Faculty of Engineering and Technology

ENCS3340

Project 1 Report

Magnetic Cave

Prepared by:

Zainab Jaradat,1201766

Manar Shawahni, 1201086

Instructor: Yazan Abu Farha.

Section: 4

Date: 20/6/2023

• Program Implementation:

The program is implemented using Python and a GUI library such as Tkinter or Pygame. Here's a more detailed breakdown of the main functions and data structures:

- **Create_board():** This function creates an empty game board as a list of lists. Each cell of the board is represented by a symbol ('■' or '□') or is empty (' ').

- **update_cell(row, col):** This function handles the logic for updating a cell on the game board when a player makes a move. It takes the row and column indices as parameters and updates the cell with the current player's symbol. It also checks for a win condition by calling **check_win()** and for a tie condition by calling is_board_full(). Additionally, it updates the current player.

- **disable_buttons():** This function disables all the buttons on the GUI when the game ends to prevent further moves.

- **print_board(board):** This function displays the current state of the game board on the console in a text-based format. It is useful for debugging and for displaying the board in the absence of a GUI.

- **check_win(board, player):** This function checks if a player has won the game by examining the rows, columns, and diagonals of the board. It counts the number of consecutive symbols belonging to the player and checks if it reaches the winning condition (e.g., five in a row). It returns True if a win is detected and False otherwise.

- **evaluate_board(board):** This function evaluates the current state of the board for the AI player. It assigns a score to different configurations of symbols based on their potential for winning. The higher the score, the more favorable the board is for the AI player.

- **evaluate_window(window, player):** This function evaluates a window of five consecutive cells to determine its score based on the number of symbols belonging to the player and the opponent. It assigns higher scores to windows that have more symbols belonging to the AI player and fewer symbols belonging to the opponent.

- **find_best_move(board, player):** This function implements the minimax algorithm with alpha-beta pruning to find the best move for the AI player. It evaluates different possible moves by calling minimax() and selects the move with the highest score.

- **minimax(board, depth, maximizing_player, alpha, beta, player):** This function performs the minimax algorithm with alpha-beta pruning to evaluate different game states and calculate the scores for different moves. It recursively explores the game tree by simulating moves for both the AI player and the opponent and selects the move that maximizes the AI player's score and minimizes the opponent's score.

- **is_board_full(board):** This function checks if the game board is full, indicating a tie. It returns True if the board is full and False otherwise.

- **get_move(player):** This function prompts the player to enter the row and column for their move through the console. It takes the player as a parameter and returns the row and column indices entered by the player.

• How the Program Runs:

To run the program, you execute the **create_gui()** function, which sets up the graphical user interface (GUI) for the game. The GUI consists of a game board displayed as a grid of buttons. Program provides an interactive game-playing experience with a GUI, allowing players to enjoy the game and play against each other or against an AI opponent.

The **heuristic** used in the implementation of Magnetic Cave is a simple evaluation function that assigns scores to different board configurations based on the presence and arrangement of bricks. The heuristic evaluates the board from the perspective of the AI player (■).

The evaluation function calculates separate scores for both the AI player and the opponent (□). The score is increased based on the number of consecutive bricks in a row, column, or diagonal. The evaluation assigns higher scores to configurations that are closer to winning (e.g., having 5 consecutive bricks) and lower scores to configurations that are less favorable.

Here is the breakdown of the scores assigned by the heuristic function:

- 1000: If the AI player has 5 consecutive bricks, it is a winning configuration. This score ensures that the AI player always prioritizes completing its bridge if it has the opportunity.

- 30: If the AI player has 4 consecutive bricks and the opponent has empty space, it is a strong position. The AI player is close to winning and should aim to complete the bridge.

- 10: If the AI player has 3 consecutive bricks and the opponent has empty space, it is a strong position. The AI player is a little bit far from winning and should aim to complete the bridge.

