Birzeit University
Faculty of Engineering & Technology
Electrical & Computer Engineering Department

ENCS3340

Project Report

Magnetic Cave Game

Prepared by: Oday Ziq 1201168

Date: 20/06/2023

Table Of Contents

Formalization

This project involves implementing the game "Magnetic Cave" on an 8x8 chess board. The goal is to build a bridge of 5 consecutive magnetic bricks. The program will use a minimax algorithm with a 3-second time limit to make automatic moves. Manual and automatic gameplay modes will be supported, and the updated game board will be displayed after each move. Optional: An alpha-beta search algorithm can be implemented for improved performance.

Output & Design Description

Procedure

Create table:

```
def create_board():
    board = [[EMPTY] * BOARD_SIZE for _ in range(BOARD_SIZE)]
    return board


def print_board(board):
    print("    " + "   ".join(COLUMN_LABELS))
    print("   +" + "---+" * BOARD_SIZE)
    for i, row in enumerate(board):
        print(f"{i+1} | " + " | ".join(row) + " |")
        print("   +" + "---+" * BOARD_SIZE)
```

The code creates and prints a game board. The create_board function initializes a nested list representing the board, with each element initially set to EMPTY. The print_board function displays the board on the screen, including column labels and row separators.

Output:

```
Magnetic Cave Game
    a   b   c   d   e   f   g   h
   +---+---+---+---+---+---+---+---+
1 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
2 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
4 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
5 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
6 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
8 |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
```

Rule of game

Code & Description:

```
def is_valid_move(board, row, col):
    if row < 0 or row >= BOARD_SIZE or col < 0 or col >= BOARD_SIZE:
        return False

    if board[row][col] != EMPTY:
        return False

    if col == 0 or col == BOARD_SIZE - 1:
        return True

    if col > 0 and board[row][col - 1] != EMPTY:
        return True

    if col < BOARD_SIZE - 1 and board[row][col + 1] != EMPTY:
        return True

    return False


def make_move(board, row, col, player):
    board[row][col] = player


def is_winning_move(board, row, col):
    player = board[row][col]
    directions = [(1, 0), (0, 1), (1, 1), (1, -1)]

    for dx, dy in directions:
        count = 1
        r, c = row, col
        while count < WINNING_LENGTH and r - dx >= 0 and r - dx < BOARD_SIZE and c - dy >= 0 and c - dy < BOARD_SIZE and board[r
- dx][c - dy] == player:
            r -= dx
            c -= dy
            count += 1

        r, c = row, col
        while count < WINNING_LENGTH and r + dx >= 0 and r + dx < BOARD_SIZE and c + dy >= 0 and c + dy < BOARD_SIZE and board[r
+ dx][c + dy] == player:
            r += dx
            c += dy
            count += 1

        if count == WINNING_LENGTH:
            return True

    return False


def is_board_full(board):
    for row in board:
        if EMPTY in row:
            return False
    return True


def evaluate_position(board, player):
    score = 0
    directions = [(1, 0), (0, 1), (1, 1), (1, -1)]

    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            if board[row][col] == player:
                for dx, dy in directions:
                    count = 1
                    r, c = row, col
                    while count < WINNING_LENGTH and r - dx >= 0 and r - dx < BOARD_SIZE and c - dy >= 0 and c - dy < BOARD_SIZE
and board[r - dx][c - dy] == player:
                        r -= dx
                        c -= dy
                        count += 1

                    r, c = row, col
                    while count < WINNING_LENGTH and r + dx >= 0 and r + dx < BOARD_SIZE and c + dy >= 0 and c + dy < BOARD_SIZE
and board[r + dx][c + dy] == player:
                        r += dx
                        c += dy
                        count += 1

                    if count == WINNING_LENGTH:
                        score += 100

    return score
```

The provided code consists of several functions related to a game board. The is_valid_move function determines if a move at a specific row and column on the board is valid by checking if the coordinates are within the board boundaries and if the corresponding cell is empty. The make_move function updates the board by placing a player's symbol at a specified row and column. The is_winning_move function checks if a move at a given position on the board results in a winning move by examining the surrounding cells in all four directions: horizontally, vertically, and diagonally. The is_board_full function determines if the board is completely filled with symbols by checking each row for any empty cells. Lastly, the evaluate_position function evaluates the board for a specific player, assigning a score based on the number of potential winning moves in all four directions.

