**1.Implement Non-AI Techniques for the following problems:**

a. Tic Tac Toe

b. N Queens

c. Magic square

**2.Implement the Water Jug problem using the following uninformed search strategies:**

a. Depth First Search

b. Breadth First Search

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

**4. Implement the Best First Search to solve the following problems:**

a. 8 puzzle

b. Robot Navigation problem

c. Cities Distance (shortest path) problem

**5.Implement the A\* algorithm to solve the following problems given by HackerRank:**

a. N puzzle

b. Robot Navigation problem

c. Cities Distance (shortest path) problem

**6.Implement Constraint Satisfaction Algorithm for the following problems:**

a. Cryptarithmetic

b. Crossword puzzle

c. Map colouring problem

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

**8.Implement NLP problems statements by HackerRank for following:**

a. POS Tagging

b. Similarity Score

c. Spell Checker

**1.Implement Non-AI Techniques for the following problems:**

**a. Tic Tac Toe –**

def print\_board(board):

    print("-------------")

    for i in range(3):

        print("|", board[i\*3], "|", board[i\*3+1], "|", board[i\*3+2], "|")

        print("-------------")

def check\_winner(board, player):

    for i in range(3):

        if board[i\*3] == board[i\*3+1] == board[i\*3+2] == player:

            return True

        if board[i] == board[i+3] == board[i+6] == player:

            return True

    if board[0] == board[4] == board[8] == player:

        return True

    if board[2] == board[4] == board[6] == player:

        return True

    return False

def tic\_tac\_toe():

    board = [" " for \_ in range(9)]

    current\_player = "X"

    game\_over = False

    while not game\_over:

        print\_board(board)

        position = int(input(f"Player {current\_player}, enter your move (1-9): ")) - 1

        if board[position] == " ":

            board[position] = current\_player

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

                print\_board(board)

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

                game\_over = True

            elif " " not in board:

                print\_board(board)

                print("It's a tie!")

                game\_over = True

            else:

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

        else:

            print("Invalid move. Try again.")

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

    tic\_tac\_toe()

**b. N Queens –**

def printSolution(board, N):

    for i in range(N):

        for j in range(N):

            if board[i][j] == 1:

                print("Q", end=" ")

            else:

                print(".", end=" ")

        print()

def isSafe(board, row, col, N):

    for i in range(col):

        if board[row][i] == 1:

            return False

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

        if board[i][j] == 1:

            return False

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

        if board[i][j] == 1:

            return False

    return True

def solveNQUtil(board, col, N):

    if col >= N:

        return True

    for i in range(N):

        if isSafe(board, i, col, N):

            board[i][col] = 1

            if solveNQUtil(board, col + 1, N):

                return True

            board[i][col] = 0

    return False

def solveNQ(N):

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

    if not solveNQUtil(board, 0, N):

        print("Solution does not exist")

        return False

    printSolution(board, N)

    return True

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

    try:

        N = int(input("Enter the value of N: "))

        if N <= 0:

            print("Please enter a positive integer.")

        else:

            solveNQ(N)

    except ValueError:

        print("Invalid input. Please enter a valid integer.")

**c. Magic square –**

def generateSquare(n):

    if n % 2 == 0:

        print("Magic squares can only be generated for odd-sized matrices.")

        return

    magicSquare = [[0 for x in range(n)] for y in range(n)]

    i = n // 2

    j = n - 1

    num = 1

    while num <= (n \* n):

        if i == -1 and j == n:

            j = n - 2

            i = 0

        else:

            if j == n:

                j = 0

            if i < 0:

                i = n - 1

        if magicSquare[int(i)][int(j)]:

            j = j - 2

            i = i + 1

            continue

        else:

            magicSquare[int(i)][int(j)] = num

            num = num + 1

        j = j + 1

        i = i - 1

    print("Magic Square for n =", n)

    print("Sum of each row or column:", n \* (n \* n + 1) // 2, "\n")

    for i in range(0, n):

        for j in range(0, n):

            print('%2d ' % (magicSquare[i][j]), end='')

            if j == n - 1:

                print()

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

    try:

        n = int(input("Enter an odd value for N: "))

        if n <= 0:

            print("Please enter a positive integer.")

        elif n % 2 == 0:

            print("Magic squares can only be generated for odd-sized matrices.")

        else:

            generateSquare(n)

    except ValueError:

        print("Invalid input. Please enter a valid integer.")

**2.Implement the Water Jug problem using the following uninformed search strategies:**

**a. Depth First Search –**

from collections import defaultdict

#dfs

# jug1 and jug2 contain the value

# for max capacity in respective jugs

# and aim is the amount of water to be measured.

jug1, jug2, aim = 3, 5, 4

visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):

 # Checks for our goal and

 # returns true if achieved.

 if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):

  print(amt1, amt2)

  return True

 # Checks if we have already visited the

 # combination or not. If not, then it proceeds further.

 if visited[(amt1, amt2)] == False:

  print(amt1, amt2)

  # Changes the boolean value of the combination as it is visited.

  visited[(amt1, amt2)] = True

  # Check for all the 6 possibilities and

  # see if a solution is found in any one of them.

  return (waterJugSolver(0, amt2) or

    waterJugSolver(amt1, 0) or

    waterJugSolver(jug1, amt2) or

    waterJugSolver(amt1, jug2) or

    waterJugSolver(amt1 + min(amt2, (jug1-amt1)),

    amt2 - min(amt2, (jug1-amt1))) or

    waterJugSolver(amt1 - min(amt1, (jug2-amt2)),

    amt2 + min(amt1, (jug2-amt2))))

 # Return False if the combination is

 # already visited to avoid repetition otherwise

 # recursion will enter an infinite loop.

 else:

  return False

print("Steps: ")

waterJugSolver(0, 0)

**b. Breadth First Search –**

from collections import deque

def waterJugSolverBFS(jug1, jug2, aim):

    queue = deque([(0, 0)])

    visited = set([(0, 0)])

    parents = {}

    while queue:

        current\_state = queue.popleft()

        amt1, amt2 = current\_state

        if amt1 == aim or amt2 == aim:

            steps = []

            while current\_state != (0, 0):

                steps.append(current\_state)

                current\_state = parents[current\_state]

            steps.append((0, 0))

            steps.reverse()

            return steps

        next\_states = [

            (jug1, amt2),

            (amt1, jug2),

            (0, amt2),

            (amt1, 0),

            (min(jug1, amt1 + amt2), max(0, amt1 + amt2 - jug1)),

            (max(0, amt1 + amt2 - jug2), min(jug2, amt1 + amt2))

        ]

        for state in next\_states:

            if state not in visited:

                queue.append(state)

                visited.add(state)

                parents[state] = current\_state

    return None

# Example usage:

jug1\_capacity = 5

jug2\_capacity = 3

desired\_quantity = 4

solution = waterJugSolverBFS(jug1\_capacity, jug2\_capacity, desired\_quantity)

if solution:

    for step in solution:

        print(step)

else:

    print("No solution found.")

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

import random

import numpy as np

import networkx as nx

#coordinate of the points/cities

coordinate = np.array([[1,2], [30,21], [56,23], [8,18], [20,50], [3,4], [11,6], [6,7], [15,20], [10,9], [12,12]])

#adjacency matrix for a weighted graph based on the given coordinates

def generate\_matrix(coordinate):

    matrix = []

    for i in range(len(coordinate)):

        for j in range(len(coordinate)) :

            p = np.linalg.norm(coordinate[i] - coordinate[j])

            matrix.append(p)

    matrix = np.reshape(matrix, (len(coordinate),len(coordinate)))

    #print(matrix)

    return matrix

#finds a random solution

def solution(matrix):

    points = list(range(0, len(matrix)))

    solution = []

    for i in range(0, len(matrix)):

        random\_point = points[random.randint(0, len(points) - 1)]

        solution.append(random\_point)

        points.remove(random\_point)

    return solution

#calculate the path based on the random solution

def path\_length(matrix, solution):

    cycle\_length = 0

    for i in range(0, len(solution)):

        cycle\_length += matrix[solution[i]][solution[i - 1]]

    return cycle\_length

#generate neighbors of the random solution by swapping cities and returns the best neighbor

def neighbors(matrix, solution):

    neighbors = []

    for i in range(len(solution)):

        for j in range(i + 1, len(solution)):

            neighbor = solution.copy()

            neighbor[i] = solution[j]

            neighbor[j] = solution[i]

            neighbors.append(neighbor)