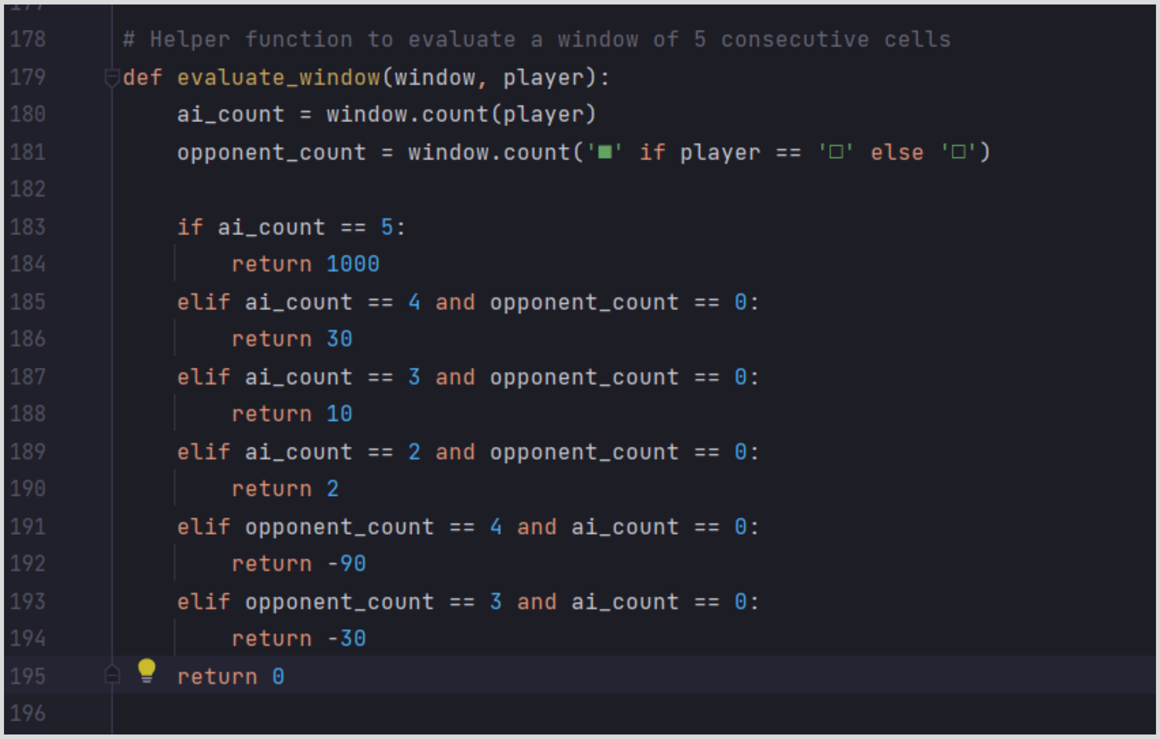
- 2: If the AI player has 2 consecutive bricks and three empty spaces (the opponent), it is a relatively neutral position. The AI player needs further moves to establish a stronger position.

- "-90": If the opponent has four consecutive bricks, so the opponent has a **big chance** to win. The AI player must put the fifth one to detect the opponent player from winning.

- “-30”: If the opponent has three consecutive bricks, so the opponent has a **chance** to win. The AI player must put the fourth one to reduce the opponent player chance to win.

The heuristic function calculates the total score by summing up the scores from different rows, columns, and diagonals on the board. By evaluating the board in this manner, the AI player can prioritize moves that lead to more favorable configurations and increase the chances of winning.

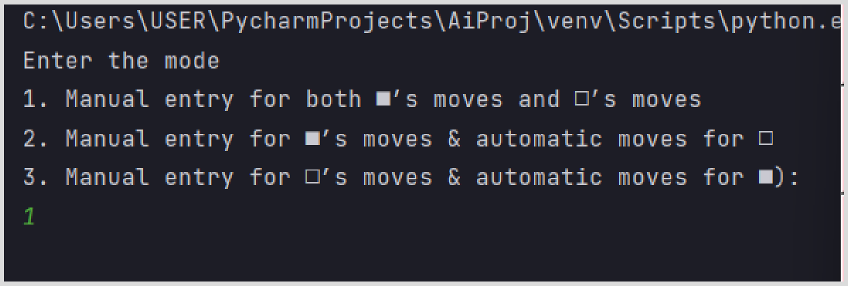
While the provided heuristic is relatively simple, it captures the basic logic of the game and allows the AI player to make informed decisions.



