1. **Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem**

Program:

```
def water_jug_dfs(jug1_capacity, jug2_capacity, target_capacity):
    def dfs(jug1, jug2, path):
        if jug1 == target_capacity or jug2 == target_capacity:
            print("Solution found:", path)
            return

        # Fill jug1
        if jug1 < jug1_capacity:
            new_jug1 = jug1_capacity
            new_jug2 = jug2
            if (new_jug1, new_jug2) not in visited:
                visited.add((new_jug1, new_jug2))
                dfs(new_jug1, new_jug2, path + f"Fill Jug1\n")

        # Fill jug2
        if jug2 < jug2_capacity:
            new_jug1 = jug1
            new_jug2 = jug2_capacity
            if (new_jug1, new_jug2) not in visited:
                visited.add((new_jug1, new_jug2))
                dfs(new_jug1, new_jug2, path + f"Fill Jug2\n")

        # Pour water from jug1 to jug2
        if jug1 > 0 and jug2 < jug2_capacity:
            pour_amount = min(jug1, jug2_capacity - jug2)
            new_jug1 = jug1 - pour_amount
            new_jug2 = jug2 + pour_amount
            if (new_jug1, new_jug2) not in visited:
                visited.add((new_jug1, new_jug2))
                dfs(new_jug1, new_jug2, path + f"Pour Jug1 into Jug2\n")

        # Pour water from jug2 to jug1
        if jug2 > 0 and jug1 < jug1_capacity:
            pour_amount = min(jug2, jug1_capacity - jug1)
            new_jug1 = jug1 + pour_amount
            new_jug2 = jug2 - pour_amount
            if (new_jug1, new_jug2) not in visited:
                visited.add((new_jug1, new_jug2))
                dfs(new_jug1, new_jug2, path + f"Pour Jug2 into Jug1\n")

        # Empty jug1
        if jug1 > 0:
```

```
            new_jug1 = 0
            new_jug2 = jug2
            if (new_jug1, new_jug2) not in visited:
                visited.add((new_jug1, new_jug2))
                dfs(new_jug1, new_jug2, path + f"Empty Jug1\n")

        # Empty jug2
        if jug2 > 0:
            new_jug1 = jug1
            new_jug2 = 0
            if (new_jug1, new_jug2) not in visited:
                visited.add((new_jug1, new_jug2))
                dfs(new_jug1, new_jug2, path + f"Empty Jug2\n")

    visited = set()
    dfs(0, 0, "")

# Example usage:
jug1_capacity = 4
jug2_capacity = 3
target_capacity = 2

water_jug_dfs(jug1_capacity, jug2_capacity, target_capacity)
```

2. Implement and Demonstrate Best First Search Algorithm on any AI problem

Program:

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]

# Function For Implementing Best First Search
# Gives output path having lowest cost


def best_first_search(actual_Src, target, n):
    visited = [False] * n
    pq = PriorityQueue()
    pq.put((0, actual_Src))
    visited[actual_Src] = True

    while pq.empty() == False:
        u = pq.get()[1]
        # Displaying the path having lowest cost
        print(u, end=" ")
        if u == target:
            break

        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()

# Function for adding edges to graph


def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))


# The nodes shown in above example(by alphabets) are
# implemented using integers addedge(x,y,cost);
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
```

```
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)

source = 0
target = 9
best_first_search(source, target, v)
```

3. Implement AO* Search algorithm.

Program:

```
class Graph:
    def __init__(self, graph, heuristicNodeList, startNode): #instantiate graph object with graph topology, heuristic values, start node
        self.graph = graph
        self.H=heuristicNodeList
        self.start=startNode
        self.parent={}
        self.status={}
        self.solutionGraph={}

    def applyAOStar(self): # starts a recursive AO* algorithm
        self.aoStar(self.start, False)

    def getNeighbors(self, v): # gets the Neighbors of a given node
        return self.graph.get(v,'')

    def getStatus(self,v): # return the status of a given node
        return self.status.get(v,0)

    def setStatus(self,v, val): # set the status of a given node
        self.status[v]=val

    def getHeuristicNodeValue(self, n):
        return self.H.get(n,0) # always return the heuristic value of a given node

    def setHeuristicNodeValue(self, n, value):
        self.H[n]=value # set the revised heuristic value of a given node

    def printSolution(self):
        print("FOR GRAPH SOLUTION, TRAVERSE THE GRAPH FROM THE START NODE:",self.start)
        print("------------------------------------------------------------")
        print(self.solutionGraph)
        print("------------------------------------------------------------")

    def computeMinimumCostChildNodes(self, v): # Computes the Minimum Cost of child nodes of a given node v
        minimumCost=0
        costToChildNodeListDict={}
        costToChildNodeListDict[minimumCost]=[]
        flag=True
```

