**Lab-10**

**Alpha Beta purning**

1. Tic tac toe

import math

# ----------- GAME LOGIC ------------

def print\_board(board):

print("\n")

for i in range(3):

print(" | ".join(board[i\*3:(i+1)\*3]))

if i < 2:

print("--+---+--")

print("\n")

def check\_winner(board):

"""Return 'X' or 'O' if there is a winner, or None otherwise."""

win\_combos = [

[0,1,2], [3,4,5], [6,7,8], # rows

[0,3,6], [1,4,7], [2,5,8], # columns

[0,4,8], [2,4,6] # diagonals

]

for combo in win\_combos:

if board[combo[0]] == board[combo[1]] == board[combo[2]] and board[combo[0]] != ' ':

return board[combo[0]]

return None

def is\_full(board):

return ' ' not in board

# ----------- ALPHA-BETA PRUNING ------------

def minimax(board, depth, alpha, beta, maximizing):

winner = check\_winner(board)

if winner == 'O': # AI wins

return 1

elif winner == 'X': # Human wins

return -1

elif is\_full(board): # Draw

return 0

if maximizing:

max\_eval = -math.inf

for i in range(9):

if board[i] == ' ':

board[i] = 'O'

eval = minimax(board, depth + 1, alpha, beta, False)

board[i] = ' '

max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval)

if beta <= alpha:

break

return max\_eval

else:

min\_eval = math.inf

for i in range(9):

if board[i] == ' ':

board[i] = 'X'

eval = minimax(board, depth + 1, alpha, beta, True)

board[i] = ' '

min\_eval = min(min\_eval, eval)

beta = min(beta, eval)

if beta <= alpha:

break

return min\_eval

def best\_move(board):

best\_val = -math.inf

move = -1

for i in range(9):

if board[i] == ' ':

board[i] = 'O'

move\_val = minimax(board, 0, -math.inf, math.inf, False)

board[i] = ' '

if move\_val > best\_val:

best\_val = move\_val

move = i

return move

# ----------- GAME LOOP ------------

def play\_game():

board = [' ' for \_ in range(9)]

print("Welcome to Tic-Tac-Toe with Alpha-Beta Pruning!")

print\_board(board)

print("You are 'X'. AI is 'O'. Enter positions 1-9 as below:")

print("1 | 2 | 3\n--+---+--\n4 | 5 | 6\n--+---+--\n7 | 8 | 9\n")

while True:

# Human move

while True:

try:

move = int(input("Enter your move (1-9): ")) - 1

if move < 0 or move > 8 or board[move] != ' ':

print("Invalid move. Try again.")

else:

board[move] = 'X'

break

except ValueError:

print("Please enter a valid number between 1 and 9.")

print\_board(board)

if check\_winner(board):

print("You win! 🎉")

break

if is\_full(board):

print("It's a draw!")

break

# AI move

print("AI is thinking...")

ai\_move = best\_move(board)

board[ai\_move] = 'O'

print\_board(board)

if check\_winner(board):

print("AI wins! 💻")

break

if is\_full(board):

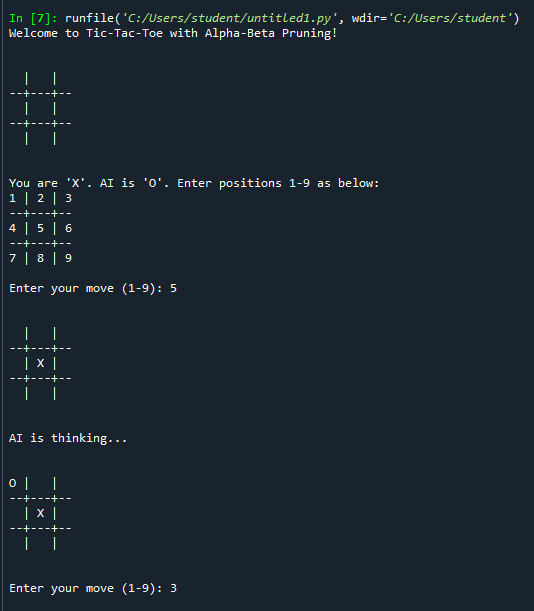
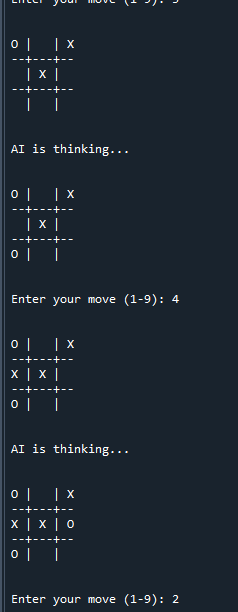
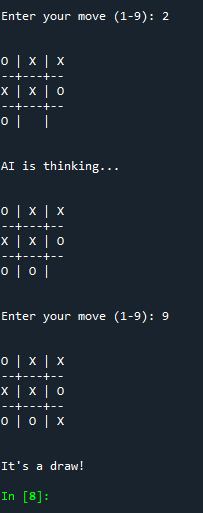
print("It's a draw!")

