DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY

Artificial Intelligence Lab (CS4271)

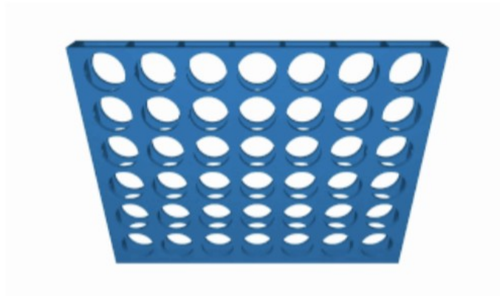
Name: SAGAR BASAK

Enrollment No: 2021CSB008

Assignment: 4

Question 1

1. Connect-4 is a strategic two-player game where participants choose a disc colour and take turns dropping their coloured discs into a seven-column, six-row grid.



Victory is achieved by forming a line of four discs horizontally, vertically, or diagonally. Several winning strategies enhance gameplay:

a. Middle Column Placement:

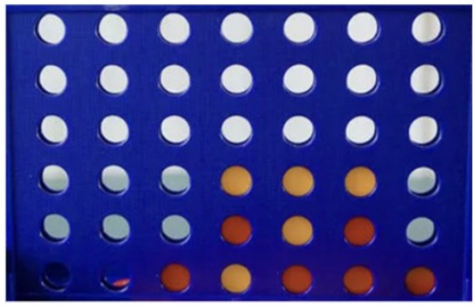
The player initiating the game benefits from placing the first disc in the middle column. This strategic move maximizes the possibilities for vertical, diagonal, and horizontal connections, totalling five potential ways to win.

b. Trapping Opponents:

To prevent losses, players strategically block their opponent's potential winning paths. For instance, placing a disc adjacent to an opponent's three-disc line disrupts their progression and protects the player from falling into traps set by the opponent.

c. "7" Formation:

Employing a "7" trap involves arranging discs to resemble the shape of a 7 on the board. This strategic move, which can be configured in various orientations, provides players with multiple directions to achieve a connect-four, adding versatility to their gameplay.



Connect-4 Implementation using Mini-Max Algorithm:

In this scenario, a user engages in a game against the computer, and the Mini-Max algorithm is employed to generate game states. Mini-Max, a backtracking algorithm widely used in decision-making and game theory, determines the optimal move for a player under the assumption that the opponent also plays optimally. Two players, the maximizer and the minimizer, aim to achieve the highest and lowest scores, respectively. A heuristic function calculates the values associated with each board state, representing the advantage of one player over the other.

Connect-4 Implementation using Alpha-Beta Pruning:

To optimize the Mini-Max algorithm, the Alpha-Beta Pruning technique is applied. Alpha-Beta Pruning involves passing two additional parameters, alpha and beta, to the Mini-Max function, reducing the number of evaluated nodes in the game tree. By introducing these parameters, the algorithm searches more efficiently, reaching greater depths in the game tree. Alpha-Beta Pruning accelerates the search process by eliminating the need to evaluate unnecessary branches when a superior move has been identified, resulting in significant computational time savings.

MINI-MAX with Alpha Beta Pruning

```
import numpy as np
import random
import sys
import time

ROW_COUNT = 6
COLUMN_COUNT = 7
PLAYER = 1
AI = 2
WINDOW_LENGTH = 4

def create_board():
    return np.zeros((ROW_COUNT, COLUMN_COUNT), dtype=int)
```

```
def drop_piece(board, row, col, piece):
    board[row][col] = piece

def is_valid_location(board, col):
    return board[ROW_COUNT - 1][col] == 0

def get_next_open_row(board, col):
    for r in range(ROW_COUNT):
        if board[r][col] == 0:
            return r

def winning_move(board, piece):
    for c in range(COLUMN_COUNT - 3):
        for r in range(ROW_COUNT):
            if all(board[r, c + i] == piece for i in
range(WINDOW_LENGTH)):
                return True
    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT - 3):
            if all(board[r + i, c] == piece for i in
range(WINDOW_LENGTH)):
                return True
    for c in range(COLUMN_COUNT - 3):
        for r in range(ROW_COUNT - 3):
            if all(board[r + i, c + i] == piece for i in
range(WINDOW_LENGTH)):
                return True
    for c in range(COLUMN_COUNT - 3):
        for r in range(3, ROW_COUNT):
            if all(board[r - i, c + i] == piece for i in
range(WINDOW_LENGTH)):
                return True
    return False

def score_position(board, piece):
    score = 0
    center_array = [board[r][COLUMN_COUNT//2] for r in
range(ROW_COUNT)]
    score += center_array.count(piece) * 3
    for r in range(ROW_COUNT):
        row_array = [board[r][c] for c in range(COLUMN_COUNT)]
        score += evaluate_window(row_array, piece)
    for c in range(COLUMN_COUNT):
        col_array = [board[r][c] for r in range(ROW_COUNT)]
        score += evaluate_window(col_array, piece)
    for r in range(ROW_COUNT - 3):
        for c in range(COLUMN_COUNT - 3):
            diag_array = [board[r + i][c + i] for i in
range(WINDOW_LENGTH)]
```

```
            score += evaluate_window(diag_array, piece)
    for r in range(ROW_COUNT - 3):
        for c in range(COLUMN_COUNT - 3):
            diag_array = [board[r + 3 - i][c + i] for i in
range(WINDOW_LENGTH)]
            score += evaluate_window(diag_array, piece)
    return score

def evaluate_window(window, piece):
    score = 0
    opp_piece = PLAYER if piece == AI else AI
    if window.count(piece) == 4:
        score += 100
    elif window.count(piece) == 3 and window.count(0) == 1:
        score += 5
    elif window.count(piece) == 2 and window.count(0) == 2:
        score += 2
    if window.count(opp_piece) == 3 and window.count(0) == 1:
        score -= 4
    return score

def minimax(board, depth, alpha, beta, maximizingPlayer):
    valid_locations = [c for c in range(COLUMN_COUNT) if
is_valid_location(board, c)]
    is_terminal = winning_move(board, PLAYER) or winning_move(board,
AI) or len(valid_locations) == 0
    if depth == 0 or is_terminal:
        if winning_move(board, AI):
            return (None, 1000000)
        elif winning_move(board, PLAYER):
            return (None, -1000000)
        elif len(valid_locations) == 0:
            return (None, 0)
        else:
            return (None, score_position(board, AI))
    if maximizingPlayer:
        value, column = -sys.maxsize, random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            temp_board = board.copy()
            drop_piece(temp_board, row, col, AI)
            new_score = minimax(temp_board, depth-1, alpha, beta,
False)[1]
            if new_score > value:
                value, column = new_score, col
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return column, value
```

```
    else:
        value, column = sys.maxsize, random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            temp_board = board.copy()