```
        for nodeInfoTupleList in self.getNeighbors(v): # iterate over all the set of child node/s
            cost=0
            nodeList=[]
            for c, weight in nodeInfoTupleList:
                cost=cost+self.getHeuristicNodeValue(c)+weight
                nodeList.append(c)
            if flag==True: # initialize Minimum Cost with the cost of first set of child node/s
                minimumCost=cost
                costToChildNodeListDict[minimumCost]=nodeList # set the Minimum Cost child
node/s
                flag=False
            else: # checking the Minimum Cost nodes with the current Minimum Cost
                if minimumCost>cost:
                    minimumCost=cost
                    costToChildNodeListDict[minimumCost]=nodeList # set the Minimum Cost child
node/s
        return minimumCost, costToChildNodeListDict[minimumCost] # return Minimum Cost and
Minimum Cost child node/s

    def aoStar(self, v, backTracking): # AO* algorithm for a start node and backTracking status
flag
        print("HEURISTIC VALUES :", self.H)
        print("SOLUTION GRAPH :", self.solutionGraph)
        print("PROCESSING NODE :", v)
        print("-----------------------------------------------------------------------------------------")
        if self.getStatus(v) >= 0: # if status node v >= 0, compute Minimum Cost nodes of v
            minimumCost, childNodeList = self.computeMinimumCostChildNodes(v)
            print(minimumCost, childNodeList)
            self.setHeuristicNodeValue(v, minimumCost)
            self.setStatus(v,len(childNodeList))
            solved=True # check the Minimum Cost nodes of v are solved
            for childNode in childNodeList:
                self.parent[childNode]=v
                if self.getStatus(childNode)!=-1:
                    solved=solved & False
            if solved==True: # if the Minimum Cost nodes of v are solved, set the current node status
as solved(-1)
                self.setStatus(v,-1)
                self.solutionGraph[v]=childNodeList # update the solution graph with the solved nodes
which may be a part of solution
            if v!=self.start: # check the current node is the start node for backtracking the current node
value
                self.aoStar(self.parent[v], True) # backtracking the current node value with
backtracking status set to true
            if backTracking==False: # check the current call is not for backtracking
```

```
        for childNode in childNodeList: # for each Minimum Cost child node
            self.setStatus(childNode,0) # set the status of child node to 0(needs exploration)
            self.aoStar(childNode, False) # Minimum Cost child node is further explored with backtracking status as false

h1 = {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2, 'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}
graph1 = {
    'A': [[('B', 1), ('C', 1)], [('D', 1)]],
    'B': [[('G', 1)], [('H', 1)]],
    'C': [[('J', 1)]],
    'D': [[('E', 1), ('F', 1)]],
    'G': [[('I', 1)]]
}

G1= Graph(graph1, h1, 'A')
G1.applyAOStar()
G1.printSolution()
```

4. Solve 8-Queens Problem with suitable assumptions

Program:

```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())
# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]
def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
def N_queens(n):
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            if (not(attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                if N_queens(n-1)==True:
                    return True
                board[i][j] = 0
    return False
N_queens(N)
for i in board:
    print (i)
```

5. Implementation of TSP using heuristic approach

```
import math

# Define a function to calculate the Euclidean distance between two points
def distance(point1, point2):
    return math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)

# Define the Nearest Neighbor algorithm
def nearest_neighbor(points):
    n = len(points)
    unvisited = set(range(n))
    tour = [0]  # Start from the first point
    unvisited.remove(0)

    while unvisited:
        current_point = tour[-1]
        nearest_point = min(unvisited, key=lambda x: distance(points[current_point], points[x]))
        tour.append(nearest_point)
        unvisited.remove(nearest_point)

