1.Implement Non-AI Techniques for Tic-Tac-Toe Problem.

def print\_board(board):

print("\n")

for row in board:

print(" | ".join(row))

print("-" \* 5)

print("\n")

def check\_winner(board, player):

# Check rows and columns

for i in range(3):

if all([cell == player for cell in board[i]]): # Row

return True

if all([board[j][i] == player for j in range(3)]): # Column

return True

# Check diagonals

if all([board[i][i] == player for i in range(3)]):

return True

if all([board[i][2 - i] == player for i in range(3)]):

return True

return False

def is\_draw(board):

return all([cell != " " for row in board for cell in row])

def play\_game():

board = [[" " for \_ in range(3)] for \_ in range(3)]

current\_player = "X"

while True:

print\_board(board)

print(f"Player {current\_player}'s turn")

try:

row = int(input("Enter row (0-2): "))

col = int(input("Enter column (0-2): "))

except ValueError:

print("Invalid input. Please enter numbers 0-2.")

continue

if 0 <= row <= 2 and 0 <= col <= 2 and board[row][col] == " ":

board[row][col] = current\_player

else:

print("Invalid move! Try again.")

continue

if check\_winner(board, current\_player):

print\_board(board)

print(f"Player {current\_player} wins!")

break

if is\_draw(board):

print\_board(board)

print("It's a draw!")

break

current\_player = "O" if current\_player == "X" else "X"

# Run the game

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

play\_game()

2. Implement Non-AI Techniques for N-Queens problem.

def print\_board(position, N):

for row in range(N):

line = ""

for col in range(N):

if position[row] == col:

line += "Q "

else:

line += ". "

print(line)

print()

def is\_safe(position, row, col):

for i in range(row):

if (

position[i] == col or

position[i] - i == col - row or

position[i] + i == col + row

):

return False

return True

def solve\_n\_queens\_util(position, row, N, solutions):

if row == N:

solutions.append(position[:])

return

for col in range(N):

if is\_safe(position, row, col):

position[row] = col

solve\_n\_queens\_util(position, row + 1, N, solutions)

def solve\_n\_queens(N):

solutions = []

position = [-1] \* N

solve\_n\_queens\_util(position, 0, N, solutions)

print(f"Total Solutions for N={N}: {len(solutions)}\n")

for sol in solutions:

print\_board(sol, N)

# Run the N-Queens solver

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

N = 5 # You can change this value to any N (e.g., 8)

solve\_n\_queens(N)

3. Implement Magic-square

def generate\_magic\_square(n):

if n % 2 == 0:

raise ValueError("Only odd-sized magic squares are supported by this method.")

magic\_square = [[0] \* n for \_ in range(n)]

num = 1

i, j = 0, n // 2 # Start from top middle

while num <= n \* n:

magic\_square[i][j] = num

num += 1

new\_i = (i - 1) % n

new\_j = (j + 1) % n

if magic\_square[new\_i][new\_j]:

i = (i + 1) % n

else:

i, j = new\_i, new\_j

return magic\_square

def print\_magic\_square(square):

n = len(square)

print(f"\nMagic Square of size {n}x{n}:\n")

for row in square:

print(" ".join(f"{num:2d}" for num in row))

print(f"\nMagic Constant: {n \* (n \* n + 1) // 2}\n")

# Example usage

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

N = 5 # Try other odd numbers like 5, 7, etc.

square = generate\_magic\_square(N)

print\_magic\_square(square)

1. Implement the Water Jug problem using Depth First Search

def water\_jug\_dfs(cap1, cap2, target):

stack = [(0, 0)] # Stack for DFS

visited = set() # Track visited states

path = [] # Store the solution path

while stack:

state = stack.pop()

# If already visited, skip it

if state in visited:

continue

# Mark as visited

visited.add(state)

path.append(state)

x, y = state

# If we reached the target, return the path

if x == target or y == target:

return path

# Generate possible next states

next\_states = [

(cap1, y), # Fill Jug 1

(x, cap2), # Fill Jug 2

(0, y), # Empty Jug 1

(x, 0), # Empty Jug 2

# Pour Jug 1 -> Jug 2

(x - min(x, cap2 - y), y + min(x, cap2 - y)),

# Pour Jug 2 -> Jug 1

(x + min(y, cap1 - x), y - min(y, cap1 - x))