            drop_piece(temp_board, row, col, PLAYER)
            new_score = minimax(temp_board, depth-1, alpha, beta, True)[1]
            if new_score < value:
                value, column = new_score, col
            beta = min(beta, value)
            if alpha >= beta:
                break
        return column, value

def print_board(board):
    print(np.flip(board, 0))

def play_game():
    board = create_board()
    game_over = False
    turn = random.randint(0, 1)
    print_board(board)
    while not game_over:
        if turn == 0:
            col = int(input("Enter column (0-6): "))
            if is_valid_location(board, col):
                row = get_next_open_row(board, col)
                drop_piece(board, row, col, PLAYER)
                if winning_move(board, PLAYER):
                    print("Player wins!")
                    game_over = True
        else:
            start_time = time.time()
            col, _ = minimax(board, 5, -sys.maxsize, sys.maxsize, True)
            end_time = time.time()
            if is_valid_location(board, col):
                row = get_next_open_row(board, col)
                print("AI drops in column ", col)
                drop_piece(board, row, col, AI)
                if winning_move(board, AI):
                    print("AI wins!")
                    game_over = True
                print(f"Time taken for AI move: {end_time - start_time:.6f} seconds")
        print_board(board)
        if not game_over and all(board[ROW_COUNT-1][c] != 0 for c in range(COLUMN_COUNT)):
```

```python
            print("Game is a Tie!")
            game_over = True
        turn ^= 1

if __name__ == "__main__":
    play_game()
```

```
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
Enter column (0-6): 2
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0]]
AI drops in column  3
Time taken for AI move: 1.323855 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 1 2 0 0 0]]
Enter column (0-6): 2
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 1 2 0 0 0]]
AI drops in column  2
Time taken for AI move: 0.906718 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 2 0 0 0 0]
 [0 0 1 0 0 0 0]
 [0 0 1 2 0 0 0]]
Enter column (0-6): 3
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 2 0 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 1 2 0 0 0]]
```

```
AI drops in column  1
Time taken for AI move: 0.706996 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 2 0 0 0 0]
 [0 0 1 1 0 0 0]
 [0 2 1 2 0 0 0]]
Enter column (0-6): 4
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 2 0 0 0 0]
 [0 0 1 1 0 0 0]
 [0 2 1 2 1 0 0]]
AI drops in column  1
Time taken for AI move: 0.410851 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 2 0 0 0 0]
 [0 2 1 1 0 0 0]
 [0 2 1 2 1 0 0]]
Enter column (0-6): 0
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 2 0 0 0 0]
 [0 2 1 1 0 0 0]
 [1 2 1 2 1 0 0]]
AI drops in column  3
Time taken for AI move: 0.602734 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 2 2 0 0 0]
 [0 2 1 1 0 0 0]
 [1 2 1 2 1 0 0]]
Enter column (0-6): 4
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 2 2 0 0 0]
 [0 2 1 1 1 0 0]
 [1 2 1 2 1 0 0]]
AI drops in column  1
Time taken for AI move: 0.358601 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
```

```
 [0 0 0 0 0 0 0]
 [0 2 2 2 0 0 0]
 [0 2 1 1 1 0 0]
 [1 2 1 2 1 0 0]]
Enter column (0-6): 5
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 2 2 2 0 0 0]
 [0 2 1 1 1 0 0]
 [1 2 1 2 1 1 0]]
AI drops in column  1
AI wins!
Time taken for AI move: 0.063639 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 2 0 0 0 0 0]
 [0 2 2 2 0 0 0]
 [0 2 1 1 1 0 0]
 [1 2 1 2 1 1 0]]
```

MINI-MAX without Alpha Beta Pruning

```
import numpy as np
import random
import sys
import time

ROW_COUNT = 6
COLUMN_COUNT = 7
PLAYER = 1
AI = 2
WINDOW_LENGTH = 4

def create_board():
    return np.zeros((ROW_COUNT, COLUMN_COUNT), dtype=int)

def drop_piece(board, row, col, piece):
    board[row][col] = piece

def is_valid_location(board, col):
    return board[ROW_COUNT - 1][col] == 0

def get_next_open_row(board, col):
    for r in range(ROW_COUNT):
        if board[r][col] == 0:
            return r
```

```
def winning_move(board, piece):
    for c in range(COLUMN_COUNT - 3):
        for r in range(ROW_COUNT):
            if all(board[r, c + i] == piece for i in
range(WINDOW_LENGTH)):
                return True
    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT - 3):
            if all(board[r + i, c] == piece for i in
range(WINDOW_LENGTH)):
                return True
    for c in range(COLUMN_COUNT - 3):
        for r in range(ROW_COUNT - 3):
            if all(board[r + i, c + i] == piece for i in
range(WINDOW_LENGTH)):
                return True
    for c in range(COLUMN_COUNT - 3):
        for r in range(3, ROW_COUNT):
            if all(board[r - i, c + i] == piece for i in
range(WINDOW_LENGTH)):
                return True
    return False

def minimax_no_pruning(board, depth, maximizingPlayer):
    valid_locations = [c for c in range(COLUMN_COUNT) if
is_valid_location(board, c)]
    is_terminal = winning_move(board, PLAYER) or winning_move(board,
AI) or len(valid_locations) == 0
    if depth == 0 or is_terminal:
        if winning_move(board, AI):
            return (None, 1000000)
        elif winning_move(board, PLAYER):
            return (None, -1000000)
        elif len(valid_locations) == 0:
            return (None, 0)
        else:
            return (None, score_position(board, AI))
    if maximizingPlayer:
        value = -sys.maxsize
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            temp_board = board.copy()
            drop_piece(temp_board, row, col, AI)
            new_score = minimax_no_pruning(temp_board, depth-1, False)
[1]
            if new_score > value:
                value = new_score
                column = col
```

```
        return column, value
    else:
        value = sys.maxsize
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            temp_board = board.copy()
            drop_piece(temp_board, row, col, PLAYER)
            new_score = minimax_no_pruning(temp_board, depth-1, True)
[1]
            if new_score < value:
                value = new_score
                column = col
        return column, value

def play_game_no_pruning():
    board = create_board()
    game_over = False
    turn = random.randint(0, 1)
    print_board(board)
    while not game_over:
        if turn == 0:
            col = int(input("Enter column (0-6): "))
            if is_valid_location(board, col):
                row = get_next_open_row(board, col)
                drop_piece(board, row, col, PLAYER)
                if winning_move(board, PLAYER):
                    print("Player wins!")
                    game_over = True
        else:
            start_time = time.time()
            col, _ = minimax_no_pruning(board, 5, True)
            end_time = time.time()
            if is_valid_location(board, col):
                row = get_next_open_row(board, col)
                print("AI drops in column ", col)
                drop_piece(board, row, col, AI)
                if winning_move(board, AI):
                    print("AI wins!")
                    game_over = True
                print(f"Time taken for AI move (without pruning):
{end_time - start_time:.6f} seconds")
        print_board(board)
        if not game_over and all(board[ROW_COUNT - 1][c] != 0 for c in
range(COLUMN_COUNT)):
            print("Game is a Tie!")
            game_over = True
        turn ^= 1
```

```python
if __name__ == "__main__":
    play_game_no_pruning()
```

```
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
AI drops in column  3
Time taken for AI move (without pruning): 6.735429 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 2 0 0 0]]
Enter column (0-6): 4
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 2 1 0 0]]
AI drops in column  3
Time taken for AI move (without pruning): 6.732170 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 0 2 1 0 0]]
Enter column (0-6): 2
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 1 2 1 0 0]]
AI drops in column  3
Time taken for AI move (without pruning): 6.708231 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 1 2 1 0 0]]
Enter column (0-6): 3
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
```

```
 [0 0 0 1 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 1 2 1 0 0]]
AI drops in column  2
Time taken for AI move (without pruning): 7.726530 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 2 2 0 0 0]
 [0 0 1 2 1 0 0]]
Enter column (0-6): 4
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 2 2 1 0 0]
 [0 0 1 2 1 0 0]]
AI drops in column  3
Time taken for AI move (without pruning): 9.049201 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 2 2 1 0 0]
 [0 0 1 2 1 0 0]]
Enter column (0-6): 4
[[0 0 0 0 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 0 1 0 0 0]
 [0 0 0 2 1 0 0]
 [0 0 2 2 1 0 0]
 [0 0 1 2 1 0 0]]
AI drops in column  4
Time taken for AI move (without pruning): 5.683603 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 0 1 2 0 0]
 [0 0 0 2 1 0 0]
 [0 0 2 2 1 0 0]
 [0 0 1 2 1 0 0]]
Enter column (0-6): 5
[[0 0 0 0 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 0 1 2 0 0]
 [0 0 0 2 1 0 0]
 [0 0 2 2 1 0 0]
 [0 0 1 2 1 1 0]]
```

```
AI drops in column  1
AI wins!
Time taken for AI move (without pruning): 4.090410 seconds
[[0 0 0 0 0 0 0]
 [0 0 0 2 0 0 0]
 [0 0 0 1 2 0 0]
 [0 0 0 2 1 0 0]
 [0 0 2 2 1 0 0]
 [0 2 1 2 1 1 0]]
```

As we can see, mini-max without alpha beta pruning algorithm takes a lot more time than mini-max with alpha beta pruning. So mini-max with alpha beta pruning works more efficiently and effectively.