    # Complete the tour by returning to the starting point
    tour.append(tour[0])

    return tour

# Example usage
if __name__ == "__main__":
    # Define the points as (x, y) coordinates
    points = [(0, 0), (1, 2), (2, 3), (3, 4),(4,2)]

    # Find the tour using the Nearest Neighbor algorithm
    tour = nearest_neighbor(points)

    print("Optimal Tour:", tour)
```

6. Implementation of the problem-solving strategies: either using Forward Chaining or Backward Chaining

Forward Chaining Program:

```
class Rule:
    def __init__(self, antecedents, consequent):
        self.antecedents = antecedents
        self.consequent = consequent

class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def apply_forward_chaining(self):
        new_facts_derived = True
        while new_facts_derived:
            new_facts_derived = False
            for rule in self.rules:
                if all(antecedent in self.facts for antecedent in rule.antecedents) and rule.consequent not in self.facts:
                    self.facts.add(rule.consequent)
                    new_facts_derived = True

if __name__ == "__main__":
    kb = KnowledgeBase()

    # Define rules and facts
    rule1 = Rule(["A", "C"], "E")
    rule2 = Rule(["A", "E"], "G")
    rule3 = Rule(["B"], "E")
    rule4 = Rule(["G"], "D")
    kb.add_rule(rule1)
    kb.add_rule(rule2)
    kb.add_rule(rule3)
    kb.add_rule(rule4)
    kb.add_fact("A")
    kb.add_fact("C")
```

```
    # Apply forward chaining
    kb.apply_forward_chaining()

    # Print the derived facts
    print("Derived Facts:", kb.facts)
```

Backward Chaining Program:

```
# Define the knowledge base as a dictionary of rules

knowledge_base = {

    "rule1": {

        "if": ["A", "B"],

        "then": "C"

    },

    "rule2": {

        "if": ["D"],

        "then": "A"

    },

    "rule3": {

        "if": ["E"],

        "then": "B"

    },

    "rule4": {

        "if": ["F"],

        "then": "D"

    },

    "rule5": {

        "if": ["G"],

        "then": "E"

    }

}
```

```
# Define a function to perform backward chaining
def backward_chaining(goal, known_facts):
    if goal in known_facts:
        return True
    for rule, value in knowledge_base.items():
        if goal in value["if"]:
            all_conditions_met = all(condition in known_facts for condition in value["if"])
            if all_conditions_met and backward_chaining(value["then"], known_facts):
                return True
    return False
# Define the goal and known facts
goal = "C"
known_facts = ["G", "F", "E"]

# Check if the goal can be reached using backward chaining
if backward_chaining(goal, known_facts):
    print(f"The goal '{goal}' can be reached.")
else:
    print(f"The goal '{goal}' cannot be reached.")
```

8. Implement K- means algorithm.

```
import numpy as np
```

```
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Generate some sample data for clustering
np.random.seed(0)
X = np.random.rand(100, 2)

# Number of clusters (k)
k = 3

# Initialize the KMeans model
kmeans = KMeans(n_clusters=k)

# Fit the model to the data
kmeans.fit(X)

# Get cluster centers and labels
cluster_centers = kmeans.cluster_centers_
labels = kmeans.labels_

# Plot the data points and cluster centers
plt.scatter(X[:, 0], X[:, 1], c=labels)
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1], marker='x', s=200, color='red')
plt.title(f'K-Means Clustering (k={k})')
plt.show()
```

9. Implement K- nearest neighbour algorithm

```
import numpy as np
```

```
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Generate some sample data for classification
np.random.seed(0)
X = np.random.rand(100, 2)  # Feature matrix
y = np.random.choice([0, 1], size=100)  # Target vector (binary classification)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a K-Nearest Neighbors classifier with k=3
k = 3
knn_classifier = KNeighborsClassifier(n_neighbors=k)

# Fit the classifier to the training data
knn_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = knn_classifier.predict(X_test)

# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')
```

10. Implement SVM

```
import numpy as np
```

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Create a synthetic dataset for classification (you can replace this with your own dataset)
X, y = datasets.make_classification(n_samples=500, n_features=3, n_informative=2,
n_redundant=0, random_state=42)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create an SVM classifier
svm_classifier = SVC(kernel='linear', C=1.0)

# Train the SVM classifier on the training data
svm_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```