

**Faculty of Engineering & Technology**

**Electrical & Computer Engineering Department**

**ENCS3340, ARTIFICIAL INTELLIGENCE**

**Prepared by:**

Majeed Issa 1201193

Ayman Qashoo 1200312

**Instructor:**

Dr. Yazan Abu Farha

**Section:** 4

**Date:** 19/6/2023

**Note important:** *We certify that this submission is the original work of members of the group and meets the Faculty's Expectations of Originality*

Contents

[**1.Introduction 🡪** 3](#_Toc138064149)

[**2.Program Description🡪** 3](#_Toc138064150)

[2.manual\_vs\_manual(): 4](#_Toc138064151)

[3.manual\_vs\_ computer 4](#_Toc138064152)

[4. computer\_vs\_manual(depth): 4](#_Toc138064153)

[5.make\_move(player, row, col): 4](#_Toc138064154)

[6.is\_game\_over(): 4](#_Toc138064155)

[7.is\_winner(player): 4](#_Toc138064156)

[8.find\_best\_move(player, depth): 4](#_Toc138064157)

[9.get\_computer\_move(player, depth): 4](#_Toc138064158)

[10.get\_player\_move(player): 4](#_Toc138064159)

[11.print\_board(): 4](#_Toc138064160)

[**3. Heuristic Description and Justification🡪** 4](#_Toc138064161)

[**4. Tournament Results and Analysis🡪** 4](#_Toc138064162)

[**5. Conclusion🡪** 5](#_Toc138064163)

[**6.code🡪** 5](#_Toc138064164)

# **1.Introduction 🡪**

The Magnetic Cave is a strategic two-player board game in which players take turns placing magnets on a 5x5 grid. The objective of the game is to form a line of three magnets in a row, column, or diagonal. In this report, we will discuss the implementation of the Magnetic Cave game, including how to run it, the main functions and data structures used, the heuristic employed, and the results of a tournament played against an opponent.

# **2.Program Description🡪**

To run the Magnetic Cave game, execute the program using a Python interpreter. The main function, select\_play\_mode (), prompts the user to select a play mode: Manual vs. Manual, Manual vs. computer, or computer vs. Manual. After selecting a mode, the game begins, and players take turns placing magnets on the grid until a winning condition is met. The game board is displayed after each move, and the game's outcome is announced at the end.

The main functions and data structures used in the program are as follows:

1.select\_play\_mode (): Allows the user to select the play mode and starts the corresponding game.

2.manual\_vs\_manual (): Implements the Manual vs. Manual game mode.

3.manual\_vs\_ computer (depth): Implements the Manual vs. computer game mode.

4. computer \_vs\_ manual (depth): Implements the computer vs. Manual game mode.

5.make\_move (player, row, col): Places a magnet on the grid for the specified player at the given row and column.

6.is\_game\_over (): Checks if the game is over by evaluating the winning condition.

7.is\_winner (player): Determines if the specified player has won the game.

8.find\_best\_move (player, depth): Uses a minimax algorithm with alpha-beta pruning to find the best move for the computer player.

9.get\_computer\_move (player, depth): Gets the move from the computer player, considering the specified player's role and the depth of the search.

10.get\_player\_move (player): Prompts the player to enter their move and returns the selected row and column.

11.print\_board (): Displays the current state of the game board.

The main data structure used is a 5x5 grid to represent the game board. Each cell in the grid can hold either 'X' (player 1) or 'O' (player 2) to represent the magnets placed by the players.

# **3. Heuristic Description and Justification🡪**

The heuristic employed in the Magnetic Cave game is a combination of the minimax algorithm and alpha-beta pruning. The minimax algorithm is a well-known approach for adversarial games, aiming to maximize the computer player's chances of winning and minimize the opponent's chances. It achieves this by evaluating the game state at different levels of depth and selecting the move that leads to the most favorable outcome.

Alpha-beta pruning is a technique used to improve the efficiency of the minimax algorithm. It eliminates branches in the game tree that do not need to be explored further, significantly reducing the number of game states evaluated. By pruning branches that are known to lead to suboptimal outcomes, the algorithm can focus on promising paths, resulting in faster and more accurate decision-making.

The combination of the minimax algorithm and alpha-beta pruning provides an effective heuristic for the computer player in the Magnetic Cave game. It allows the computer to consider multiple possible moves, anticipate the opponent's moves, and select the move that maximizes its chances of winning.

# **4. Tournament Results and Analysis🡪**

During the tournament, the program competed against opponents in various game modes, including Manual vs. Manual, Manual vs. computer, and computer vs. Manual. The tournament aimed to evaluate the program's performance and assess its strengths and weaknesses.