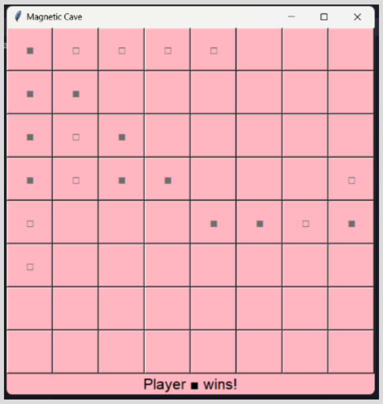
```python
# Helper function to evaluate a window of 5 consecutive cells
def evaluate_window(window, player):
    ai_count = window.count(player)
    opponent_count = window.count('■' if player == '□' else '□')

    if ai_count == 5:
        return 1000
    elif ai_count == 4 and opponent_count == 0:
        return 30
    elif ai_count == 3 and opponent_count == 0:
        return 10
    elif ai_count == 2 and opponent_count == 0:
        return 2
    elif opponent_count == 4 and ai_count == 0:
        return -90
    elif opponent_count == 3 and ai_count == 0:
        return -30
    return 0
```

Testing:

In mode 1:

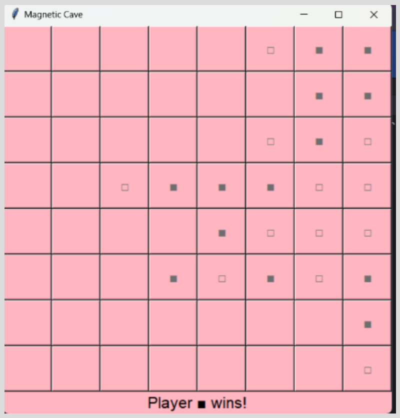


```
C:\Users\USER\PycharmProjects\AiProj\venv\Scripts\python.e
Enter the mode
1. Manual entry for both ■'s moves and □'s moves
2. Manual entry for ■'s moves & automatic moves for □
3. Manual entry for □'s moves & automatic moves for ■):
1
```



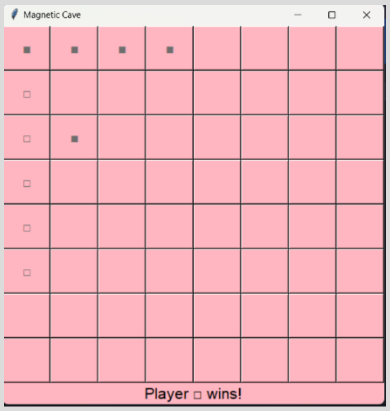
Player ■ wins!

Diagonal (L-to-R)



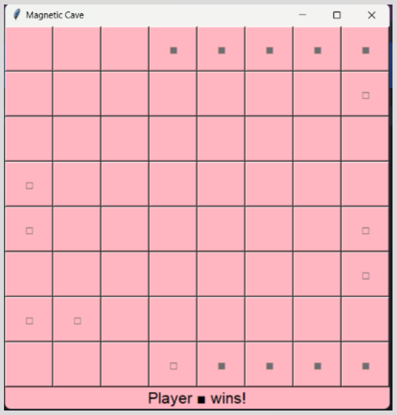
Player ■ wins!

Diagonal (R-to-L)



Player □ wins!

Column



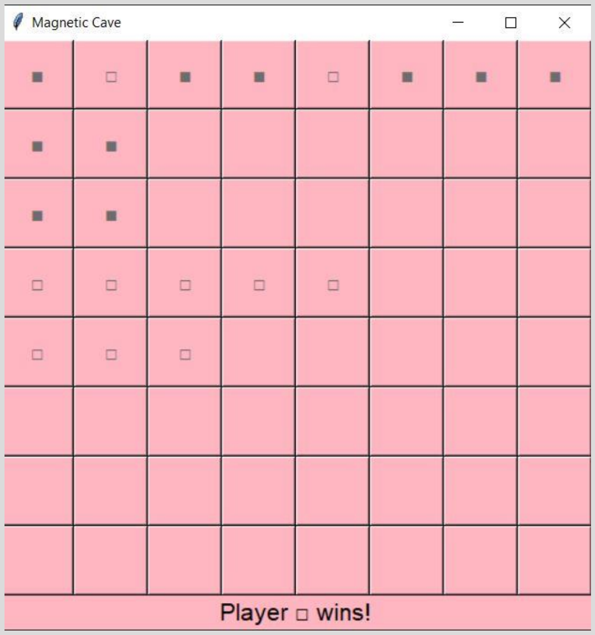
Player ■ wins!

Row

In mode 2:

When AI player is '□':

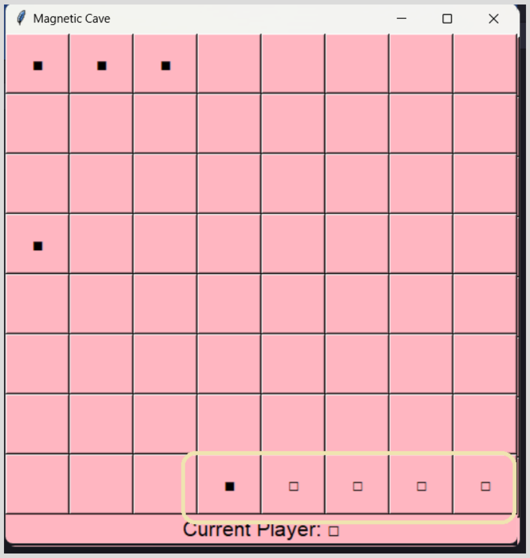
As shown below **AI Player won!**



Player □ wins!

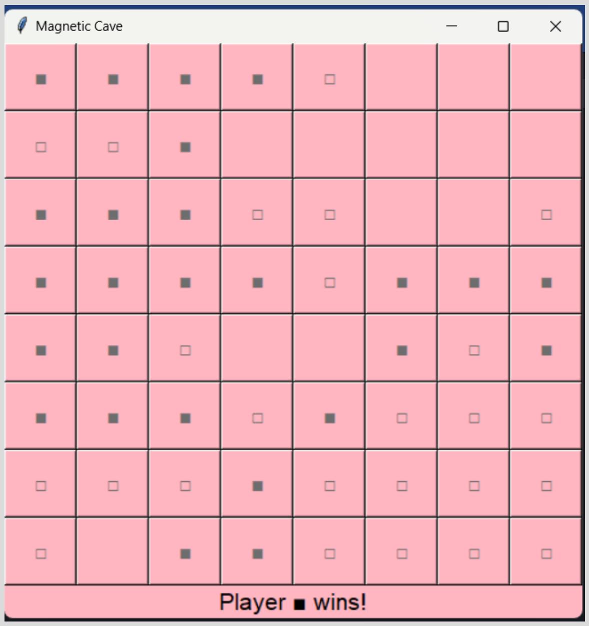
In mode 3:

Here AI Player trying to stop me from winning:



Current Player: □

Here **AI Player won**:



Player ■ wins!

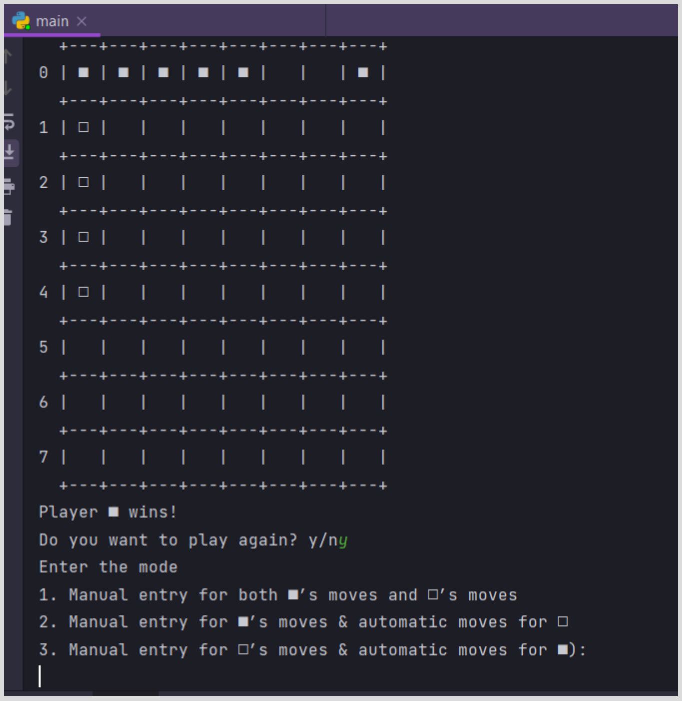
Why AI Player Wins?

