**Artificial Intelligence Lab**

**A\* Search Task**



**Ms. Saba Aslam**

Student Name: Abdul Salam Wasti

Roll No: 18F-0326 Section: CS(6B)

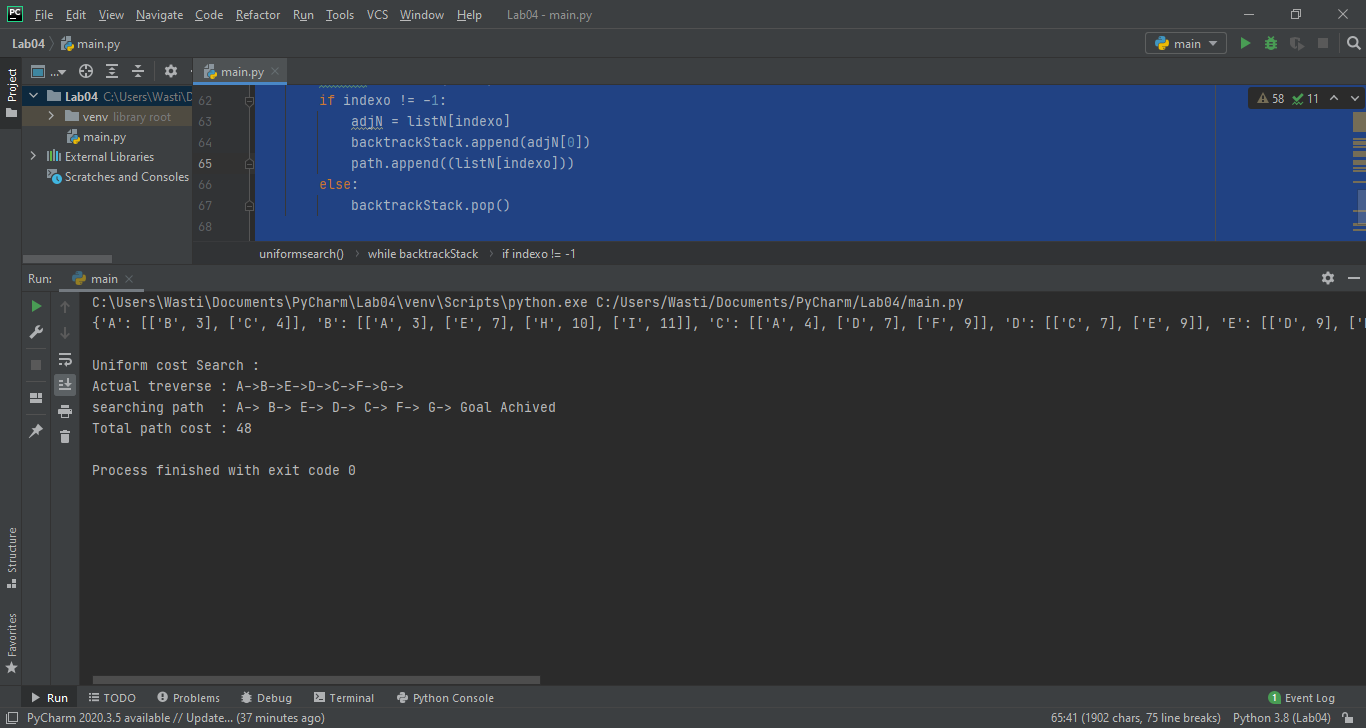
Date performed: 05/05/2021

# Task

# Code:

Uniform Cost search:

Graph = {  
 "A": [["B",3], ["C",4]],  
 "B": [["A",3], ["E",7], ["H",10], ["I",11]],  
 "C": [["A",4], ["D", 7], ["F",9]],  
 "D": [["C",7], ["E",9]],  
 "E": [["D",9], ["B",7], ["F",11], ["H",13]],  
 "F": [["C",9], ["E",11], ["G",13]],  
 "G": [["F",13], ["H",15], ["K",18]],  
 "H": [["B",10], ["E",13], ["G",15], ["K",19], ["L",20], ["J",18]],  
 "I": [["B",11], ["",19]],  
 "J": [["H",18], ["I",19]],  
 "K": [["H",19], ["G",18]],  
 "L": [["H",20]]  
}  
  
  
  
print(Graph)  
  
def minlol(lists):  
 minValue = 999  
 index = 0  
 Mindex = -1  
 while index < len(lists):  
 lol = lists[index]  
 if minValue > lol[1] :  
 minValue = lol[1]  
 Mindex = index  
 index +=1  
 return Mindex  
  
def uniformsearch(Graph,startingN,goalN):  
 listN = []  
 backtrackStack = []  
 visited = []  
 path = []  
  
 path.append([startingN,0])  
 pathcost = 0  
 print("Actual treverse :", end=" ")  
  
 backtrackStack.append(startingN)  
 while backtrackStack:  
 listN.clear()  
 Node = backtrackStack[-1]  
 print(Node + "->", end="")  
 visited.append(Node)  
 if Node == goalN:  
  
 print("\nsearching path :", end=" ")  
 for pathv in path:  
 pathcost = pathv[1] + pathcost  
 print(str(pathv[0]) + "->", end=" ")  
 print("Goal Achived")  
 print("Total path cost : " + str(pathcost))  
 break  
  
 for conectedNode in Graph[Node]:  
 if conectedNode[0] not in visited and conectedNode[0] not in backtrackStack:  
 listN.append(conectedNode)  
 indexo = minlol(listN)  
 if indexo != -1:  
 adjN = listN[indexo]  
 backtrackStack.append(adjN[0])  
 path.append((listN[indexo]))  
 else:  
 backtrackStack.pop()  
  