In the Manual vs. Manual mode, the program performed well, as both players made moves based on manual input. The game outcome in this mode largely depended on the players' strategic decisions, and the program achieved satisfactory results.

In the Manual vs. computer mode, the program faced challenges in defeating the Manual opponent. The computer player's performance relied on its ability to search for the best moves and anticipate the Manual player's strategies. However, the computer's performance might have been affected by limitations in the heuristic, such as the depth of the search. In some cases, the computer failed to exploit winning opportunities, allowing the Manual player to win.

In the computer vs. Manual mode, the program exhibited strong performance. The computer player's ability to evaluate game states and select optimal moves gave it a strategic advantage over the Manual player. The program consistently made winning moves and outperformed the Manual opponent.

To improve the program's performance in future tournaments, several enhancements can be considered. Increasing the depth of the search in the minimax algorithm can allow the computer player to explore more game states and make more informed decisions. Additionally, refining the evaluation function used to assess the desirability of game states can further improve the computer player's decision-making.

# **5. Conclusion🡪**

The Magnetic Cave game provides an engaging and strategic experience for two players. The program's implementation successfully captures the game's rules and mechanics, allowing users to enjoy different play modes against Manual or computer opponents. The heuristic employed combines the minimax algorithm and alpha-beta pruning, enabling the computer player to make intelligent and strategic moves.

Throughout the tournament, the program demonstrated competitive performance, achieving satisfactory results against human opponents and showcasing the strength of the computer player. However, there is room for improvement in exploiting winning opportunities and enhancing the evaluation function to increase the program's chances of success.

In conclusion, the Magnetic Cave game and its accompanying program provide an enjoyable gaming experience while showcasing the power of computer algorithms in strategic decision-making.

# **6.code🡪**

import sys  
import time  
  
# Constants  
EMPTY = "\_"  
PLAYER1 = "■"  
PLAYER2 = "□"  
WINNING\_LENGTH = 5  
BOARD\_SIZE = 8  
  
# Create an empty game board  
board = []  
for \_ in range(BOARD\_SIZE):  
 row = [EMPTY] \* BOARD\_SIZE  
 board.append(row)  
  
# Function to print the game board  
def print\_board():  
 print(" " + " ".join([chr(ord('a') + i) for i in range(BOARD\_SIZE)]))  
 for i, row in enumerate(board):  
 print(str(i + 1) + " " + " ".join(row) + " " + str(i + 1))  
 print(" " + " ".join([chr(ord('a') + i) for i in range(BOARD\_SIZE)]))  
  
# Function to check if a move is valid  
def is\_valid\_move(row, col):  
 if 0 <= row < BOARD\_SIZE and 0 <= col < BOARD\_SIZE:  
 if (col == 0 or board[row][col-1] != EMPTY) or (col == BOARD\_SIZE-1 or board[row][col+1] != EMPTY):  
 return board[row][col] == EMPTY  
 return False  
  
# Function to check if the game has ended  
def is\_game\_over():  
 return is\_winner(PLAYER1) or is\_winner(PLAYER2) or is\_board\_full()  
  
# Function to check if the board is full  
def is\_board\_full():  
 for row in board:  
 if EMPTY in row:  
 return False  
 return True  
  
# Function to check if a player has won  
def is\_winner(player):  
 # Check rows  
 for row in board:  
 count = 0  
 for cell in row:  
 if cell == player:  
 count += 1  
 if count == WINNING\_LENGTH:  
 return True  
 else:  
 count = 0  
  
 # Check columns  
 for col in range(BOARD\_SIZE):  
 count = 0  
 for row in range(BOARD\_SIZE):  
 if board[row][col] == player:  
 count += 1  
 if count == WINNING\_LENGTH:  
 return True  
 else:  
 count = 0  
  
 # Check diagonals  
 for row in range(BOARD\_SIZE):  
 for col in range(BOARD\_SIZE):  
 count = 0  
 for i in range(WINNING\_LENGTH):  
 if row + i < BOARD\_SIZE and col + i < BOARD\_SIZE:  
 if board[row + i][col + i] == player:  
 count += 1  
 if count == WINNING\_LENGTH:  
 return True  
 else:  
 count = 0  
  
 count = 0  
 for i in range(WINNING\_LENGTH):  
 if row + i < BOARD\_SIZE and col - i >= 0:  
 if board[row + i][col - i] == player:  
 count += 1  
 if count == WINNING\_LENGTH:  
 return True  
 else:  
 count = 0  
  