    #assume that the first neighbor in the list is the best neighbor

    best\_neighbor = neighbors[0]

    best\_path = path\_length(matrix, best\_neighbor)

    #check if there is a better neighbor

    for neighbor in neighbors:

        current\_path = path\_length(matrix, neighbor)

        if current\_path < best\_path:

            best\_path = current\_path

            best\_neighbor = neighbor

    return best\_neighbor, best\_path

def hill\_climbing(coordinate):

    matrix = generate\_matrix(coordinate)

    current\_solution = solution(matrix)

    current\_path = path\_length(matrix, current\_solution)

    neighbor = neighbors(matrix,current\_solution)[0]

    best\_neighbor, best\_neighbor\_path = neighbors(matrix, neighbor)

    while best\_neighbor\_path < current\_path:

        current\_solution = best\_neighbor

        current\_path = best\_neighbor\_path

        neighbor = neighbors(matrix, current\_solution)[0]

        best\_neighbor, best\_neighbor\_path = neighbors(matrix, neighbor)

    return current\_path, current\_solution

final\_solution = hill\_climbing(coordinate)

print("The solution is \n", final\_solution[1])

**4. Implement the Best First Search to solve the following problems:**

**a. 8 puzzle –**

import heapq

class PuzzleState:

    def \_\_init\_\_(self, board, parent=None, move=""):

        self.board = [row[:] for row in board]

        self.parent = parent

        self.move = move

        self.depth = 0

        if parent:

            self.depth = parent.depth + 1

    def \_\_eq\_\_(self, other):

        return self.board == other.board

    def \_\_lt\_\_(self, other):

        return self.heuristic() < other.heuristic()

    def \_\_hash\_\_(self):

        return hash(tuple(tuple(row) for row in self.board))

    def \_\_str\_\_(self):

        return '\n'.join([' '.join(map(str, row)) for row in self.board])

    def find\_blank(self):

        for i in range(3):

            for j in range(3):

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

                    return i, j

    def heuristic(self):

        # Manhattan distance heuristic

        distance = 0

        for i in range(3):

            for j in range(3):

                if self.board[i][j] != 0:

                    x, y = divmod(self.board[i][j] - 1, 3)

                    distance += abs(x - i) + abs(y - j)

        return distance

    def get\_successors(self):

        successors = []

        i, j = self.find\_blank()

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

        for di, dj, move in moves:

            ni, nj = i + di, j + dj

            if 0 <= ni < 3 and 0 <= nj < 3:

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

                new\_board[i][j], new\_board[ni][nj] = new\_board[ni][nj], new\_board[i][j]

                successors.append(PuzzleState(new\_board, self, move))

        return successors

    def is\_goal(self, goal\_board):

        return self.board == goal\_board

def best\_first\_search(initial\_board, goal\_board):

    initial\_state = PuzzleState(initial\_board)

    goal\_state = PuzzleState(goal\_board)

    open\_list = []

    closed\_set = set()

    heapq.heappush(open\_list, (initial\_state.heuristic(), initial\_state))

    while open\_list:

        \_, current = heapq.heappop(open\_list)

        if current.is\_goal(goal\_board):

            path = []

            while current.parent:

                path.append(current.move)

                current = current.parent

            return path[::-1]

        closed\_set.add(current)

        for successor in current.get\_successors():

            if successor not in closed\_set:

                heapq.heappush(open\_list, (successor.heuristic(), successor))

    return None  # No solution found

def get\_puzzle\_input(prompt):

    print(prompt)

    board = []

    for i in range(3):

        while True:

            row\_str = input(f"Enter row {i+1} (3 numbers separated by spaces, 0 for blank): ")

            try:

                row = list(map(int, row\_str.split()))

                if len(row) != 3:

                    raise ValueError

                board.append(row)

                break

            except ValueError:

                print("Invalid input. Please enter 3 numbers separated by spaces.")

    return board

def main():

    print("8-Puzzle Solver using Best-First Search")

    print("Enter the initial state:")

    initial\_board = get\_puzzle\_input("Enter the initial board (0 represents the blank space):")

    print("\nEnter the goal state:")

    goal\_board = get\_puzzle\_input("Enter the goal board (0 represents the blank space):")

    solution = best\_first\_search(initial\_board, goal\_board)

    if solution:

        print("\nSolution found in", len(solution), "moves:")

        print(" -> ".join(solution))

    else:

        print("\nNo solution found!")