  
  
print("\nUniform cost Search :")  
uniformsearch(Graph,"A","G")

Screenshot: 

# Code:

A\* Search by updating BFS:

#18F-0326 Abdul Salam Wasti BFS updation to A\* Search  
from collections import defaultdict  
  
class Graph:  
  
  
 def \_\_init\_\_(self):  
 self.graph = defaultdict(list)  
  
 def addEdge(self, u, v):  
  
 self.graph[u].append(v)  
  
 def printGraph(self):  
 print("Given Graph",self.graph)  
  
 # Function to print a BFS of graph  
 def BFS(self, s, e):  
 print("This is our starting node", s)  
 print("This is our ending node",e)  
 visited=[]  
 queue = []  
 path=[]  
 queue.append(s)  
 visited.append(s)  
 while queue:  
 nodetoselect=g.findHeuristic(queue,huristic)  
 s=nodetoselect  
 path.append(s)  
 print("Node to Select",s)  
 indexofselectedvalue = queue.index(s)  
 queue.pop(indexofselectedvalue)  
 if s==e:  
 break  
 for i in self.graph[s]:  
 if i not in visited:  
 queue.append(i)  
 visited.append(i)  
 print("this is our final path",path)  
 g.findactualpathvalue(path)  
  
 def AStarSearch(self, s, e):  
 print("This is our starting node", s)  
 print("This is our ending node",e)  
 visited=[]  
 queue = []  
 path=[]  
 parentChild = [] #need parent child track to get path value  
 c = s  
 parentChild.append(s)  
 parentChild.append(c)  
 queue.append(parentChild)  
 visited.append(s)  
 while queue:  
 nodetoselect=g.findHeuristicAndPath(queue,huristic,actualvalues)  
 c=nodetoselect  
 path.append(c)  
 print("Node to Select",c)  
 parentChild.clear()  
 parentChild.append(s)  
 parentChild.append(c)  
 indexofselectedvalue = queue.index(parentChild)  
 queue.pop(indexofselectedvalue)  
 if c==e:  
 break  
 s = c  
 for i in self.graph[s]:  
 if i not in visited:  
 parentChild = []  
 parentChild.append(s)  
 parentChild.append(i)  
 queue.append(parentChild)  
 visited.append(i)  
  
 print("this is our final path",path)  
 g.findactualpathvalue(path)  
  
 def findactualpathvalue(self,mypath):  
  
 actualvaluequue=[]  
 cost=[]  
 for i in range(len(mypath)-1):  
 a=str(mypath[i])  
 b=str(mypath[i+1])  
 valuetoappend=(a+b)  
 print('from ', a,'to', b, end=' ')  
 actualvaluequue.append(valuetoappend)  
 print("this is the path to go",actualvaluequue)  
 for i in actualvaluequue:  
 if i in actualvalues.keys():  
 cost.append(actualvalues.get(i))  
 print('This is the actual cost', sum(cost))  
  
 def findHeuristic(self,pqueu,dich):  
  
 keyslist=[]  
 valueslist=[]  
 for i in pqueu:  
 if i in dich.keys():  
 keyslist.append(i)  
 valueslist.append(dich.get(i))  
 indexofvalue=valueslist.index(min(valueslist))  
 keyofminvalue=keyslist[indexofvalue]  
 print('Keys list:', keyslist)  
 print("Value list", valueslist)  
 return keyofminvalue  
  
 def findHeuristicAndPath(self,pqueu,dich, actualPath):  
 #function to compute path value + heuristic value  
   
 keyslist=[]  
 valueslist=[]  
 for i in pqueu:  
 if i[1] in dich.keys():   
 keyslist.append(i[1])  
 pathCost = 0  
 if actualPath.get( str(i[0]+i[1]) ): #getting pathvalue by selected node and parent node  
 pathCost = actualPath[str(i[0]+i[1])]  
 valueslist.append(dich.get(i[1]) + pathCost)  
  
 indexofvalue=valueslist.index(min(valueslist))  
 keyofminvalue=keyslist[indexofvalue]  
 print('Keys list:', keyslist)  
 print("Value list", valueslist)  
 return keyofminvalue  
  
# Driver code  
  
g = Graph()  
  
g.addEdge('A', 'B')  
g.addEdge('A', 'C')  
g.addEdge('A', 'D')  
g.addEdge('B', 'E')  
g.addEdge('C', 'E')  
g.addEdge('C', 'F')  
g.addEdge('D', 'F')  
g.addEdge('E', 'H')  
g.addEdge('F', 'G')  
g.addEdge('H', 'G')  
  
huristic=\  
 {'A': 40,  
 'B':32,  
 'C':25,  
 'D':35,  
 'E':19,  
 'F':17,  
 'H':10,  
 'G':0  
}  
  
actualvalues=\  
 {'AB':11,  
 'AC':14,  
 'AD':7,  
 'BE':15,  
 'CE':8,  
 'CF':10,  
 'DF':25,  
 'FG':20,  
 'EH':9,  
 'HG':10,  
 }  
  
g.printGraph()  
print("heuristic Values",huristic)  
print("Actual Cost of the graph", actualvalues)  
print("\n\n\t\tBFS\n---------------------------------------------\n")  
g.BFS('A','G')  
print("\n\n\t\tA\* Search\n---------------------------------------------\n")  
g.AStarSearch('A','G')

# Screenshot:

