1) Problem Statement: Identify and Implement heuristic and search strategy for Travelling Salesperson Problem.

```
import copy

inf = float('inf')

class TSP_AI:
    # Travelling Salesman Problem Using Nearest Neighbor
    def __init__(self, city_matrix=None, source=0):
        self.city_matrix = [[0]*4 for _ in range(4)] if city_matrix is None else city_matrix
        self.n = len(self.city_matrix)
        self.source = source

    def Input(self):
        self.n = int(input('Enter city count: '))
        self.city_matrix = []
        for i in range(self.n):
            row = []
            for j in range(self.n):
                if i == j:
                    row.append(inf)
                else:
                    row.append(int(input(f'Cost to travel from city {i+1} to {j+1}: ')))
            self.city_matrix.append(row)
        self.source = int(input('Source: ')) % self.n

    def solve(self):
        minCost = inf
        for i in range(self.n):
            print("Path", end='')
```

```
            cost = self._solve(copy.deepcopy(self.city_matrix), i, i)
            print(f" -> {i+1} : Cost = {cost}")
            if cost and cost < minCost:
                minCost = cost
        return minCost

    def _solve(self, city_matrix, currCity=0, source=0):
        if self.n < 2:
            return 0

        print(f" -> {currCity+1}", end='')

        for i in range(self.n):
            city_matrix[i][currCity] = inf

        currMin = inf
        currMinPos = -1

        for j in range(self.n):
            if currMin > city_matrix[currCity][j]:
                currMin = city_matrix[currCity][j]
                currMinPos = j

        if currMin == inf:
            return self.city_matrix[currCity][source]

        # Mark the visited edge as infinity to prevent revisiting
        city_matrix[currCity][currMinPos] = city_matrix[currMinPos][currCity] = inf

        return currMin + self._solve(city_matrix, currMinPos, source)
```

```
# Test the class
if __name__ == '__main__':
    city_matrix = [
        [inf, 20, 15, 10],
        [20, inf, 45, 25],
        [15, 45, inf, 40],
        [10, 25, 40, inf]
    ]
    source_city = 0
    tsp = TSP_AI(city_matrix, source_city)
    print(f"Optimal Cost: {tsp.solve()}")
```

//

2) Problem Statement: Implement N-Queens Problem as Constraints Satisfaction Problem.

```
# Function to check if two queens threaten each other
def isSafe(mat, r, c):
    # Check column
    for i in range(r):
        if mat[i][c] == 'Q':
            return False

    # Check `\` diagonal (top-left to bottom-right)
    i, j = r, c
    while i >= 0 and j >= 0:
```

```
        if mat[i][j] == 'Q':
            return False
        i -= 1
        j -= 1

    # Check `/` diagonal (top-right to bottom-left)
    i, j = r, c
    while i >= 0 and j < len(mat):
        if mat[i][j] == 'Q':
            return False
        i -= 1
        j += 1

    return True

# Function to print the solution
def printSolution(mat):
    for row in mat:
        print(' '.join(row))
    print()

# Recursive function to solve N-Queen problem
def nQueen(mat, r):
    # Base condition: if all queens are placed
    if r == len(mat):
        printSolution(mat)
        return

    for i in range(len(mat)):
        if isSafe(mat, r, i):
            mat[r][i] = 'Q'       # Place queen
```

```
            nQueen(mat, r + 1)     # Recur for next row
            mat[r][i] = '–'        # Backtrack

# Driver code
if __name__ == '__main__':
    N = 4  # Size of the board
    mat = [['–' for _ in range(N)] for _ in range(N)]
    nQueen(mat, 0)
```

///

3) Problem Statement: Implement Water-Jug Problem using Rule Based Reasoning Technique.

```
def pour(jug1, jug2, visited):
    max1, max2, target = 3, 4, 2

    # If the current state is already visited, skip it
    if (jug1, jug2) in visited:
        return
    visited.add((jug1, jug2))

    print(f"{jug1}\t{jug2}")

    # If either jug has the target amount, stop
    if jug1 == target or jug2 == target:
        print("Reached the goal!")
        return
```

```
    # Fill Jug1
    pour(max1, jug2, visited)

    # Fill Jug2
    pour(jug1, max2, visited)

    # Empty Jug1
    pour(0, jug2, visited)

    # Empty Jug2
    pour(jug1, 0, visited)

    # Pour Jug1 -> Jug2
    transfer = min(jug1, max2 - jug2)
    pour(jug1 - transfer, jug2 + transfer, visited)

    # Pour Jug2 -> Jug1
    transfer = min(jug2, max1 - jug1)
    pour(jug1 + transfer, jug2 - transfer, visited)

# Driver code
print("JUG1\tJUG2")
pour(0, 0, set())
```