Output:

```
Player ■, make your move (e.g., a1):
a1
    a   b   c   d   e   f   g   h
  +---+---+---+---+---+---+---+---+
1 | ■ |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
2 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
4 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
5 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
6 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
8 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
Player □, make your move (e.g., a1):
e5
```

Alpha Beta search

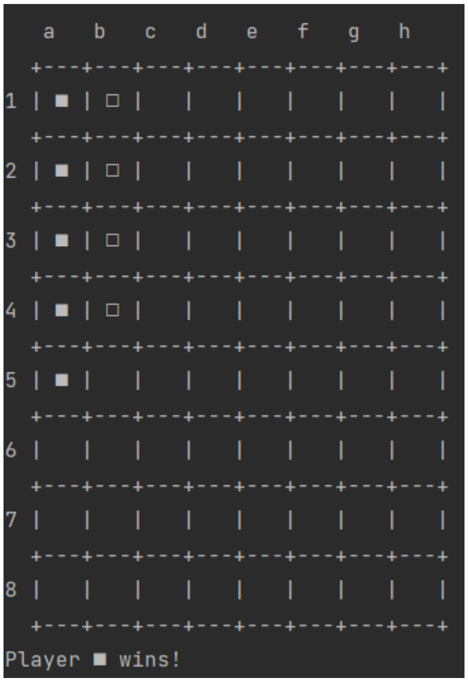
Code & Description:

```
def alpha_beta_search(board, depth, alpha, beta, maximizing_player):
    if depth == 0 or is_board_full(board):
        return None, evaluate_position(board, PLAYER_1) - evaluate_position(board,
PLAYER_2)

    if maximizing_player:
        max_score = float("-inf")
        best_move = None

        for row in range(BOARD_SIZE):
            for col in range(BOARD_SIZE):
                if is_valid_move(board, row, col):
                    make_move(board, row, col, PLAYER_1)
                    _, score = alpha_beta_search(board, depth - 1, alpha, beta, False)
                    make_move(board, row, col, EMPTY)

                    if score > max_score:
                        max_score = score
                        best_move = (row, col)

                    alpha = max(alpha, score)
                    if alpha >= beta:
                        break

        return best_move, max_score
    else:
        min_score = float("inf")
        best_move = None

        for row in range(BOARD_SIZE):
            for col in range(BOARD_SIZE):
                if is_valid_move(board, row, col):
                    make_move(board, row, col, PLAYER_2)
                    _, score = alpha_beta_search(board, depth - 1, alpha, beta, True)
                    make_move(board, row, col, EMPTY)

                    if score < min_score:
                        min_score = score
                        best_move = (row, col)

                    beta = min(beta, score)
                    if alpha >= beta:
                        break

        return best_move, min_score
```

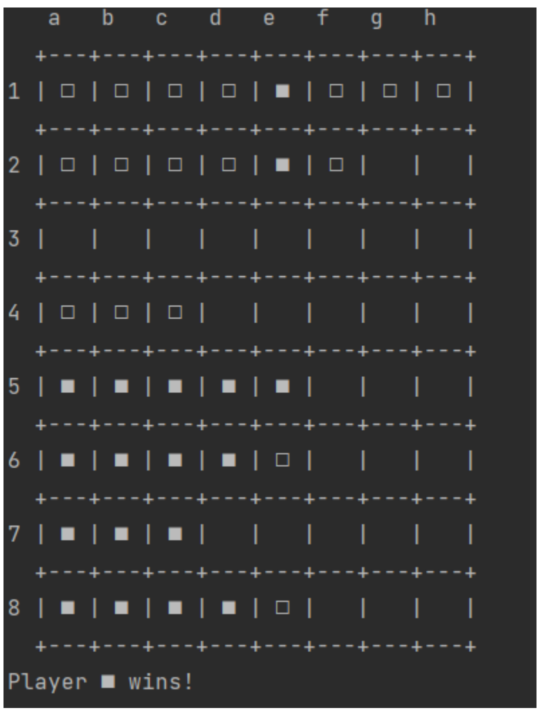
The code implements the alpha-beta search algorithm to find the best move in a game. It recursively evaluates possible moves and uses alpha-beta pruning to optimize the search. The algorithm alternates between maximizing and minimizing players, keeping track of the best move and its associated score. The function returns the best move and its score.

