

**Department of Computing**

**CS370: Artificial Intelligence**

**Class: BSCS-10AB**

**Lab 06: AI-Based Game Solver**

**Date: 09-03-2023**

**Instructor: Dr. Hashir Kiani**

**Lab Engineer: Ms Shakeela Bibi**

**Lab Task: AI-Based Game Solver**

**Overview:** In this lab, you will implement an AI-based game solver to solve your chosen classic board game or puzzle. The game solver will use one or more of the AI techniques studied in the course to find the optimal solution or move sequence for the game.

**Lab Tasks:**

1. Game selection: You will choose a classic board game or puzzle to solve with an AI-based solver. Some possible options include chess, checkers, go sudoku, crossword puzzles, Rubik's cube, etc.
2. Algorithm design: You will design an AI algorithm to solve the game. The algorithm can be based on one or more of the AI techniques studied in the course. The algorithm should be designed to find the optimal solution or move sequence for the game.
3. Implementation: You will implement the AI algorithm in Python. You must also implement any necessary game logic, such as a game board, piece movements, and game rules.

**Description:**

* **Connect 4** is a two-player game where players take turns dropping coloured discs into a vertical board. The objective is to connect four discs of the same colour in a row, either horizontally, vertically, or diagonally.
* Alpha-beta pruning is an AI technique used to optimize the search algorithm used by the computer player in the game, allowing it to evaluate and select moves more efficiently by eliminating unnecessary branches of the game tree.

**Code:**

import numpy as np

# Define constants

row\_count = 6

col\_count = 7

nul = 0

Player1 = 1

Player2 = 2

win\_score = 1000000

dep\_limit = 6

positions=[1,2,3,4,5,6,7]

# Define the heuristic function

def evaluate\_board(board, player):

score = 0

opponent = Player1 if player == Player2 else Player2

# Check rows

for row in range(row\_count):

for col in range(col\_count - 3):

window = board[row, col:col+4]

if np.count\_nonzero(window == player) == 4:

score += win\_score

elif np.count\_nonzero(window == player) == 3 and np.count\_nonzero(window == nul) == 1:

score += 100

elif np.count\_nonzero(window == player) == 2 and np.count\_nonzero(window == nul) == 2:

score += 10

elif np.count\_nonzero(window == opponent) == 3 and np.count\_nonzero(window == nul) == 1:

score -= 100

# Check columns

for col in range(col\_count):

for row in range(row\_count - 3):

window = board[row:row+4, col]

if np.count\_nonzero(window == player) == 4:

score += win\_score

elif np.count\_nonzero(window == player) == 3 and np.count\_nonzero(window == nul) == 1:

score += 100

elif np.count\_nonzero(window == player) == 2 and np.count\_nonzero(window == nul) == 2:

score += 10

elif np.count\_nonzero(window == opponent) == 3 and np.count\_nonzero(window == nul) == 1:

score -= 100

# Check diagonals

for row in range(row\_count - 3):

for col in range(col\_count - 3):

window = np.array([board[row+i, col+i] for i in range(4)])

if np.count\_nonzero(window == player) == 4:

score += win\_score

elif np.count\_nonzero(window == player) == 3 and np.count\_nonzero(window == nul) == 1:

score += 100

elif np.count\_nonzero(window == player) == 2 and np.count\_nonzero(window == nul) == 2:

score += 10

elif np.count\_nonzero(window == opponent) == 3 and np.count\_nonzero(window == nul) == 1:

score -= 100

for row in range(row\_count - 3):

for col in range(3, col\_count):

window = np.array([board[row+i, col-i] for i in range(4)])

if np.count\_nonzero(window == player) == 4:

score += win\_score

elif np.count\_nonzero(window == player) == 3 and np.count\_nonzero(window == nul) == 1:

score += 100

elif np.count\_nonzero(window == player) == 2 and np.count\_nonzero(window == nul) == 2:

score += 10

elif np.count\_nonzero(window == opponent) == 3 and np.count\_nonzero(window == nul) == 3:

score -= 100

return score

def alpha\_beta\_pruning(board, depth, alpha, beta, maximizing\_player):

# Check if the game is over or depth limit is reached

if depth == 0 or np.count\_nonzero(board == nul) == 0:

return None, evaluate\_board(board, Player2)

# Check if maximizing player wins

if evaluate\_board(board, Player1) == win\_score:

return None, win\_score

# Check if minimizing player wins

if evaluate\_board(board, Player2) == win\_score:

return None, -win\_score

# Initialize best move and best score

best\_move = None

best\_score = -np.inf if maximizing\_player else np.inf

# Iterate over all possible moves

for col in range(col\_count):

# Check if the column is full

if board[row\_count-1, col] == nul:

# Make the move

row = np.argmax(board[:, col] == nul)

board[row, col] = Player1 if maximizing\_player else Player2

# Recursively search the next move

\_, score = alpha\_beta\_pruning(board, depth-1, alpha, beta, not maximizing\_player)

# Update the best move and best score

if maximizing\_player and score > best\_score:

best\_move = col

best\_score = score

alpha = max(alpha, score)

elif not maximizing\_player and score < best\_score:

best\_move = col

best\_score = score

beta = min(beta, score)

# Undo the move

board[row, col] = nul

# Prune the search if possible

if alpha >= beta:

break

return best\_move, best\_score

# Define the main function

def main():

# Initialize the game board

board = np.zeros((row\_count, col\_count), dtype=int)

# Print the initial game board

print("")

print(positions)

print(board)

# Play the game

while True:

# Player's move

col = int(input("Enter your move (1-7): "))

row = np.argmax(board[:, col] == nul)

board[row, col] = Player2

# Print the updated game board

print("Player's Turn:")

print(positions)

print(board)

# Check if the game is over

if evaluate\_board(board, Player2) == win\_score:

print("You win!")

break

elif np.count\_nonzero(board == nul) == 0:

print("It's a tie!")

break

# Computer's move

move, score = alpha\_beta\_pruning(board, dep\_limit, -np.inf, np.inf, True)

row = np.argmax(board[:, move] == nul)

board[row, move] = Player1

# Print the updated game board

print("Computer's Turn:")

print(positions)

print(board)

# Check if the game is over

if evaluate\_board(board, Player1) == win\_score:

print("You lose!")

break

elif np.count\_nonzero(board == nul) == 0:

print("It's a tie!")

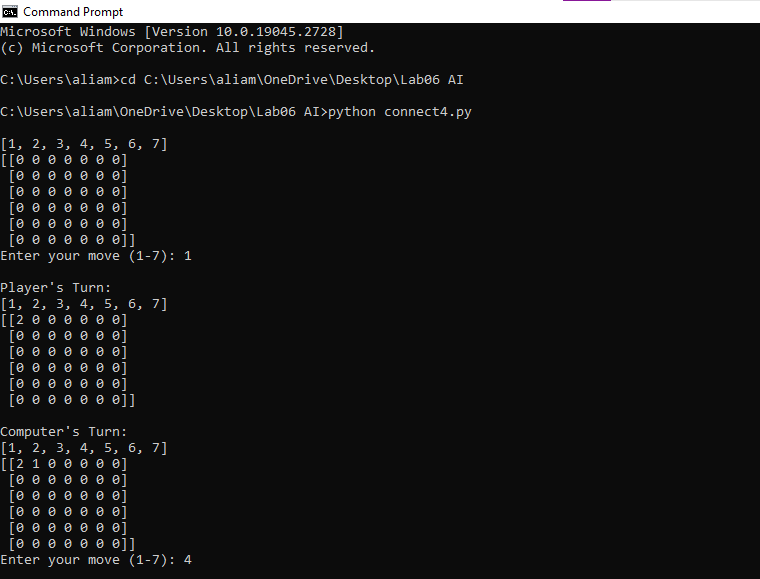
break

#Main function

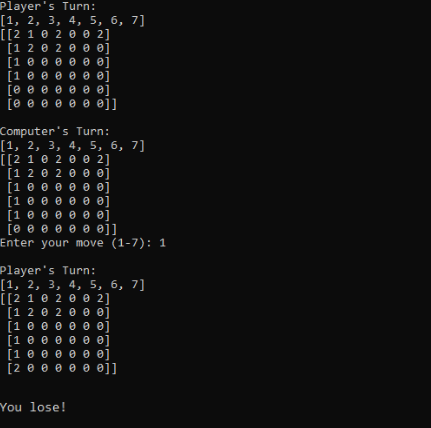
if \_\_name\_\_ == '\_\_main\_\_':

main()

**Screenshot**

****

* **The game takes input for the user and the computer plays its output.**

****

* **Once the Ai Completes its 4 pairs then the program outputs the winner.**