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()
```