Test project

Put a mode 1 and start play and the first player won in align 5 consecutive bricks in a column.



Put a mode 2 and start play and the first player try to won in align 5 consecutive bricks in a lot of way and the computer try to forbid him and finally won in a row.



Conclusion

Summary of the implemented Magnetic Cave game using the minimax algorithm. Recap of the key features, including player turns, winning conditions, and play modes. Reflection on the performance and effectiveness of the implemented algorithm. Suggestions for future enhancements and optimizations.

Appendix

```
import time
import random

# Constants
EMPTY = " "
PLAYER_1 = "■"
PLAYER_2 = "□"
WINNING_LENGTH = 5
BOARD_SIZE = 8
COLUMN_LABELS = "abcdefgh"


def create_board():
    board = [[EMPTY] * BOARD_SIZE for _ in range(BOARD_SIZE)]
    return board


def print_board(board):
    print("   " + "   ".join(COLUMN_LABELS))
    print("  +" + "---+" * BOARD_SIZE)
    for i, row in enumerate(board):
        print(f"{i+1} | " + " | ".join(row) + " |")
        print("  +" + "---+" * BOARD_SIZE)


def is_valid_move(board, row, col):
    if row < 0 or row >= BOARD_SIZE or col < 0 or col >= BOARD_SIZE:
        return False

    if board[row][col] != EMPTY:
        return False

    if col == 0 or col == BOARD_SIZE - 1:
        return True

    if col > 0 and board[row][col - 1] != EMPTY:
        return True

    if col < BOARD_SIZE - 1 and board[row][col + 1] != EMPTY:
        return True

    return False


def make_move(board, row, col, player):
    board[row][col] = player


def is_winning_move(board, row, col):
    player = board[row][col]
    directions = [(1, 0), (0, 1), (1, 1), (1, -1)]

    for dx, dy in directions:
```

```python
        count = 1
        r, c = row, col
        while count < WINNING_LENGTH and r - dx >= 0 and r - dx < BOARD_SIZE and c - dy >= 0 and c - dy < BOARD_SIZE and board[r - dx][c - dy] == player:
            r -= dx
            c -= dy
            count += 1

        r, c = row, col
        while count < WINNING_LENGTH and r + dx >= 0 and r + dx < BOARD_SIZE and c + dy >= 0 and c + dy < BOARD_SIZE and board[r + dx][c + dy] == player:
            r += dx
            c += dy
            count += 1

        if count == WINNING_LENGTH:
            return True

    return False


def is_board_full(board):
    for row in board:
        if EMPTY in row:
            return False
    return True


def evaluate_position(board, player):
    score = 0
    directions = [(1, 0), (0, 1), (1, 1), (1, -1)]

    for row in range(BOARD_SIZE):
        for col in range(BOARD_SIZE):
            if board[row][col] == player:
                for dx, dy in directions:
                    count = 1
                    r, c = row, col
                    while count < WINNING_LENGTH and r - dx >= 0 and r - dx < BOARD_SIZE and c - dy >= 0 and c - dy < BOARD_SIZE and board[r - dx][c - dy] == player:
                        r -= dx
                        c -= dy
                        count += 1

                    r, c = row, col
                    while count < WINNING_LENGTH and r + dx >= 0 and r + dx < BOARD_SIZE and c + dy >= 0 and c + dy < BOARD_SIZE and board[r + dx][c + dy] == player:
                        r += dx
                        c += dy
                        count += 1

                    if count == WINNING_LENGTH:
                        score += 100

    return score
```

```python
def alpha_beta_search(board, depth, alpha, beta, maximizing_player):
    if depth == 0 or is_board_full(board):
        return None, evaluate_position(board, PLAYER_1) -
evaluate_position(board, PLAYER_2)

    if maximizing_player:
        max_score = float("-inf")
        best_move = None

        for row in range(BOARD_SIZE):
            for col in range(BOARD_SIZE):
                if is_valid_move(board, row, col):
                    make_move(board, row, col, PLAYER_1)
                    _, score = alpha_beta_search(board, depth - 1, alpha,
beta, False)
                    make_move(board, row, col, EMPTY)

                    if score > max_score:
                        max_score = score
                        best_move = (row, col)

                    alpha = max(alpha, score)
                    if alpha >= beta:
                        break

        return best_move, max_score
    else:
        min_score = float("inf")
        best_move = None

        for row in range(BOARD_SIZE):
            for col in range(BOARD_SIZE):
                if is_valid_move(board, row, col):
