1. **Implement Non-Al Techniques for Tic-Tac Toe the problem.**

def print\_board(board):

for row in board:

print(" | ".join(row))

print()

def check\_winner(board):

for row in board: # Check rows

if row[0] == row[1] == row[2] and row[0] != " ":

return row[0]

for col in range(3): # Check columns

if board[0][col] == board[1][col] == board[2][col] and board[0][col] != " ":

return board[0][col]

if board[0][0] == board[1][1] == board[2][2] and board[0][0] != " ": # Diagonal check

return board[0][0]

if board[0][2] == board[1][1] == board[2][0] and board[0][2] != " ": # Diagonal check

return board[0][2]

return None

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

player = "X"

for \_ in range(9):

print\_board(board)

row, col = map(int, input(f"Player {player}, enter row and column (0-2): ").split())

if board[row][col] == " ":

board[row][col] = player

winner = check\_winner(board)

if winner:

print\_board(board)

print(f"Player {winner} wins!")

break

player = "O" if player == "X" else "X"

else:

print("Invalid move! Try again.")

else:

print\_board(board)

print("It's a draw!")

1. **Implement Non-Al Techniques for N-Queens the problem.**

def solve\_n\_queens(n):

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

# Check this row on left side

for i in range(col):

if board[row][i] == 1:

return False

# Check upper diagonal on left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

# Check lower diagonal on left side

for i, j in zip(range(row, n), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solve(board, col):

if col >= n:

return True

for i in range(n):

if is\_safe(board, i, col):

board[i][col] = 1

if solve(board, col + 1):

return True

board[i][col] = 0 # Backtrack

return False

# Initialize empty board

board = [[0 for \_ in range(n)] for \_ in range(n)]

if not solve(board, 0):

print("Solution does not exist")

return None

return board

def print\_solution(board):

for row in board:

print(" ".join("Q" if cell == 1 else "." for cell in row))

# Example usage

n = 4 # Change this value for different board sizes

solution = solve\_n\_queens(n)

if solution:

print\_solution(solution)

**3.implement Non-Al Techniques for Magic Square the problem.**

def magic\_square(n):

if n % 2 == 0:

print("Only odd-order magic squares are supported.")

return

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

row, col = 0, n // 2

for num in range(1, n \* n + 1):

square[row][col] = num

new\_row, new\_col = (row - 1) % n, (col + 1) % n

if square[new\_row][new\_col]:

row = (row + 1) % n

else:

row, col = new\_row, new\_col

for row in square:

print(" ".join(str(num) for num in row))

n = int(input("Enter an odd number: "))

magic\_square(n)

**4.Implement the Water Jug problem using Depth First Search.**

from collections import deque

def is\_goal(state, target):

return target in state

def get\_next\_states(state, capacities):

next\_states = set()

for i in range(2):

for j in range(2):

if i != j: # Pour water from one jug to the other

transfer = min(state[i], capacities[j] - state[j])

new\_state = list(state)

new\_state[i] -= transfer

new\_state[j] += transfer

next\_states.add(tuple(new\_state))

# Fill a jug

new\_state = list(state)

new\_state[i] = capacities[i]

next\_states.add(tuple(new\_state))

# Empty a jug

new\_state = list(state)

new\_state[i] = 0

next\_states.add(tuple(new\_state))

return next\_states

def dfs(capacities, target):

stack = deque([(0, 0)]) # Initial state (both jugs empty)

visited = set()

while stack:

state = stack.pop()

if state in visited:

continue

visited.add(state)

print("Current state:", state)

if is\_goal(state, target):

print("Solution found!")

return

stack.extend(get\_next\_states(state, capacities))

print("No solution.")

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: "))

dfs((jug1\_capacity, jug2\_capacity), target\_amount)

**5.Implement the Water Jug problem using Breadth First Search**.

from collections import deque

def is\_goal(state, target):

return target in state

def get\_next\_states(state, capacities):

next\_states = set()

for i in range(2):

for j in range(2):

if i != j: # Pour water from one jug to another

transfer = min(state[i], capacities[j] - state[j])

new\_state = list(state)

new\_state[i] -= transfer

new\_state[j] += transfer

next\_states.add(tuple(new\_state))

# Fill a jug

new\_state = list(state)

new\_state[i] = capacities[i]

next\_states.add(tuple(new\_state))

# Empty a jug

new\_state = list(state)

new\_state[i] = 0

next\_states.add(tuple(new\_state))

return next\_states

def bfs(capacities, target):

queue = deque([(0, 0)]) # Initial state (both jugs empty)

visited = set()

while queue:

state = queue.popleft()

if state in visited:

continue

visited.add(state)

print("Current state:", state)

if is\_goal(state, target):

print("Solution found!")

return

queue.extend(get\_next\_states(state, capacities))

print("No solution.")

