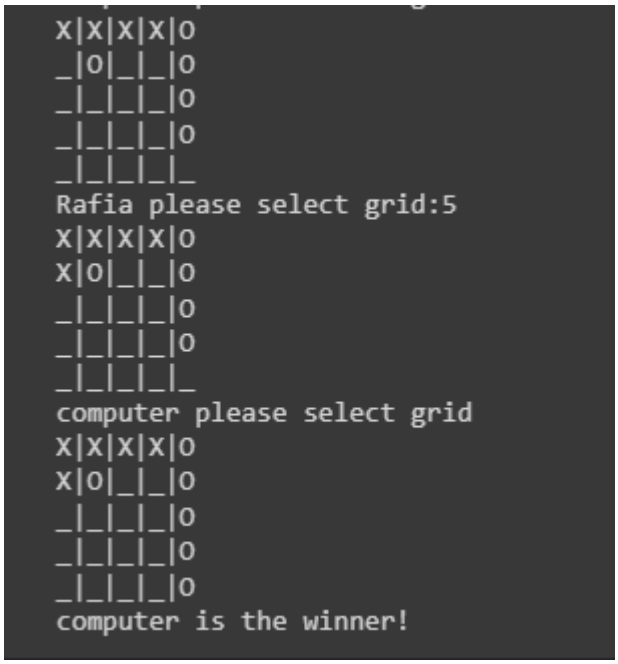
The Minimax algorithm with alpha-beta pruning is used to find the best move for the AI opponent. The iterativeDeepSearch() function in the Game class performs the Minimax algorithm with alpha-beta pruning to find the best move in a given time limit. The minmax2() function is the recursive function that performs the Minimax algorithm with alpha-beta pruning. It takes in the depth of the game tree, whether it is the maximizer or the minimizer's turn, the alpha and beta values, the start time, and the time limit. It generates all possible moves, evaluates the score of each move, and recursively calls itself with the new board state and updated alpha and beta values. It returns the best score and position found.

The evaluate() function in the Game class evaluates the score of the current board state. It calculates the score of each row, column, and diagonal of the board and returns the sum of all scores. The getScoreLine() function calculates the score of a single row, column, or diagonal. It counts the number of X's, O's, and empty spaces in the line and returns a score based on the number of X's and O's.

Result of the Games:



Player Wins the game



Computer Wins the game

Learnings during implementing the idea:

While we were implementing the code for the game, we came to learn about the some of the listed observations:

- Object-oriented programming using classes and objects.
- Implementation of a game using loops, conditionals, and functions.
- Use of error handling to handle exceptions and input validation.
- Implementation of the iterative deepening search algorithm with alpha-beta pruning to find the best move in a game.
- Use of datetime module to measure time elapsed during the search.
- Implementation of a scoring function to evaluate the game board.
- Use of list comprehension to extract rows and columns from the game board.
- Use of tuples to return multiple values from a function.
- Use of enums to represent game states.
- Use of random module to randomly select the starting player.

Conclusion:

We implemented a Tic Tac Toe game code with a computer player using the iterative deepening search algorithm which called the Minimax algorithm with alpha-beta pruning. The game can be played with two players, here one of the players is an AI, and the board size can be customized. We also include functions for checking the game state, getting player moves, and evaluating the game board.