**AO Star Algorithm**

**Tabish, CSE-09/16**

**Department of Computer Science & Engineering, NIT Srinagar**

**Introduction to the Algorithm:**

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. One AND are may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND.

In AO\* algorithm expanded nodes are re-examined so that the current best path can be selected.The AO\* algorithm uses a single structure G which represents the part of the search graph generated so far. Each node in G points down to its immediate successors and up to its immediate predecessors, and also has with it the value of h' cost of a path from itself to a set of solution nodes. The cost of getting from the start nodes to the current node "g" is not stored as in the A\* algorithm. This is because it is not possible to compute a single such value since there may be many paths to the same state. In AO\* algorithm serves as the estimate of goodness of a node.

*F(y) =g(y) +h(y)*

*G(y)* is cost to reach the current state and *H(y)* is the cost of reaching goal state from current state.

**Algorithm:**

1. Place the start node on open.

2. Using the search tree, compute the most promising solution tree TP.

3. Select node n that is both on open and a part of tp, remove n from open and place it no closed.

4. If n is a goal node, label n as solved. If the start node is solved, exit with success where tp is the solution tree, remove all nodes from open with a solved ancestor.

5. If n is not solvable node, label n as unsolvable. If the start node is labelled as unsolvable, exit with failure. Remove all nodes from open, with unsolvable ancestors.

6. Otherwise, expand node n generating all of its successor compute the cost of for each newly generated node and place all such nodes on open.

7. Go back to step(2)

To search an AND-OR graph, the following three things must be done.

1. Traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.
2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ' (cost of the remaining distance) for each of them.
3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

**Code:**

import networkx as nx

costs = {'n0': 0 , 'n1':2 , 'n2':4,'n3':4,'n4':1,'n5':1,'n6':2,'n7':0,'n8':0}

goalNodes = ['n7','n8']

def makeG(mg):

for i in mg.nodes:

mg.node[i]['solved']=False

mg.node[i]['cost'] = costs[i]

for i in mg.edges:

mg[i[0]][i[1]]['marked']=False

mg[i[0]][i[1]]['andby']=None

def cost(mg,node):

return nx.get\_node\_attributes(mg,'cost')[node]

def solved(mg,node):

return nx.get\_node\_attributes(mg,'solved')[node]

def andby(mg,edge1,edge2):

return nx.get\_edge\_attributes(mg,'andby')[edge1,edge2]

def marked(mg,edge1,edge2):

return nx.get\_edge\_attributes(mg,'marked')[edge1,edge2]

def findleaf(g):

for i in g.nodes:

if not g.\_\_getitem\_\_(i):

if i not in goalNodes:

return i

return None

def addToG(n,nodeList):

for i in nodeList:

# print(i)

if not g.has\_edge(n,i):

g.add\_edge(n,i,marked=False,andby=andby(mainGraph,n,i))

g.node[i]['cost'] = cost(mainGraph,i)

g.node[i]['solved'] = False

if i in goalNodes :

g.node[i]['solved'] = True

def findMin(q,m):

ls = []

MinNode = min(q,key=q.get)

ls.append(MinNode)

if andby(g,m,MinNode):

ls.append(andby(g,m,MinNode))

return ls

mainGraph = nx.DiGraph()

startNode = 'n0'

edges = [

('n0','n1'),

('n0','n4'),

('n0','n5'),

('n1','n3'),

('n2','n4'),

('n2','n5'),

('n2','n3'),

('n3','n5'),

('n3','n6'),

('n4','n5'),

('n4','n8'),

('n5','n7'),

('n5','n8'),

('n6','n7'),

('n6','n8')

]

mainGraph.add\_edges\_from(edges)

makeG(mainGraph)

mainGraph.edges['n0','n4']['andby'] = 'n5'

mainGraph.edges['n0','n5']['andby'] = 'n4'

mainGraph.edges['n2','n4']['andby'] = 'n5'

mainGraph.edges['n2','n5']['andby'] = 'n4'

mainGraph.edges['n3','n5']['andby'] = 'n6'

mainGraph.edges['n3','n6']['andby'] = 'n5'

mainGraph.edges['n5','n7']['andby'] = 'n8'

mainGraph.edges['n5','n8']['andby'] = 'n7'

mainGraph.edges['n6','n7']['andby'] = 'n8'

mainGraph.edges['n6','n8']['andby'] = 'n7'

g = mainGraph.subgraph(startNode).copy()

if startNode in goalNodes :

g.node[startNode]['solved'] = True

while (solved(g,startNode) != True):

gprime = g.subgraph(startNode).copy()

for i in g.edges:

if marked(g,i[0],i[1]):

gprime.add\_edge(i[0],i[1])

n = findleaf(gprime)

nodeList = list(mainGraph.neighbors(n))

addToG(n,nodeList)

# print(g.nodes)

s = [n]

while (len(s) != 0):

for i in s:

if i not in list(g.neighbors(i)):

m = s.pop(s.index(i))

break

q = {}

for i in list(g.neighbors(m)):

if andby(g,m,i):

q[i] = 2 + costs[i]+costs[andby(g,m,i)]

else:

q[i] = 1 + costs[i]

markList = findMin(q,m)

for i in markList:

g.edges[m,i]['marked'] = True

for j in list(g.neighbors(m)):

if andby(g,m,j) != i and i != j:

g.edges[m,j]['marked'] = False

for a in list(g.neighbors(j)):

g.edges[j,a]['marked'] = False

checkNeighbors = True

for i in g.neighbors(m):

if marked(g,m,i):

if solved(g,i) == False:

checkNeighbors = False

if checkNeighbors:

g.node[m]['solved'] = True

if costs[m] != q[markList[0]]:

costs[m] = q[markList[0]]

for i in list(g.predecessors(m)):

if marked(g,i,m):

s.append(i)

if solved(g,m):

for i in list(g.predecessors(m)):

if marked(g,i,m):

s.append(i)

markedge = nx.get\_edge\_attributes(g,'marked')

print("The answer graph contain below edges:")

for i in markedge:

if markedge[i]:

print(i)

**Output Observed for different Inputs:**

* **Input passed:**

costs = {'n0': 0 , 'n1':2 , 'n2':4,'n3':4,'n4':1,'n5':1,'n6':2,'n7':0,'n8':0}

goalNodes = ['n7','n8']

* **Output Obtained:**

The answer graph contain below edges:

('n0', 'n4')

('n0', 'n5')

('n4', 'n8')

('n5', 'n7')

('n5', 'n8')

* **Input passed:**

costs = {'n0': 6 , 'n1':2 , 'n2':1,'n3':3,'n4':1,'n5':1,'n6':2,'n7':4,'n8':0}

goalNodes = ['n1','n5']

* **Output Obtained:**

The answer graph contain below edges:

('n0', 'n1')

* **Input passed:**

costs = {'n0': 2 , 'n1':0 , 'n2':2,'n3':4,'n4':1,'n5':1,'n6':2,'n7':3,'n8':0}

goalNodes = ['n7','n8']

* **Output Obtained:**

The answer graph contain below edges:

('n0', 'n4')

('n0', 'n5')

('n4', 'n8')

('n5', 'n7')

('n5', 'n8')

('n3', 'n5')

('n3', 'n6')