]

# Push new states to stack (DFS behavior)

for next\_state in next\_states:

if next\_state not in visited:

stack.append(next\_state)

return "No solution"

# User Input

jug1\_capacity = int(input("Enter capacity of Jug 1: "))

jug2\_capacity = int(input("Enter capacity of Jug 2: "))

target\_amount = int(input("Enter target amount: "))

# Solve the problem

solution = water\_jug\_dfs(jug1\_capacity, jug2\_capacity, target\_amount)

# Output the solution

print("Solution Path:", solution)

1. Implement the Water Jug problem using Breadth First Search

from collections import deque

def water\_jug\_bfs(cap1, cap2, target):

queue = deque([(0, 0)]) # BFS queue

visited = set() # Track visited states

parent = {} # Track the path

while queue:

state = queue.popleft()

# If already visited, skip

if state in visited:

continue

# Mark as visited

visited.add(state)

x, y = state

# Check if we reached the target

if x == target or y == target:

# Reconstruct the path

path = []

while state in parent:

path.append(state)

state = parent[state]

path.append((0, 0))

path.reverse()

return path

# Generate possible next states

next\_states = [

(cap1, y), # Fill Jug 1

(x, cap2), # Fill Jug 2

(0, y), # Empty Jug 1

(x, 0), # Empty Jug 2

# Pour Jug 1 -> Jug 2

(x - min(x, cap2 - y), y + min(x, cap2 - y)),

# Pour Jug 2 -> Jug 1

(x + min(y, cap1 - x), y - min(y, cap1 - x))

]

# Add new states to queue if not visited

for next\_state in next\_states:

if next\_state not in visited:

queue.append(next\_state)

parent[next\_state] = state # Store the path

return "No solution"

# Taking user input

jug1\_capacity = int(input("Enter capacity of Jug 1: "))

jug2\_capacity = int(input("Enter capacity of Jug 2: "))

target\_amount = int(input("Enter target amount: "))

solution = water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target\_amount)

print("Solution Path:", solution)

1. Implement the Hill Climbing technique to solve the 8 puzzle problem.

import copy

GOAL\_STATE = [

[1, 2, 3],

[4, 5, 6],

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

]

# Heuristic: Number of misplaced tiles

def heuristic(state):

misplaced = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0 and state[i][j] != GOAL\_STATE[i][j]:

misplaced += 1

return misplaced

# Get position of blank tile (0)

def find\_blank(state):

for i in range(3):

for j in range(3):

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

return i, j

# Generate all possible moves

def get\_neighbors(state):

neighbors = []

x, y = find\_blank(state)

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right

for dx, dy in moves:

nx, ny = x + dx, y + dy

if 0 <= nx < 3 and 0 <= ny < 3:

new\_state = copy.deepcopy(state)

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(new\_state)

return neighbors

# Hill Climbing algorithm

def hill\_climbing(start\_state, max\_iterations=1000):

current = start\_state

current\_h = heuristic(current)

steps = 0

while steps < max\_iterations:

neighbors = get\_neighbors(current)

next\_state = None

best\_h = current\_h

for neighbor in neighbors:

h = heuristic(neighbor)

if h < best\_h:

next\_state = neighbor

best\_h = h

if next\_state is None:

break # Local optimum reached

current = next\_state

current\_h = best\_h

steps += 1

print(f"Step {steps} - Heuristic: {current\_h}")

for row in current:

print(row)

print()

if current\_h == 0:

print("Goal reached!")

return current

print("Stopped at local optimum or max iterations.")

return current

# Example usage

start = [

[1, 2, 3],

[4, 5, 6],

[0, 7, 8]

]

final\_state = hill\_climbing(start)

1. Implement 8 Puzzle problem using Best First Search Algorithm

import heapq

# Heuristic: number of misplaced tiles

def heuristic(state, goal):

return sum(1 for i in range(9) if state[i] != goal[i] and state[i] != 0)