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

    main()

**b. Robot Navigation problem –**

from collections import deque

def robot\_navigation\_bfs(grid, start, goal):

  rows, cols = len(grid), len(grid[0])

  if not (0 <= start[0] < rows and 0 <= start[1] < cols and

          0 <= goal[0] < rows and 0 <= goal[1] < cols and

          grid[start[0]][start[1]] == 0 and grid[goal[0]][goal[1]] == 0):

    return None  # Invalid start/goal or obstacles

  queue = deque([(start, [start])])  # Queue stores (position, path\_so\_far)

  visited = {start}  # Keep track of visited cells to avoid cycles

  while queue:

    (current\_row, current\_col), path = queue.popleft()

    if (current\_row, current\_col) == goal:

      return path

    # Explore valid neighboring cells

    for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:

      next\_row, next\_col = current\_row + dr, current\_col + dc

      if (0 <= next\_row < rows and 0 <= next\_col < cols and

          grid[next\_row][next\_col] == 0 and (next\_row, next\_col) not in visited):

        queue.append(((next\_row, next\_col), path + [(next\_row, next\_col)]))

        visited.add((next\_row, next\_col))

  return None  # No path found

# Example Usage:

grid = [

    [0, 0, 0, 0, 0],

    [0, 1, 1, 0, 0],

    [0, 0, 0, 0, 0],

    [0, 1, 1, 1, 0],

    [0, 0, 0, 0, 0]

]

start\_position = (0, 0)

goal\_position = (4, 4)

path = robot\_navigation\_bfs(grid, start\_position, goal\_position)

if path:

  print("Path found:")

  for cell in path:

    print(cell)

else:

  print("No path found.")

**c. Cities Distance (shortest path) problem-**

import heapq

class CityGraph:

    def \_\_init\_\_(self):

        self.graph = {}

    def add\_edge(self, city1, city2, distance):

        if city1 not in self.graph:

            self.graph[city1] = {}

        if city2 not in self.graph:

            self.graph[city2] = {}

        self.graph[city1][city2] = distance

        self.graph[city2][city1] = distance

class CityState:

    def \_\_init\_\_(self, city, parent=None, cost=0):

        self.city = city

        self.parent = parent

        self.cost = cost

    def \_\_eq\_\_(self, other):

        return self.city == other.city

    def \_\_lt\_\_(self, other):

        return self.heuristic() < other.heuristic()

    def \_\_hash\_\_(self):

        return hash(self.city)

    def heuristic(self, goal\_city, straight\_line\_distances):

        # Use straight-line distance as heuristic (must be admissible)

        return straight\_line\_distances.get(self.city, {}).get(goal\_city, 0)

def best\_first\_search\_cities(graph, start, goal, straight\_line\_distances):

    open\_list = []

    closed\_set = set()

    start\_state = CityState(start)

    heapq.heappush(open\_list, (start\_state.heuristic(goal, straight\_line\_distances), start\_state))

    while open\_list:

        \_, current = heapq.heappop(open\_list)

        if current.city == goal:

            path = []

            total\_distance = current.cost

            while current.parent:

                path.append(current.city)

                current = current.parent

            path.append(start)

            return path[::-1], total\_distance

        closed\_set.add(current.city)

        for neighbor, distance in graph.graph[current.city].items():

            if neighbor not in closed\_set:

                new\_cost = current.cost + distance

                new\_state = CityState(neighbor, current, new\_cost)

                heapq.heappush(open\_list,

                             (new\_state.heuristic(goal, straight\_line\_distances), new\_state))

    return None, None  # No path found

# Example usage

city\_graph = CityGraph()

city\_graph.add\_edge('A', 'B', 4)

city\_graph.add\_edge('A', 'C', 2)

city\_graph.add\_edge('B', 'C', 1)

city\_graph.add\_edge('B', 'D', 5)

city\_graph.add\_edge('C', 'D', 8)

city\_graph.add\_edge('C', 'E', 10)

city\_graph.add\_edge('D', 'E', 2)