 return False  
  
# Function to make a move  
def make\_move(player, row, col):  
 board[row][col] = player  
  
# Function to undo a move  
def undo\_move(row, col):  
 board[row][col] = EMPTY  
  
# Minimax algorithm with alpha-beta pruning  
def minimax(player, depth, alpha, beta):  
 if depth == 0 or is\_game\_over():  
 return evaluate()  
  
 if player == PLAYER1: # Maximizer's turn  
 max\_eval = float('-inf')  
 for row in range(BOARD\_SIZE):  
 for col in range(BOARD\_SIZE):  
 if is\_valid\_move(row, col):  
 make\_move(PLAYER1, row, col)  
 eval = minimax(PLAYER2, depth - 1, alpha, beta)  
 undo\_move(row, col)  
 max\_eval = max(max\_eval, eval)  
 alpha = max(alpha, eval)  
 if beta <= alpha:  
 break # Beta cut-off  
 return max\_eval  
 else: # Minimizer's turn  
 min\_eval = float('inf')  
 for row in range(BOARD\_SIZE):  
 for col in range(BOARD\_SIZE):  
 if is\_valid\_move(row, col):  
 make\_move(PLAYER2, row, col)  
 eval = minimax(PLAYER1, depth - 1, alpha, beta)  
 undo\_move(row, col)  
 min\_eval = min(min\_eval, eval)  
 beta = min(beta, eval)  
 if beta <= alpha:  
 break # Alpha cut-off  
 return min\_eval  
def get\_possible\_moves():  
 moves = []  
 for row in range(BOARD\_SIZE):  
 for col in range(BOARD\_SIZE):  
 if is\_valid\_move(row, col):  
 moves.append((row, col))  
 return moves  
  
# Recursive function to find the best move using minimax algorithm with alpha-beta pruning  
def find\_best\_move(player, depth, alpha=float('-inf'), beta=float('inf')):  
 best\_eval = float('-inf') if player == PLAYER1 else float('inf')  
 best\_move = None  
  
 if depth == 0 or is\_game\_over():  
 return evaluate(), best\_move  
  
 possible\_moves = get\_possible\_moves()  
  
 for move in possible\_moves:  
 make\_move(player, move[0], move[1])  
 eval = find\_best\_move(PLAYER2 if player == PLAYER1 else PLAYER1, depth - 1, alpha, beta)[0]  
 undo\_move(move[0], move[1])  
  
 if player == PLAYER1 and eval > best\_eval:  
 best\_eval = eval  
 best\_move = move  
 alpha = max(alpha, best\_eval)  
 elif player == PLAYER2 and eval < best\_eval:  
 best\_eval = eval  
 best\_move = move  
 beta = min(beta, best\_eval)  
  
 if beta <= alpha:  
 break  
  
 return best\_eval, best\_move  
  
# Function to get the move from the player  
def get\_player\_move(player):  
 while True:  
 move = input("Enter your move (row and column): ")  
 if len(move) == 2 and move[0].isdigit() and move[1].isalpha():  
 row = int(move[0]) - 1  
 col = ord(move[1].lower()) - ord('a')  
 if is\_valid\_move(row, col):  
 return row, col  
 print("Invalid move. Try again.")  
  
# Function to get the move from the AI player  
def get\_computer\_move(player, depth):  
 print(player + "'s turn")  
 start\_time = time.time()  
 best\_move = find\_best\_move(player, depth)[1]  
 end\_time = time.time()  
 print("computer move found in {:.3f} seconds".format(end\_time - start\_time))  
 return best\_move  
  
# Function to evaluate the current state of the board  
def evaluate():  
 score = 0  
 # Evaluate rows  
 for row in board:  
 for i in range(BOARD\_SIZE - WINNING\_LENGTH + 1):  
 window = row[i:i+WINNING\_LENGTH]  
 score += evaluate\_window(window)  
  
 # Evaluate columns  
 for col in range(BOARD\_SIZE):  
 for i in range(BOARD\_SIZE - WINNING\_LENGTH + 1):  
 window = [board[j][col] for j in range(i, i+WINNING\_LENGTH)]  
 score += evaluate\_window(window)  
  
 # Evaluate diagonals (positive slope)  
 for i in range(BOARD\_SIZE - WINNING\_LENGTH + 1):  
 for j in range(BOARD\_SIZE - WINNING\_LENGTH + 1):  
 window = [board[i+k][j+k] for k in range(WINNING\_LENGTH)]  
 score += evaluate\_window(window)  
  
