**Program 1**

**A\* search Algorithm**

def aStarAlgo(start\_node, stop\_node):

open\_set = set(start\_node)

closed\_set = set()

g = {} #store distance from starting node

parents = {}# parents contains an adjacency map of all nodes

#ditance of starting node from itself is zero

g[start\_node] = 0

#start\_node is root node i.e it has no parent nodes

#so start\_node is set to its own parent node

parents[start\_node] = start\_node

while len(open\_set) > 0:

n = None

#node with lowest f() is found

for v in open\_set:

if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

n = v

if n == stop\_node or Graph\_nodes[n] == None:

pass

else:

for (m, weight) in get\_neighbors(n):

#nodes 'm' not in first and last set are added to first

#n is set its parent

if m not in open\_set and m not in closed\_set:

open\_set.add(m)

parents[m] = n

g[m] = g[n] + weight

#for each node m,compare its distance from start i.e g(m) to the

#from start through n node

else:

if g[m] > g[n] + weight:

#update g(m)

g[m] = g[n] + weight

#change parent of m to n

parents[m] = n

#if m in closed set,remove and add to open

if m in closed\_set:

closed\_set.remove(m)

open\_set.add(m)

if n == None:

print('Path does not exist!')

return None

# if the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

if n == stop\_node:

path = []

while parents[n] != n:

path.append(n)

n = parents[n]

path.append(start\_node)

path.reverse()

print('Path found: {}'.format(path))

return path

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_set.remove(n)

closed\_set.add(n)

print('Path does not exist!')

return None

#define fuction to return neighbor and its distance

#from the passed node

def get\_neighbors(v):

if v in Graph\_nodes:

return Graph\_nodes[v]

else:

return None

#for simplicity we ll consider heuristic distances given

#and this function returns heuristic distance for all nodes

def heuristic(n):

H\_dist = {

'A': 11,

'B': 6,

'C': 99,

'D': 1,

'E': 7,

'G': 0,

}

return H\_dist[n]

#Describe your graph here

Graph\_nodes = {

'A': [('B', 2), ('E', 3)],

'B': [('C', 1),('G', 9)],

'C': None,

'E': [('D', 6)],

'D': [('G', 1)],

}

aStarAlgo('A', 'G')

Program 2

A0\*

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)

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, 'T': 3}

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()

h2 = {'A': 1, 'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7} # Heuristic values of Nodes

graph2 = { # Graph of Nodes and Edges

'A': [[('B', 1), ('C', 1)], [('D', 1)]], # Neighbors of Node 'A', B, C & D with repective weights

'B': [[('G', 1)], [('H', 1)]], # Neighbors are included in a list of lists

'D': [[('E', 1), ('F', 1)]] # Each sublist indicate a "OR" node or "AND" nodes

}

G2 = Graph(graph2, h2, 'A') # Instantiate Graph object with graph, heuristic values and start Node

G2.applyAOStar() # Run the AO\* algorithm

G2.printSolution() # Print the solution graph as output of the AO\* algorithm search