//

4) Problem Statement: Write a program for the Information Retrieval System using appropriate NLP tools (such as NLTK, Open NLP, ...) a. Text tokenization b. Count word frequency c. Remove stop words d. POS tagging

```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk import pos_tag
from collections import Counter
import string

# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger')

def tokenize_text(text):
    tokens = word_tokenize(text.lower())
    tokens = [word for word in tokens if word.isalpha()]  # Remove punctuation/numbers
    return tokens

def count_word_frequency(tokens):
    return Counter(tokens)

def remove_stop_words(tokens):
    stop_words = set(stopwords.words('english'))
    return [word for word in tokens if word not in stop_words]

def pos_tagging(tokens):
    return pos_tag(tokens)

# Take user input
text = input("Enter a sentence: ")

tokens = tokenize_text(text)
```

```
word_freq = count_word_frequency(tokens)
filtered_tokens = remove_stop_words(tokens)
tagged_tokens = pos_tagging(filtered_tokens)

print("\nTokens:", tokens)
print("\nWord Frequency:", dict(sorted(word_freq.items(), key=lambda x: x[1], reverse=True)))
print("\nFiltered Tokens:", filtered_tokens)
print("\nPOS Tagged Tokens:", tagged_tokens)
```

//

4) Problem Statement: Write a program for the Tic-Tac-Toe game using mini-max algorithm

```
import math

# Board setup
board = [' ' for _ in range(9)]

# Function to print the board
def print_board():
  for row in [board[i*3:(i+1)*3] for i in range(3)]:
    print('| ' + ' | '.join(row) + ' |')

# Check for winner
def is_winner(brd, player):
  win_conditions = [
    [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 condition in win_conditions:
```

```
        if all(brd[i] == player for i in condition):
            return True
    return False

# Check if board is full
def is_full(brd):
    return ' ' not in brd

# Get available moves
def get_available_moves(brd):
    return [i for i in range(9) if brd[i] == ' ']

# Minimax Algorithm
def minimax(brd, depth, is_maximizing):
    if is_winner(brd, 'O'):
        return 1
    elif is_winner(brd, 'X'):
        return -1
    elif is_full(brd):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for move in get_available_moves(brd):
            brd[move] = 'O'
            score = minimax(brd, depth + 1, False)
            brd[move] = ' '
            best_score = max(score, best_score)
        return best_score
    else:
        best_score = math.inf
```

```
        for move in get_available_moves(brd):
            brd[move] = 'X'
            score = minimax(brd, depth + 1, True)
            brd[move] = ' '
            best_score = min(score, best_score)
        return best_score

# AI makes move
def ai_move():
    best_score = -math.inf
    best_move = None
    for move in get_available_moves(board):
        board[move] = 'O'
        score = minimax(board, 0, False)
        board[move] = ' '
        if score > best_score:
            best_score = score
            best_move = move
    board[best_move] = 'O'

# Play game
def play_game():
    print("Welcome to Tic Tac Toe!")
    print_board()

    while True:
        # Human move
        try:
            move = int(input("Enter your move (1-9): ")) - 1
        except ValueError:
            print("Invalid input. Please enter a number from 1 to 9.")
```

```
            continue

        if move < 0 or move >= 9 or board[move] != ' ':
            print("Invalid move, try again.")
            continue

        board[move] = 'X'
        print_board()

        if is_winner(board, 'X'):
            print("You win!")
            break
        elif is_full(board):
            print("It's a tie!")
            break

        # AI move
        print("AI is making a move...")
        ai_move()
        print_board()

        if is_winner(board, 'O'):
            print("AI wins!")
            break
        elif is_full(board):
            print("It's a tie!")
            break

# Run the game
if __name__ == "__main__":
    play_game()
```