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: "))

bfs((jug1\_capacity, jug2\_capacity), target\_amount)

**6.Implement the Hill Climbing technique to solve the 8 puzzle problem.**

import random

def misplaced\_tiles(state, goal):

"""Heuristic: Count of misplaced tiles compared to goal state."""

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

def get\_neighbors(state):

"""Generates possible moves (neighbors) by swapping the empty tile."""

neighbors = []

zero\_index = state.index(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]}

for move in moves[zero\_index]:

new\_state = state[:]

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

neighbors.append(new\_state)

return neighbors

def hill\_climbing(start, goal):

"""Solves 8-puzzle using Hill Climbing with misplaced tiles heuristic."""

current\_state = start

while True:

print("Current state:", current\_state)

if current\_state == goal:

print("Solved!")

return

neighbors = get\_neighbors(current\_state)

best\_neighbor = min(neighbors, key=lambda x: misplaced\_tiles(x, goal))

if misplaced\_tiles(best\_neighbor, goal) >= misplaced\_tiles(current\_state, goal):

print("Stuck in local optimum.")

return

current\_state = best\_neighbor

start\_state = [1, 2, 3, 4, 0, 5, 6, 7, 8] # Example start state (0 represents the empty tile)

goal\_state = [1, 2, 3, 4, 5, 6, 7, 8, 0] # Goal state

hill\_climbing(start\_state, goal\_state)

7. **Implement 8 Puzzle problem using Best First Search Algorithm .**

import heapq

def heuristic(state, goal):

    """Heuristic: Number of misplaced tiles."""

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

def get\_neighbors(state):

    """Generates possible moves by swapping the empty tile (0) with neighbors."""

    neighbors = []

    zero\_index = state.index(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]}

    for move in moves[zero\_index]:

        new\_state = state[:]

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

        neighbors.append(new\_state)

    return neighbors

def best\_first\_search(start, goal):

    """Solves 8-Puzzle using Best First Search."""

    priority\_queue = []

    heapq.heappush(priority\_queue, (heuristic(start, goal), start))

    visited = set()

    while priority\_queue:

        \_, current\_state = heapq.heappop(priority\_queue)

        print("Current state:", current\_state)

        if current\_state == goal:

            print("Solved!")

            return

        visited.add(tuple(current\_state))

        for neighbor in get\_neighbors(current\_state):

            if tuple(neighbor) not in visited:

                heapq.heappush(priority\_queue, (heuristic(neighbor, goal), neighbor))

start\_state = [1, 2, 3, 4, 0, 5, 6, 7, 8]  # Example start state

goal\_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]  # Goal state

best\_first\_search(start\_state, goal\_state)

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

import heapq

def best\_first\_search(graph, start, goal):

    """Finds the shortest path using Best First Search."""

    priority\_queue = []

    heapq.heappush(priority\_queue, (0, [start]))  # (heuristic, path)

    visited = set()

    while priority\_queue:

        \_, path = heapq.heappop(priority\_queue)

        current\_city = path[-1]

        if current\_city == goal:

            print("Shortest path found:", " -> ".join(path))

            return path

        visited.add(current\_city)

        for neighbor, cost in graph.get(current\_city, []):

            if neighbor not in visited:

                heapq.heappush(priority\_queue, (cost, path + [neighbor]))

    print("No path found.")

    return None

# Example graph with cities and distances

graph = {

    "A": [("B", 4), ("C", 2)],

    "B": [("D", 5), ("E", 10)],

    "C": [("D", 8), ("F", 3)],

    "D": [("E", 2), ("G", 6)],

    "E": [("G", 3)],

    "F": [("G", 7)],

    "G": []

}

start\_city = input("Enter start city: ")

goal\_city = input("Enter goal city: ")

best\_first\_search(graph, start\_city, goal\_city)

**9.implement 8 Puzzle problem using A\* Algorithm.**

import heapq

def heuristic(city, goal, coordinates):

    """Calculates heuristic using straight-line (Euclidean) distance."""

    x1, y1 = coordinates[city]

    x2, y2 = coordinates[goal]

    return ((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2) \*\* 0.5

def a\_star(graph, start, goal, coordinates):

    """Finds the shortest path using A\* Algorithm."""

    priority\_queue = []

    heapq.heappush(priority\_queue, (0, [start], 0))  # (heuristic + cost, path, cost)

    visited = set()

    while priority\_queue:

        \_, path, cost\_so\_far = heapq.heappop(priority\_queue)

        current\_city = path[-1]

        if current\_city == goal:

            print("Shortest path found:", " -> ".join(path), f"with cost: {cost\_so\_far}")

            return path

        visited.add(current\_city)

        for neighbor, cost in graph.get(current\_city, []):

            if neighbor not in visited:

                total\_cost = cost\_so\_far + cost

                heuristic\_cost = heuristic(neighbor, goal, coordinates)

                heapq.heappush(priority\_queue, (total\_cost + heuristic\_cost, path + [neighbor], total\_cost))

    print("No path found.")

