

Lab Program 2: AO* Search 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()

```

```

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

```