break

# ----------- RUN GAME ------------

if \_\_name\_\_ == "\_\_main\_\_":

play\_game()

1. 8 puzzle game

import heapq

goal\_state = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 0] # 0 represents the blank tile

]

# --- Helper functions ---

def find\_blank(board):

for i in range(3):

for j in range(3):

if board[i][j] == 0:

return i, j

def manhattan\_distance(board):

distance = 0

for i in range(3):

for j in range(3):

value = board[i][j]

if value != 0:

goal\_i, goal\_j = divmod(value - 1, 3)

distance += abs(i - goal\_i) + abs(j - goal\_j)

return distance

def get\_neighbors(board):

neighbors = []

i, j = find\_blank(board)

moves = [(1,0), (-1,0), (0,1), (0,-1)]

for di, dj in moves:

new\_i, new\_j = i + di, j + dj

if 0 <= new\_i < 3 and 0 <= new\_j < 3:

new\_board = [row[:] for row in board]

new\_board[i][j], new\_board[new\_i][new\_j] = new\_board[new\_i][new\_j], new\_board[i][j]

neighbors.append(new\_board)

return neighbors

def board\_to\_tuple(board):

return tuple(tuple(row) for row in board)

def print\_board(board):

for row in board:

print(" ".join(str(x) if x != 0 else '\_' for x in row))

print()

# --- A\* Search Algorithm ---

def a\_star(start):

open\_set = []

heapq.heappush(open\_set, (manhattan\_distance(start), 0, start))

came\_from = {}

g\_score = {board\_to\_tuple(start): 0}

visited = set()

while open\_set:

\_, cost, current = heapq.heappop(open\_set)

if current == goal\_state:

# Reconstruct path

path = [current]

while board\_to\_tuple(current) in came\_from:

current = came\_from[board\_to\_tuple(current)]

path.append(current)

path.reverse()

return path

visited.add(board\_to\_tuple(current))

for neighbor in get\_neighbors(current):

tentative\_g = g\_score[board\_to\_tuple(current)] + 1

neighbor\_tup = board\_to\_tuple(neighbor)

if neighbor\_tup in visited:

continue

if tentative\_g < g\_score.get(neighbor\_tup, float('inf')):

came\_from[neighbor\_tup] = current

g\_score[neighbor\_tup] = tentative\_g

f\_score = tentative\_g + manhattan\_distance(neighbor)

heapq.heappush(open\_set, (f\_score, tentative\_g, neighbor))

return None # No solution

# --- Main Program ---

if \_\_name\_\_ == "\_\_main\_\_":

print("Enter initial 8-puzzle configuration (use 0 for blank):")

start = []

for i in range(3):

row = list(map(int, input(f"Row {i+1}: ").split()))

start.append(row)

print("\nSolving...\n")

solution\_path = a\_star(start)

if solution\_path:

print(f"✅ Solution found in {len(solution\_path)-1} steps.\n")

for step, board in enumerate(solution\_path):

print(f"Step {step}:")

print\_board(board)

else:

print("❌ No solution exists for this puzzle.")