# Valid moves for each index position of the blank (0)

moves = {

0: [1, 3],

1: [0, 2, 4],

2: [1, 5],

3: [0, 4, 6],

4: [1, 3, 5, 7],

5: [2, 4, 8],

6: [3, 7],

7: [4, 6, 8],

8: [5, 7]

}

def best\_first\_8\_puzzle(start, goal):

visited = set()

pq = [(heuristic(start, goal), start, [])] # (cost, state, path)

while pq:

cost, state, path = heapq.heappop(pq)

if tuple(state) in visited:

continue

visited.add(tuple(state))

if state == goal:

return path + [state]

zero\_pos = state.index(0)

for move\_pos in moves[zero\_pos]:

new\_state = state[:]

new\_state[zero\_pos], new\_state[move\_pos] = new\_state[move\_pos], new\_state[zero\_pos]

if tuple(new\_state) not in visited:

heapq.heappush(pq, (heuristic(new\_state, goal), new\_state, path + [state]))

return None # No solution found

# Example usage

start = [1, 2, 3, 4, 0, 5, 6, 7, 8]

goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]

solution = best\_first\_8\_puzzle(start, goal)

# Print the path

if solution:

print("8 Puzzle Path:")

for step in solution:

print(step[:3])

print(step[3:6])

print(step[6:])

print("---")

else:

print("No solution found.")

1. Implement Cities Distance (shortest path) problem using Best First Search Algorithm

import heapq

def best\_first\_city(graph, start, goal, heuristics):

pq = [(heuristics[start], start, [])] # (heuristic, current\_city, path\_so\_far)

visited = set()

while pq:

\_, city, path = heapq.heappop(pq)

if city in visited:

continue

visited.add(city)

if city == goal:

return path + [city]

for neighbor, cost in graph.get(city, []):

if neighbor not in visited:

heapq.heappush(pq, (heuristics[neighbor], neighbor, path + [city]))

return None

# Graph as adjacency list: node -> [(neighbor, cost)]

graph = {

'A': [('B', 1), ('C', 2)],

'B': [('G', 4)],

'C': [('G', 1)],

'G': []

}

# Heuristic estimates from each city to goal 'G'

heuristics = {

'A': 3,

'B': 5,

'C': 2,

'G': 0

}

path = best\_first\_city(graph, 'A', 'G', heuristics)

print("City Path:", path)

1. Implement 8 Puzzle problem using A\* Algorithm

import heapq

# Heuristic: number of misplaced tiles

def heuristic(state, goal):

return sum(1 for i in range(9) if state[i] != goal[i] and state[i] != 0)

# Possible moves for each position of the blank tile (0)

moves = {

0: [1, 3],

1: [0, 2, 4],

2: [1, 5],

3: [0, 4, 6],

4: [1, 3, 5, 7],

5: [2, 4, 8],

6: [3, 7],

7: [4, 6, 8],

8: [5, 7]

}

def a\_star\_8\_puzzle(start, goal):

# Priority queue elements: (f = g + h, g, state, path)

pq = [(heuristic(start, goal), 0, start, [])]

visited = set()

while pq:

f, g, state, path = heapq.heappop(pq)

if tuple(state) in visited:

continue

visited.add(tuple(state))

if state == goal:

return path + [state]

zero\_pos = state.index(0)

for move\_pos in moves[zero\_pos]:

new\_state = state[:]

new\_state[zero\_pos], new\_state[move\_pos] = new\_state[move\_pos], new\_state[zero\_pos]

if tuple(new\_state) not in visited:

new\_g = g + 1

h = heuristic(new\_state, goal)

heapq.heappush(pq, (new\_g + h, new\_g, new\_state, path + [state]))

return None # No solution found

# Example

start = [1, 2, 3, 4, 0, 5, 6, 7, 8]

goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]

solution = a\_star\_8\_puzzle(start, goal)

# Print the solution path

if solution:

print("A\* 8 Puzzle Path:")

for step in solution:

print(step[:3])

print(step[3:6])

print(step[6:])

print("---")

else:

print("No solution found.")

