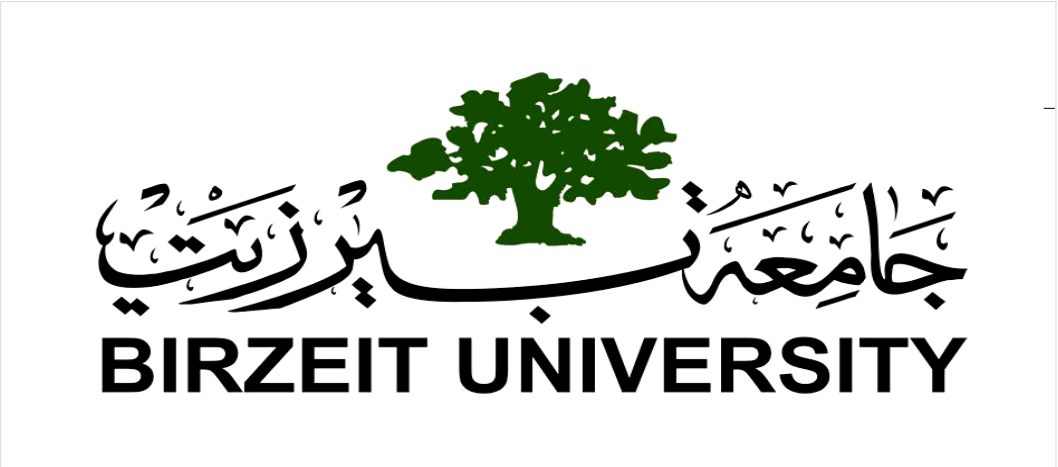

Faculty of Engineering & Technology

Electrical & Computer Engineering Department

ENCS3340

Project 1 Report

Magnetic Cave

Prepared by:

Saja Shareef	**1200901**
Shereen Ibdah	**1200373**

Instructor: Dr. Yazan Abu Farha

Date: 20/6/2023

About The Project:

Our program is an implementation of the game "Magnetic Cave" using Python. It allows two players to take turns making moves on an 8x8 board. The goal of the game is to create a sequence of five consecutive markers (■ or □) either horizontally, vertically, or diagonally.

Our program's structure and main functions:

- **Class square:**
    1. Represents a square on the game board.
    2. Each square has a value ('E' for empty, '■', or '□') and a private attribute __validity to determine if it's a valid move.
    3. Provides getter and setter methods for __validity.
- **Game board:**
    1. Created as a 2D list of `square` objects, initialized with empty squares.
    2. The board is represented by the `board` variable.
- **`print board(board)` function:**
    1. Prints the current state of the game board.
- **`isEmptySquare(board, x, y)` function:**
    1. Checks if a square on the board is empty.
    2. Returns `True` if the square is empty, `False` otherwise.
- **`initialState(board)` function:**
    1. Sets the validity of squares in columns 0 and 7 to `True`.

- **`isValidMove(board, x, y)` function:**
    1. Checks if a move is valid based on the square's validity attribute.
    2. Returns `True` if the move is valid, `False` otherwise.

- **`playGame()` function:**
    1. The main function that runs the game loop.
    2. Displays the initial board and prompts the players for their moves.
    3. Checks move validity and updates the board accordingly.
    4. Checks for a winner or a draw condition.
    5. Implements an AI opponent that uses the Minimax algorithm to make automatic moves.

- **`menu()` function:**
    1. Displays the game mode options.

- **`togglePlayer(currentPlayer)` function:**
    1. Switches the current player between '■' and '□'.

- **Winner checking functions:**
    1. `checkWinner(board, currentPlayer)`: Checks if the current player has won the game.
    2. `checkFiveInRow(board, currentPlayer)`: Checks for five markers in a row.

3. `checkFiveInColumn(board, currentPlayer)`: Checks for five markers in a column.
4. `checkFiveInDiagonal(board, currentPlayer)`: Checks for five markers in a diagonal.

- **`nearness_to_victory(board, currentPlayer)` function:**
    1. Determines the nearness to victory for the current player.
    2. Checks for the maximum count of consecutive markers in rows, columns, and diagonals.

- **Minimax algorithm functions:**
    1. `minimax(board, depth, isMaximizingPlayer, alpha, beta, currentPlayer)`: Implements the Minimax algorithm for the AI opponent.
    2. Evaluates the score of a move and selects the best move based on the current player and depth.
    3. Uses alpha-beta pruning to optimize the algorithm.

Our program provides three game modes: manual entry for both players, manual entry for one player with automatic moves for the other, and automatic moves for both players.

Heuristic Function: nearness_to_victory:

The heuristic function nearness_to_victory is used to guide the AI's choice of move, by predicting which moves will bring the AI closer to forming such a sequence.

The nearness_to_victory function takes two parameters: board, which is an 8x8 list of lists representing the game state; and currentPlayer, which is a marker representing the current player (either ■ or □).

The function works by iterating over the rows, columns, and diagonals of the board, counting the maximum number of consecutive blocks of the current player it finds. It returns this maximum count, which can be used to determine the value of a given board state: the higher the count, the more advantageous the state is for the current player.

To perform such an analysis in our function we first iterate over all rows and columns while counting the maximum number of consecutive current player blocks found in such directions by checking each cell located in both groups. Upon finding a continuous trail we update our current best count variable (initially at zero) whenever one is larger than its previous count.

We further find diagonal sets containing continuous player blocks by iterating over all possible starting points utilizing both top left to bottom right and top right to bottom left movements through each cell on these diagonals.

Once all these elements are done with, our function enables a report back with the maximum count found within all said groups of movements allowing insight into how much closer our current player is to victory.

In summary, our game-playing algorithm displayed an impressive in the tournament, securing numerous victories through its strategic evaluation, efficient move selection, and adaptability. While some losses occurred, they shed light on the need to address opponent strategies, search depth limitations, and execution time constraints. To improve future tournament outcomes, we recommend integrating advanced heuristics, employing adaptive search depth mechanisms, exploring parallelization techniques, and considering the application of machine learning. These enhancements aim to enhance the algorithm's decision-making abilities and competitiveness, enabling it to excel in upcoming competitions.

////

"We certify that this submission is the original work of members of the group and meets the Faculty's Expectations of Originality",