# Straight-line distances (heuristic) - must be <= actual distance

straight\_line\_dist = {

    'A': {'E': 10},

    'B': {'E': 8},

    'C': {'E': 7},

    'D': {'E': 2},

    'E': {'E': 0}

}

start\_city = 'A'

goal\_city = 'E'

path, distance = best\_first\_search\_cities(city\_graph, start\_city, goal\_city, straight\_line\_dist)

print("Cities Path Solution:", path)

print("Total Distance:", distance)

**5.Implement the A\* algorithm to solve the following problems given by HackerRank:**

**a. N puzzle –**

import heapq

class NPuzzleState:

def \_\_init\_\_(self, board, parent=None, move=""):

self.board = [row[:] for row in board]

self.parent = parent

self.move = move

self.g = 0 # cost from start

if parent:

self.g = parent.g + 1

def \_\_eq\_\_(self, other):

return self.board == other.board

def \_\_lt\_\_(self, other):

return (self.g + self.heuristic()) < (other.g + other.heuristic())

def \_\_hash\_\_(self):

return hash(tuple(tuple(row) for row in self.board))

def find\_blank(self):

for i in range(len(self.board)):

for j in range(len(self.board[0])):

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

return i, j

def heuristic(self):

# Manhattan distance heuristic

distance = 0

n = len(self.board)

for i in range(n):

for j in range(n):

if self.board[i][j] != 0:

x, y = divmod(self.board[i][j] - 1, n)

distance += abs(x - i) + abs(y - j)

return distance

def get\_successors(self):

successors = []

i, j = self.find\_blank()

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

for di, dj, move in moves:

ni, nj = i + di, j + dj

if 0 <= ni < len(self.board) and 0 <= nj < len(self.board[0]):

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

new\_board[i][j], new\_board[ni][nj] = new\_board[ni][nj], new\_board[i][j]

successors.append(NPuzzleState(new\_board, self, move))

return successors

def a\_star\_npuzzle(initial\_board, goal\_board):

initial\_state = NPuzzleState(initial\_board)

goal\_state = NPuzzleState(goal\_board)

open\_list = []

closed\_set = set()

heapq.heappush(open\_list, (initial\_state.g + initial\_state.heuristic(), initial\_state))

while open\_list:

\_, current = heapq.heappop(open\_list)

if current.board == goal\_state.board:

path = []

while current.parent:

path.append(current.move)

current = current.parent

return path[::-1]

closed\_set.add(current)

for successor in current.get\_successors():

if successor not in closed\_set:

heapq.heappush(open\_list, (successor.g + successor.heuristic(), successor))

return None

# Example usage for 8-puzzle (3x3)

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

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

print("N-Puzzle Solution:", a\_star\_npuzzle(initial, goal))

**b. Robot Navigation problem –**

import heapq

import math

class RobotState:

    def \_\_init\_\_(self, position, parent=None, action=None):

        self.position = position

        self.parent = parent

        self.action = action

        self.g = 0  # cost from start

        if parent:

            self.g = parent.g + 1

    def \_\_eq\_\_(self, other):

        return self.position == other.position

    def \_\_lt\_\_(self, other):

        return (self.g + self.heuristic()) < (other.g + other.heuristic())

    def \_\_hash\_\_(self):

        return hash(self.position)

    def heuristic(self, goal\_position):

        # Euclidean distance heuristic

        return math.sqrt((self.position[0] - goal\_position[0])\*\*2 +

                        (self.position[1] - goal\_position[1])\*\*2)

    def get\_successors(self, grid):

        successors = []

        rows, cols = len(grid), len(grid[0])

        x, y = self.position

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

        for dx, dy, action in moves:

            nx, ny = x + dx, y + dy

            if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] != 1:

                successors.append(RobotState((nx, ny), self, action))

        return successors

def a\_star\_robot(grid, start, goal):

    open\_list = []

    closed\_set = set()

    start\_state = RobotState(start)

    heapq.heappush(open\_list, (start\_state.g + start\_state.heuristic(goal), start\_state))

    while open\_list:

        \_, current = heapq.heappop(open\_list)

        if current.position == goal:

            path = []

            while current.parent:

                path.append(current.action)

                current = current.parent

            return path[::-1]

        closed\_set.add(current)

        for successor in current.get\_successors(grid):

            if successor not in closed\_set:

                heapq.heappush(open\_list,

                             (successor.g + successor.heuristic(goal), successor))

    return None

# Example usage

grid = [

    [0, 0, 0, 0, 1],

    [1, 1, 0, 1, 0],

    [0, 0, 0, 0, 0],

    [0, 1, 1, 1, 0],

    [0, 0, 0, 0, 0]

]

start = (0, 0)

goal = (4, 4)

print("Robot Navigation Solution:", a\_star\_robot(grid, start, goal))

**c. Cities Distance (shortest path) problem-**

import heapq

class CityGraph:

    def \_\_init\_\_(self):

        self.graph = {}

    def add\_edge(self, city1, city2, distance):

        if city1 not in self.graph:

            self.graph[city1] = {}

        if city2 not in self.graph:

            self.graph[city2] = {}

        self.graph[city1][city2] = distance

        self.graph[city2][city1] = distance

class CityState:

    def \_\_init\_\_(self, city, parent=None, cost=0, goal\_city=None, straight\_line\_distances=None):

        self.city = city

        self.parent = parent

        self.cost = cost  # g value

        self.goal\_city = goal\_city

        self.straight\_line\_distances = straight\_line\_distances

    def \_\_eq\_\_(self, other):

        return self.city == other.city

    def \_\_lt\_\_(self, other):

        return (self.cost + self.heuristic()) < (other.cost + other.heuristic())

    def \_\_hash\_\_(self):

        return hash(self.city)

    def heuristic(self):

        # Use straight-line distance as heuristic (must be admissible)

        return self.straight\_line\_distances.get(self.city, {}).get(self.goal\_city, 0)

def a\_star\_cities(graph, start, goal, straight\_line\_distances):

    open\_list = []

    closed\_set = set()

    start\_state = CityState(start, None, 0, goal, straight\_line\_distances)

    heapq.heappush(open\_list,

                  (start\_state.cost + start\_state.heuristic(),

                  start\_state))

    while open\_list:

        \_, current = heapq.heappop(open\_list)

        if current.city == goal:

            path = []

            total\_distance = current.cost

            while current.parent:

                path.append(current.city)

                current = current.parent

            path.append(start)

            return path[::-1], total\_distance

        closed\_set.add(current.city)

        for neighbor, distance in graph.graph[current.city].items():

            if neighbor not in closed\_set:

                new\_cost = current.cost + distance

                new\_state = CityState(neighbor, current, new\_cost, goal, straight\_line\_distances)

                heapq.heappush(open\_list,

                             (new\_state.cost + new\_state.heuristic(),

                             new\_state))

    return None, None

# Example usage

city\_graph = CityGraph()

city\_graph.add\_edge('A', 'B', 4)

city\_graph.add\_edge('A', 'C', 2)

city\_graph.add\_edge('B', 'C', 1)

city\_graph.add\_edge('B', 'D', 5)

city\_graph.add\_edge('C', 'D', 8)

city\_graph.add\_edge('C', 'E', 10)

city\_graph.add\_edge('D', 'E', 2)

# Straight-line distances (must be <= actual distance)

straight\_line\_dist = {

    'A': {'E': 10},

    'B': {'E': 8},

    'C': {'E': 7},

    'D': {'E': 2},

    'E': {'E': 0}

}

start = 'A'

goal = 'E'

path, distance = a\_star\_cities(city\_graph, start, goal, straight\_line\_dist)

print("Cities Path Solution:", path)

print("Total Distance:", distance)

**6.Implement Constraint Satisfaction Algorithm for the following problems:**

**a. Cryptarithmetic –**

from itertools import permutations

def solve\_cryptarithmetic():

    print("Cryptarithmetic Puzzle Solver")

    print("Enter the puzzle in the format: SEND + MORE = MONEY")

    puzzle = input("Enter the puzzle: ").upper().replace(" ", "")

    try:

        # Parse the input

        parts = puzzle.split('=')

        left\_part = parts[0].split('+')

        word1 = left\_part[0].strip()

        word2 = left\_part[1].strip()

        result\_word = parts[1].strip()

    except:

        print("Invalid input format. Please use format like 'SEND + MORE = MONEY'")

        return None

    # Get all unique letters

    letters = sorted(list(set(word1 + word2 + result\_word)))

    if len(letters) > 10:

        print("Error: Too many unique letters (maximum 10)")

        return None

    print(f"\nSolving: {word1} + {word2} = {result\_word}")

    print(f"Unique letters: {', '.join(letters)}")

    digits = range(10)

    first\_letters = {word1[0], word2[0], result\_word[0]}

    for perm in permutations(digits, len(letters)):

        sol = dict(zip(letters, perm))

        # Skip solutions where first letters are 0

        if any(sol[letter] == 0 for letter in first\_letters):

            continue

        # Calculate numerical values

        def word\_to\_num(word):

            num = 0

            for c in word:

                num = num \* 10 + sol[c]

            return num

        num1 = word\_to\_num(word1)

        num2 = word\_to\_num(word2)

        result = word\_to\_num(result\_word)

        if num1 + num2 == result:

            print("\nSolution found:")

            print(f"{word1}: {num1}")

            print(f"{word2}: {num2}")

            print(f"{result\_word}: {result}")

            print("Letter assignments:")

            for letter in letters:

                print(f"{letter}: {sol[letter]}")

            return sol

    print("\nNo solution found")

    return None

# Example usage:

# When prompted, enter: SEND + MORE = MONEY

solve\_cryptarithmetic()

**b. Crossword puzzle –**

class CrosswordCSP:

    def \_\_init\_\_(self, variables, domains, overlaps):

        self.variables = variables

        self.domains = domains

        self.overlaps = overlaps  # (var1, var2): (i, j) means var1[i] == var2[j]

        self.assignment = {}

    def is\_consistent(self, var, word):

        for other\_var in self.assignment:

            if var == other\_var:

                continue

            if (var, other\_var) in self.overlaps:

                i, j = self.overlaps[(var, other\_var)]

                if word[i] != self.assignment[other\_var][j]:

                    return False

            elif (other\_var, var) in self.overlaps:

                j, i = self.overlaps[(other\_var, var)]

                if word[i] != self.assignment[other\_var][j]:

                    return False

        return True

    def backtrack(self):

        if len(self.assignment) == len(self.variables):

            return self.assignment

        unassigned\_vars = [v for v in self.variables if v not in self.assignment]

        var = unassigned\_vars[0]

        for word in self.domains[var]:

            if self.is\_consistent(var, word):

                self.assignment[var] = word

                result = self.backtrack()

                if result:

                    return result

                del self.assignment[var]

        return None

# Variables

variables = ['H1', 'H2', 'V1', 'V2']

# Domains (word list of length 3)

word\_list = ['CAT', 'DOG', 'RAT', 'MAT', 'CAR', 'BAR']

domains = {v: word\_list for v in variables}

# Overlap constraints: (var1, var2): (i, j)

# H1[0] == V1[0], H1[2] == V2[0], H2[0] == V1[2], H2[2] == V2[2]

overlaps = {

    ('H1', 'V1'): (0, 0),

    ('H1', 'V2'): (2, 0),

    ('H2', 'V1'): (0, 2),

    ('H2', 'V2'): (2, 2),

}

# Solve

csp = CrosswordCSP(variables, domains, overlaps)

solution = csp.backtrack()

# Display

if solution:

    for var in sorted(solution):

        print(f"{var}: {solution[var]}")

else:

    print("No solution found.")