1. Implement Cities Distance (shortest path) problem using A\* Algorithm

import heapq

def a\_star\_city(graph, start, goal, heuristics):

# Priority queue: (f = g + h, g, current\_city, path\_so\_far)

pq = [(heuristics[start], 0, start, [])]

visited = set()

while pq:

f, g, city, path = heapq.heappop(pq)

if city in visited:

continue

visited.add(city)

if city == goal:

return path + [city]

for neighbor, cost in graph.get(city, []):

if neighbor not in visited:

new\_g = g + cost

h = heuristics[neighbor]

heapq.heappush(pq, (new\_g + h, new\_g, neighbor, path + [city]))

return None # No path found

# Graph structure: city -> [(neighbor, distance)]

graph = {

'A': [('B', 1), ('C', 2)],

'B': [('G', 4)],

'C': [('G', 1)],

'G': []

}

# Heuristic values (estimated cost from each city to goal 'G')

heuristics = {

'A': 3,

'B': 5,

'C': 2,

'G': 0

}

# Find and print the path

path = a\_star\_city(graph, 'A', 'G', heuristics)

print("A\* City Path:", path)

1. Implement the Minimax algorithm to solve the Tic Tac Toe problem

import math

# Create empty 3x3 board

def create\_board():

return [[' ' for \_ in range(3)] for \_ in range(3)]

# Evaluate board state: +10 (X wins), -10 (O wins), 0 otherwise

def evaluate(board):

lines = []

# Rows and Columns

for i in range(3):

lines.append(board[i]) # rows

lines.append([board[0][i], board[1][i], board[2][i]]) # columns

# Diagonals

lines.append([board[0][0], board[1][1], board[2][2]])

lines.append([board[0][2], board[1][1], board[2][0]])

for line in lines:

if line == ['X', 'X', 'X']:

return 10

elif line == ['O', 'O', 'O']:

return -10

return 0

# Check if any move is left

def is\_moves\_left(board):

return any(cell == ' ' for row in board for cell in row)

# Minimax recursive algorithm

def minimax(board, depth, is\_max):

score = evaluate(board)

# Terminal states

if score == 10 or score == -10:

return score

if not is\_moves\_left(board):

return 0

if is\_max:

best = -math.inf

for i in range(3):

for j in range(3):

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

board[i][j] = 'X'

best = max(best, minimax(board, depth + 1, False))

board[i][j] = ' '

return best

else:

best = math.inf

for i in range(3):

for j in range(3):

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

board[i][j] = 'O'

best = min(best, minimax(board, depth + 1, True))

board[i][j] = ' '

return best

# AI chooses the best move

def find\_best\_move(board):

best\_val = -math.inf

best\_move = (-1, -1)

for i in range(3):

for j in range(3):

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

board[i][j] = 'X'

move\_val = minimax(board, 0, False)

board[i][j] = ' '

if move\_val > best\_val:

best\_val = move\_val

best\_move = (i, j)

return best\_move

# Print the current board

def print\_board(board):

for row in board:

print(' | '.join(row))

print('-' \* 5)

# Game Loop: Human (O) vs AI (X)

def play\_game():

board = create\_board()

print("Tic Tac Toe: You (O) vs AI (X)")

print\_board(board)

while True:

# Human move

try:

row = int(input("Enter row (0-2): "))

col = int(input("Enter col (0-2): "))

except ValueError:

print("Invalid input. Please enter 0, 1, or 2.")

continue

if not (0 <= row <= 2 and 0 <= col <= 2) or board[row][col] != ' ':

print("Invalid move!")

continue

board[row][col] = 'O'

if evaluate(board) == -10:

print\_board(board)

print("You win!")

break

if not is\_moves\_left(board):

print\_board(board)

print("Draw!")

break

# AI move

ai\_move = find\_best\_move(board)

board[ai\_move[0]][ai\_move[1]] = 'X'

print\_board(board)

if evaluate(board) == 10:

print("AI wins!")

break

if not is\_moves\_left(board):

print("Draw!")

break

# Run the game

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

play\_game()