The AI player wins against a manual player due to its advanced techniques and computational advantages. It uses the minimax algorithm with alpha-beta pruning, which evaluates the game state and makes optimal moves. An evaluation function assigns scores to board configurations, helping the AI player identify winning opportunities. The depth-limited search focuses on immediate moves, leading to faster decision-making. Alpha-beta pruning further enhances efficiency by cutting off unnecessary branches. The AI player benefits from the computer's processing power, allowing it to analyze the game state effectively. These factors combine to give the AI player a significant advantage, resulting in more wins against human opponents.

When a player won, we print

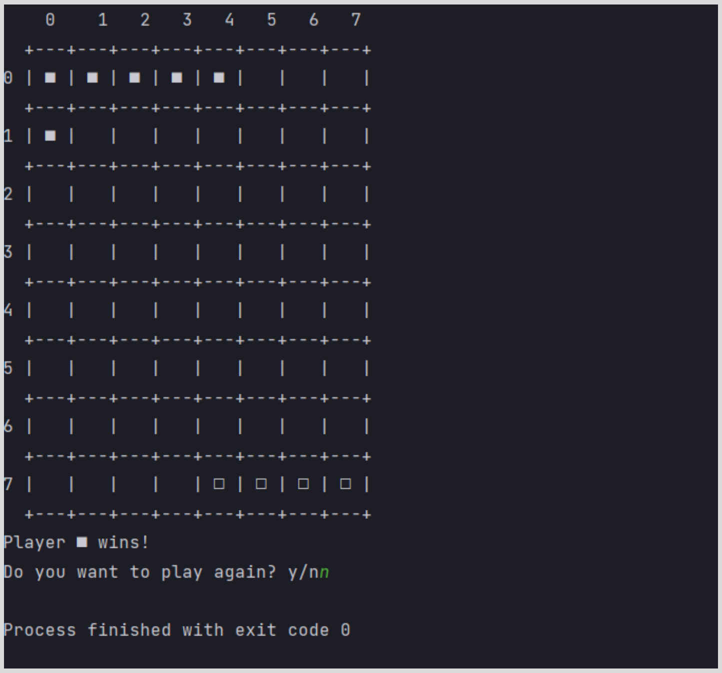
” Do you want to play again? y/n”

If “**y**”, select other mode:



```
   +---+---+---+---+---+---+---+---+
 0 | ■ | ■ | ■ | ■ | ■ |   |   | ■ |
   +---+---+---+---+---+---+---+---+
 1 | □ |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
 2 | □ |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
 3 | □ |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
 4 | □ |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
 5 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
 6 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
 7 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
Player ■ wins!
Do you want to play again? y/ny
Enter the mode
1. Manual entry for both ■'s moves and □'s moves
2. Manual entry for ■'s moves & automatic moves for □
3. Manual entry for □'s moves & automatic moves for ■):
```

If “n”, **exit** program:



```
    0   1   2   3   4   5   6   7
  +---+---+---+---+---+---+---+---+
0 | ■ | ■ | ■ | ■ | ■ |   |   |   |
  +---+---+---+---+---+---+---+---+
1 | ■ |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
2 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
4 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
5 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
6 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
7 |   |   |   |   | □ | □ | □ | □ |
  +---+---+---+---+---+---+---+---+
Player ■ wins!
Do you want to play again? y/nn

Process finished with exit code 0
```

Conclusion:

⚠-We noticed that AI Player works well and is more defensive than offensive as it prevents my chances of winning, and tries to win at the same time.

⚠-However more advanced heuristics could be developed by considering additional factors such as strategic positioning, offensive moves, and anticipating the opponent's moves.