**c. Map colouring problem -**

def solve\_map\_coloring():

    # Australia map with adjacent regions

    map = {

        'WA': ['NT', 'SA'],

        'NT': ['WA', 'SA', 'Q'],

        'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],

        'Q': ['NT', 'SA', 'NSW'],

        'NSW': ['Q', 'SA', 'V'],

        'V': ['SA', 'NSW'],

        'T': []

    }

    colors = ['Red', 'Green', 'Blue']

    def backtrack(assignment):

        if len(assignment) == len(map):

            return assignment

        var = select\_unassigned\_variable(assignment)

        for color in colors:

            if is\_consistent(var, color, assignment, map):

                new\_assignment = assignment.copy()

                new\_assignment[var] = color

                result = backtrack(new\_assignment)

                if result is not None:

                    return result

        return None

    def select\_unassigned\_variable(assignment):

        # MRV heuristic

        unassigned = [v for v in map if v not in assignment]

        return min(unassigned, key=lambda v: len(get\_legal\_colors(v, assignment, map)))

    def get\_legal\_colors(var, assignment, map):

        used\_colors = {assignment[n] for n in map[var] if n in assignment}

        return [c for c in colors if c not in used\_colors]

    def is\_consistent(var, color, assignment, map):

        for neighbor in map[var]:

            if neighbor in assignment and assignment[neighbor] == color:

                return False

        return True

    solution = backtrack({})

    if solution:

        print("Map coloring solution:")

        for region, color in solution.items():

            print(f"{region}: {color}")

    else:

        print("No solution found")

solve\_map\_coloring()

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

import math

# Initialize board

def create\_board():

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

# Print board

def print\_board(board):

    for row in board:

        print('|'.join(row))

        print('-' \* 5)

# Check for winner

def check\_winner(board):

    # Check rows and columns

    for i in range(3):

        if board[i][0] == board[i][1] == board[i][2] != ' ':

            return board[i][0]

        if board[0][i] == board[1][i] == board[2][i] != ' ':

            return board[0][i]

    # Check diagonals

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

        return board[0][0]

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

        return board[0][2]

    return None

# Check if board is full

def is\_full(board):

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

# Minimax function

def minimax(board, is\_maximizing):

    winner = check\_winner(board)

    if winner == 'X':

        return 1

    elif winner == 'O':

        return -1

    elif is\_full(board):

        return 0

    if is\_maximizing:

        best\_score = -math.inf

        for i in range(3):

            for j in range(3):

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

                    board[i][j] = 'X'

                    score = minimax(board, False)

                    board[i][j] = ' '

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

        return best\_score

    else:

        best\_score = math.inf

        for i in range(3):

            for j in range(3):

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

                    board[i][j] = 'O'

                    score = minimax(board, True)

                    board[i][j] = ' '

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

        return best\_score

# Best move for AI

def best\_move(board):

    best\_score = -math.inf

    move = (-1, -1)

    for i in range(3):

        for j in range(3):

            if board[i][j] == ' ':

                board[i][j] = 'X'

                score = minimax(board, False)

                board[i][j] = ' '

                if score > best\_score:

                    best\_score = score

                    move = (i, j)

    return move

# Game loop

def play\_game():

    board = create\_board()

    print("Welcome to Tic Tac Toe! You are O, AI is X.")

    print\_board(board)

    while True:

        # Human move

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

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

        if board[row][col] != ' ':

            print("Invalid move! Try again.")

            continue

        board[row][col] = 'O'

        if check\_winner(board) == 'O':

            print\_board(board)

            print("You win!")

            break

        if is\_full(board):

            print\_board(board)

            print("It's a draw!")

            break

        # AI move

        ai\_row, ai\_col = best\_move(board)

        board[ai\_row][ai\_col] = 'X'

        print("AI plays:")

        print\_board(board)

        if check\_winner(board) == 'X':

            print("AI wins!")

            break

        if is\_full(board):

            print("It's a draw!")

            break

# Run the game

play\_game()

**8.Implement NLP problems statements by HackerRank for following:**

**a. POS Tagging –**

**b. Similarity Score –**

from difflib import SequenceMatcher

# Take user input

str1 = input("Enter the first sentence: ")

str2 = input("Enter the second sentence: ")

# Calculate similarity ratio

similarity = SequenceMatcher(None, str1, str2).ratio()

# Display score

print(f"\nSimilarity Score: {similarity:.2f}")

**c. Spell Checker-**

from textblob import TextBlob

# Take user input

text = input("Enter text for spell checking: ")

# Perform spell correction

blob = TextBlob(text)

corrected\_text = blob.correct()

# Display result

print(f"\nCorrected Text: {corrected\_text}")