    return None

# Example cities graph with distances

graph = {

    "A": [("B", 4), ("C", 2)],

    "B": [("D", 5), ("E", 10)],

    "C": [("D", 8), ("F", 3)],

    "D": [("E", 2), ("G", 6)],

    "E": [("G", 3)],

    "F": [("G", 7)],

    "G": []

}

# Coordinates (for heuristic estimation)

coordinates = {

    "A": (0, 0), "B": (2, 1), "C": (1, 3), "D": (4, 2), "E": (5, 5),

    "F": (2, 4), "G": (6, 6)

}

start\_city = input("Enter start city: ")

goal\_city = input("Enter goal city: ")

a\_star(graph, start\_city, goal\_city, coordinates)

**10. implement Cities Distance (shortest path) problem using A" Algorithm.**

import heapq

def heuristic(city, goal, coordinates):

    """Calculates heuristic using straight-line (Euclidean) distance."""

    x1, y1 = coordinates[city]

    x2, y2 = coordinates[goal]

    return ((x1 - x2) \*\* 2 + (y1 - y2) \*\* 2) \*\* 0.5

def a\_star(graph, start, goal, coordinates):

    """Finds the shortest path using A\* Algorithm."""

    priority\_queue = []

    heapq.heappush(priority\_queue, (0, [start], 0))  # (heuristic + cost, path, cost)

    visited = set()

    while priority\_queue:

        \_, path, cost\_so\_far = heapq.heappop(priority\_queue)

        current\_city = path[-1]

        if current\_city == goal:

            print("Shortest path found:", " -> ".join(path), f"with cost: {cost\_so\_far}")

            return path

        visited.add(current\_city)

        for neighbor, cost in graph.get(current\_city, []):

            if neighbor not in visited:

                total\_cost = cost\_so\_far + cost

                heuristic\_cost = heuristic(neighbor, goal, coordinates)

                heapq.heappush(priority\_queue, (total\_cost + heuristic\_cost, path + [neighbor], total\_cost))

    print("No path found.")

    return None

# Example cities graph with distances

graph = {

    "A": [("B", 4), ("C", 2)],

    "B": [("D", 5), ("E", 10)],

    "C": [("D", 8), ("F", 3)],

    "D": [("E", 2), ("G", 6)],

    "E": [("G", 3)],

    "F": [("G", 7)],

    "G": []

}

# Coordinates (for heuristic estimation)

coordinates = {

    "A": (0, 0), "B": (2, 1), "C": (1, 3), "D": (4, 2), "E": (5, 5),

    "F": (2, 4), "G": (6, 6)

}

start\_city = input("Enter start city: ")

goal\_city = input("Enter goal city: ")

a\_star(graph, start\_city, goal\_city, coordinates)

**11. Implement the Minimax algorithm to solve the Tic Tac Toe problem.**

import math

def print\_board(board):

    for row in board:

        print(" | ".join(row))

    print()

def check\_winner(board):

    for row in board:  # Check rows

        if row[0] == row[1] == row[2] and row[0] != " ":

            return row[0]

    for col in range(3):  # Check columns

        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != " ":

            return board[0][col]

    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != " ":  # Diagonal check

        return board[0][0]

    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != " ":  # Diagonal check

        return board[0][2]

    return None

def is\_draw(board):

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

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

    winner = check\_winner(board)

    if winner == "X":

        return -10 + depth

    elif winner == "O":

        return 10 - depth

    elif is\_draw(board):

        return 0

    if is\_maximizing:

        best\_score = -math.inf

        for row in range(3):

            for col in range(3):

                if board[row][col] == " ":

                    board[row][col] = "O"

                    score = minimax(board, depth + 1, False)

                    board[row][col] = " "

                    best\_score = max(best\_score, score)

        return best\_score

    else:

        best\_score = math.inf

        for row in range(3):

            for col in range(3):

                if board[row][col] == " ":

                    board[row][col] = "X"

                    score = minimax(board, depth + 1, True)

                    board[row][col] = " "

                    best\_score = min(best\_score, score)

        return best\_score

def best\_move(board):

    best\_score = -math.inf

    move = (-1, -1)

    for row in range(3):

        for col in range(3):

            if board[row][col] == " ":

                board[row][col] = "O"

                score = minimax(board, 0, False)

                board[row][col] = " "

                if score > best\_score:

                    best\_score = score

                    move = (row, col)

    return move

# Initialize board

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

for turn in range(9):

    print\_board(board)

    if turn % 2 == 0:

        row, col = map(int, input("Player X, enter row and column (0-2): ").split())

        if board[row][col] == " ":

            board[row][col] = "X"

        else:

            print("Invalid move! Try again.")

            continue

    else:

        row, col = best\_move(board)

        board[row][col] = "O"

        print(f"Computer chooses: {row}, {col}")

    winner = check\_winner(board)

    if winner:

        print\_board(board)

        print(f"Player {winner} wins!")

        break

else:

    print\_board(board)

    print("It's a draw!")