**Implement AO\* Search algorithm.**

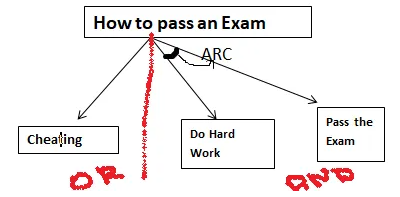
**AO\* Algorithm**

AO\* Algorithm is basically based on  problem decomposition (Breakdown problem into small pieces)

When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, **AND-OR graphs**or **AND - OR trees**are used for representing the solution.

The decomposition of the problem or problem reduction generates AND arcs.

**AND-OR Graph**



**The figure shows an AND-OR graph**

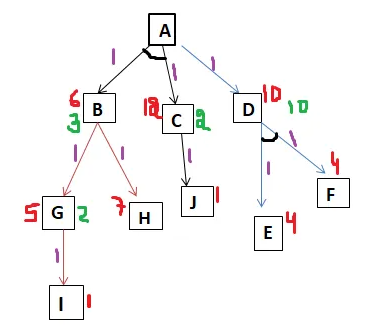
1. To pass any exam, we have two options, either cheating or hard work.
2. In this graph we are given two choices, first do cheating **or (The red line)**work hard and **(The arc)**pass.
3. When we have more than one choice and we have to pick one, we apply **OR condition**to choose one. (That's what we did here).
4. Basically the **ARC**here denotes**AND condition**.
5. Here we have replicated the arc between the work hard and the pass because by doing the hard work possibility of passing an exam is more than cheating.

**A\* Vs AO\***

1. Both are part of informed search technique and use heuristic values to solve the problem.
2. The solution is guaranteed in both algorithms.
3. A\* always gives an **optimal solution** (shortest path with low cost) But It is not guaranteed to that**AO\***always provide **an optimal solutions**.
4. **Reason:** Because AO\* does not explore the entire solution path once it got solution.

### ****How AO\* works****

Let's try to understand it with the following diagram



The algorithm always moves towards a **lower cost value.**

Basically, We will calculate the **cost function** here **(F(n)= G (n) + H (n))**

**H:** **heuristic/ estimated**value of the nodes. And**G:**actual cost or edge value (here unit value).

Here we have taken the **edges value 1,** meaning we have to focus solely on the **heuristic value.**

1. The **Purple color**values are **edge values (here all are same that is one).**
2. The **Red color**values are **Heuristic values for nodes.**
3. **The Green color**values are**New Heuristic values for nodes.**

**Procedure:**

1. In the above diagram we have two ways from**A to D**or**A to B-C**(because of and condition). calculate cost to select a path
2. **F(A-D)= 1+10 = 11**            and               **F(A-BC) = 1 + 1 + 6 +12 = 20**
3. As we can see**F(A-D)**is less than **F(A-BC)**then the algorithm choose the path **F(A-D).**
4. Form D we have one choice that is **F-E.**
5. **F(A-D-FE) = 1+1+ 4 +4 =10**
6. Basically **10** is the cost of reaching **FE from D.** And **Heuristic value of node D** also denotes the cost of reaching **FE from D**. So, the new Heuristic value of D is 10.
7. And the Cost from A-D remains same that is **11**.

Suppose we have searched this path and we have got the **Goal State**, then we will never explore the other path. (This is what AO\* says but here we are going to explore other path as well to see what happen)

**Let's Explore the other path:**

1. In the above diagram we have two ways from**A to D**or**A to B-C**(because of and condition). calculate cost to select a path
2. **F(A-D)= 1+10 = 11**            and               **F(A-BC) = 1 + 1 + 6 +12 = 20**
3. As we know the cost is more of **F(A-BC)**but let's take a look
4. Now from B we have two path G and H , let's calculate the cost
5. **F(B-G)= 5+1 =6**  and **F(B-H)= 7 + 1 = 8**
6. So, cost from**F(B-H)** is more than **F(B-G)**we will take the path B-G.
7. The Heuristic value fro**m G to I is 1 but let's calculate the cost form G to I.**
8. **F(G-I) = 1 +1 = 2 w**hich is less than **Heuristic value 5**. So, the new**Heuristic value from G to I is 2.**
9. If it is a new value, then the cost from**G to B**must also have changed. Let's see the new **cost form (B to G)**
10. F(B-G)= 1+2 =3 . Mean the**New Heuristic value of B is 3.**
11. **But A is associated with both B and C .**
12. As we can see from the diagram**C only have one choice or one node to explore that is J.** The Heuristic value of C is 12.
13. Cost form C to **J= F(C-J) = 1+1= 2** Which is less than Heuristic value
14. No**w the New Heuristic value of C is 2.**
15. **And the New Cost from A- BC that is F(A-BC) = 1+1+2+3 = 7 which is less than F(A-D)=11.**
16. In this case Choosing path A-BC is more cost effective and good than that of A-D.

But this will only happen when the algorithm explores this path as well. But according to the algorithm, algorithm will not accelerate this path **(here we have just did it to see how the other path can also be correct).**

But it is not the case in all the cases that it will happen in some cases that the algorithm will get optimal solution.

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

print ("Graph - 2")

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