                    make_move(board, row, col, PLAYER_2)
                    _, score = alpha_beta_search(board, depth - 1, alpha,
beta, True)
                    make_move(board, row, col, EMPTY)

                    if score < min_score:
                        min_score = score
                        best_move = (row, col)

                    beta = min(beta, score)
                    if alpha >= beta:
                        break

        return best_move, min_score


def play_game(mode):
    board = create_board()
    print("Magnetic Cave Game")
    print_board(board)

    last_move_1 = None
```

```
    last_move_2 = None

    while True:
        if is_board_full(board):
            print("It's a tie!")
            break

        # Player ■ move
        while True:
            if mode == 1 or mode == 2:
                print("Player ■, make your move (e.g., a1): ")
                move = input().strip().lower()

                if len(move) != 2 or move[0] not in COLUMN_LABELS or move[1]
not in "12345678":
                    print("Invalid move. Please enter a valid move in the
format 'a1', 'b2', etc.")
                    continue

                col = COLUMN_LABELS.index(move[0])
                row = int(move[1]) - 1

                if is_valid_move(board, row, col):
                    make_move(board, row, col, PLAYER_1)
                    last_move_1 = (row, col)
                    break
                else:
                    print("Invalid move. Please try again.")
            else:
                start_time = time.time()
                best_move, _ = alpha_beta_search(board, depth=4,
alpha=float("-inf"), beta=float("inf"),
                                                 maximizing_player=True)
                end_time = time.time()
                print(f"Time taken for Player ■'s move: {end_time -
start_time} seconds")

                if best_move is None:
                    print("Player ■ cannot make a legal move. It's a tie!")
                    break

                make_move(board, best_move[0], best_move[1], PLAYER_1)
                last_move_1 = (best_move[0], best_move[1])
                print("Player ■ makes a move: ", COLUMN_LABELS[best_move[1]]
+ str(best_move[0] + 1))
                break

        print_board(board)

        if last_move_1 and is_winning_move(board, last_move_1[0],
last_move_1[1]):
            print("Player ■ wins!")
            break

        if is_board_full(board):
            print("It's a tie!")
            break
```

```
        # Player □ move
        while True:
            if mode == 1 or mode == 3:
                print("Player □, make your move (e.g., a1): ")
                move = input().strip().lower()

                if len(move) != 2 or move[0] not in COLUMN_LABELS or move[1]
not in "12345678":
                    print("Invalid move. Please enter a valid move in the
format 'a1', 'b2', etc.")
                    continue

                col = COLUMN_LABELS.index(move[0])
                row = int(move[1]) - 1

                if is_valid_move(board, row, col):
                    make_move(board, row, col, PLAYER_2)
                    last_move_2 = (row, col)
                    break
                else:
                    print("Invalid move. Please try again.")
            else:
                start_time = time.time()
                best_move, _ = alpha_beta_search(board, depth=4,
alpha=float("-inf"), beta=float("inf"),
                                                 maximizing_player=False)
                end_time = time.time()
                print(f"Time taken for Player □'s move: {end_time -
start_time} seconds")

                if best_move is None:
                    print("Player □ cannot make a legal move. It's a tie!")
                    break

                make_move(board, best_move[0], best_move[1], PLAYER_2)
                last_move_2 = (best_move[0], best_move[1])
                print("Player □ makes a move: ", COLUMN_LABELS[best_move[1]]
+ str(best_move[0] + 1))
                break

        print_board(board)

        if last_move_2 and is_winning_move(board, last_move_2[0],
last_move_2[1]):
            print("Player □ wins!")
            break


# Start the game
print("Select the game mode:")
print("1. Manual entry for both ■'s moves and □'s moves.")
print("2. Manual entry for ■'s moves and automatic moves for □.")
print("3. Manual entry for □'s moves and automatic moves for ■.")
mode = int(input("Enter the game mode (1, 2, or 3): "))

play_game(mode)
```