 # Evaluate diagonals (negative slope)  
 for i in range(BOARD\_SIZE - WINNING\_LENGTH + 1):  
 for j in range(WINNING\_LENGTH - 1, BOARD\_SIZE):  
 window = [board[i+k][j-k] for k in range(WINNING\_LENGTH)]  
 score += evaluate\_window(window)  
  
 return score  
  
# Function to evaluate a window of cells  
def evaluate\_window(window):  
 score = 0  
 player\_count = window.count(PLAYER1)  
 opponent\_count = window.count(PLAYER2)  
  
 if player\_count == 0:  
 if opponent\_count == 1:  
 score -= 1  
 elif opponent\_count == 2:  
 score -= 10  
 elif opponent\_count == 3:  
 score -= 100  
 elif opponent\_count == 4:  
 score -= 1000  
 elif opponent\_count == 0:  
 if player\_count == 1:  
 score += 1  
 elif player\_count == 2:  
 score += 10  
 elif player\_count == 3:  
 score += 100  
 elif player\_count == 4:  
 score += 1000  
 else:  
 return 0 # Window contains both player's and opponent's pieces, no score  
  
 # Give higher score for windows that have a higher number of the computer player's pieces  
 if player\_count > 0:  
 score \*= 2  
  
 # Give even higher score if the window has a potential to win  
 if player\_count == WINNING\_LENGTH - 1 and opponent\_count == 0:  
 score \*= 10  
  
 return score  
  
# Function for manual move entry  
def manual\_move(player):  
 print(f"{player}'s turn!")  
 while True:  
 try:  
 move = input("Enter the move (row and column): ")  
 if len(move) == 2 and move[0].isdigit() and move[1].isalpha():  
 row = int(move[0]) - 1  
 col = ord(move[1].lower()) - ord('a')  
 if is\_valid\_move(row, col):  
 make\_move(player, row, col)  
 print\_board()  
 if is\_winner(player):  
 print(f"{player} wins!")  
 break  
 elif is\_board\_full():  
 print("It's a tie!")  
 else:  
 break  
 else:  
 print("Invalid move. Try again.")  
 else:  
 print("Invalid input. Try again.")  
 except ValueError:  
 print("Invalid input. Try again.")  
  
def manual\_vs\_manual():  
 print("Manual vs. Manual Mode")  
 while not is\_game\_over():  
 manual\_move(PLAYER1)  
 if is\_game\_over():  
 break  
 manual\_move(PLAYER2)  
  
# Function to start the manual vs. computer game  
def manual\_vs\_computer(depth):  
 print("Manual vs. computer Mode")  
 print\_board()  
 while not is\_game\_over():  
 # Player's turn  
 row, col = get\_player\_move(PLAYER1)  
 make\_move(PLAYER1, row, col)  
 print\_board()  
 if is\_game\_over():  
 break  
  
 # computer's turn  
 row, col = get\_computer\_move(PLAYER2, depth)  
 make\_move(PLAYER2, row, col)  
 print\_board()  
  
 if is\_winner(PLAYER1):  
 print("You win!")  
 elif is\_winner(PLAYER2):  
 print("computer wins!")  
 else:  
 print("It's a tie!")  
def computer\_vs\_manual(depth):  
 print("Computer vs. Manual Mode")  
 print\_board()  
 while not is\_game\_over():  
 # Computer's turn  
 row, col = get\_computer\_move(PLAYER1, depth)  
 make\_move(PLAYER1, row, col)  
 print("Computer played: " + str(row+1) + chr(ord('a') + col))  
 print\_board()  
 if is\_game\_over():  
 break  
  
 # Player's turn  
 row, col = get\_player\_move(PLAYER2)  
 make\_move(PLAYER2, row, col)  
 print\_board()  
  
 if is\_winner(PLAYER1):  
 print("Computer wins!")  
 elif is\_winner(PLAYER2):  
 print("You win!")  
 else:  
 print("It's a tie!")  
  
# Main function  
def select\_play\_mode():  
 print("Welcome to Magnetic Cave!")  
 print\_board()  
 print("Select play mode:")  
 print("1. Manual vs. Manual")  
 print("2. Manual vs. computer")  
 print("3. Computer vs. Manual")  
 print("4.exit")  
 mode = int(input("Enter the mode number: "))  
 if mode == 1:  
 manual\_vs\_manual()  
 elif mode == 2:  
 depth = int(input("Enter the depth of the search for the computer : "))  
 manual\_vs\_computer(depth)  
 elif mode == 3:  
 depth = int(input("Enter the depth of the search for the computer : "))  
 computer\_vs\_manual(depth)  
 elif mode == 4:  
 sys.exit()  
 else:  
 print("Invalid mode number. Try again.")  
  
# Start the game  
select\_play\_mode()