Artificial Intelligence

Assignment 2(Maze)

|  |  |
| --- | --- |
| Name | ID |
| Noor Al-Fattha Tarek Wahdan | **20192107** |
| Youssef Ahmed El-Sayed | **20191800** |
| Omar AbdEl-Aziz | **20192801** |

Under Supervision TA/ Ayman Adel Iskandar

Group: A1.1 Monday from 1:00 To 3:00

A picture containing text

Description automatically generated

-

import queue  
'''Team Members  
Name / ID  
Noor al-Fattha Tarek / 20192107  
Omar AbdEL-Aziz / 20192801  
Youssef Ahmed El-Sayed / 20191800   
'''  
class Node:  
 id = None # Unique value for each node.  
 up = None # Represents value of neighbors (up, down, left, right).  
 down = None  
 left = None  
 right = None  
 previousNode = None # Represents value of neighbors.  
 edgeCost = None # Represents the cost on the edge from any parent to this node.  
 gOfN = None # Represents the total edge cost  
 hOfN = None # Represents the heuristic value  
 heuristicFn = None # Represents the value of heuristic function  
  
 def \_\_init\_\_(self, value):  
 self.value = value  
  
class SearchAlgorithms:  
 *''' \* DON'T change Class, Function or Parameters Names and Order  
 \* You can add ANY extra functions,  
 classes you need as long as the main  
 structure is left as is '''* path = [] # Represents the correct path from start node to the goal node.  
 fullPath = [] # Represents all visited nodes from the start node to the goal node.  
 totalCost = -1 # Represents the total cost in case using AStar (Euclidean)  
  
 mazeStr = None  
 maze2D = []  
  
 startNode = None  
 endNode = None  
  
 def \_\_init\_\_(self, mazeStr, edgeCost=None):  
 *''' mazeStr contains the full board  
 The board is read row wise,  
 the nodes are numbered 0-based starting  
 the leftmost node'''* self.mazeStr = mazeStr  
 pass  
  
 def Create2DMaze(self, mazeStr):  
 mazeRows = mazeStr.split(' ');  
 for x in mazeRows:  
 mazeCol = x.split(',')  
 self.maze2D.append(mazeCol)  
  
 initialID = 0;  
 for i in range(0, len(self.maze2D)):  
 for j in range(0, len(self.maze2D[i])):  
 tempNode = Node(self.maze2D[i][j])  
 tempNode.id = initialID  
 self.maze2D[i][j] = tempNode  
 initialID += 1  
 pass  
  
 def DefineNodesPositions(self):  
 for i in range(0, len(self.maze2D)):  
 for j in range(0, len(self.maze2D[i])):  
  
 if self.startNode is None or self.endNode is None:  
 if self.maze2D[i][j].value == 'S':  
 self.startNode = self.maze2D[i][j]  
 elif self.maze2D[i][j].value == 'E':  
 self.endNode = self.maze2D[i][j]  
  
 if i-1 >= 0:  
 self.maze2D[i][j].up = self.maze2D[i-1][j]  
 if i+1 < len(self.maze2D):  
 self.maze2D[i][j].down = self.maze2D[i+1][j]  
 if j-1 >= 0:  
 self.maze2D[i][j].left = self.maze2D[i][j-1]  
 if j+1 < len(self.maze2D[i]):  
 self.maze2D[i][j].right = self.maze2D[i][j+1]  
 pass  
  
 def PrintMaze(self, maze2D):  
 print('\n')  
 for i in range(0, len(maze2D)):  
 for j in range(0, len(maze2D[i])):  
 print(maze2D[i][j].value, end=" ")  
 print()  
 print('\n')  
 pass  
  
 def DFS(self):  
 # Fill the correct path in self.path  
 # self.fullPath should contain the order of visited nodes  
 visitedNodes = []  
 availableNodes = []  
 availableNodes.append(self.startNode)  
  
 while availableNodes:  
 currentNode = availableNodes.pop()  
 visitedNodes.append(currentNode)  
 self.fullPath.append(currentNode.id)  
 if currentNode.id == self.endNode.id:  
 current = currentNode  
 self.path.append(current.id)  
 for i in range(0, len(self.fullPath)):  
 if current.id != 0:  
 if current.up in visitedNodes and current.up.id not in self.path:  
 self.path.append(current.up.id)  
 current = current.up  
 elif current.down in visitedNodes and current.down.id not in self.path:  
 self.path.append(current.down.id)  
 current = current.down  
 elif current.left in visitedNodes and current.left.id not in self.path:  
 self.path.append(current.left.id)  
 current = current.left  
 elif current.right in visitedNodes and current.right.id not in self.path:  
 self.path.append(current.right.id)  
 current = current.right  
 self.path.reverse()  
 return self.path, self.fullPath  
  
 if currentNode.left is not None and currentNode.left not in availableNodes and currentNode.left not in visitedNodes and currentNode.left.value != '#':  
 availableNodes.append(currentNode.left)  
 if currentNode.right is not None and currentNode.right not in availableNodes and currentNode.right not in visitedNodes and currentNode.right.value != '#':  
 availableNodes.append(currentNode.right)  
 if currentNode.up is not None and currentNode.up not in availableNodes and currentNode.up not in visitedNodes and currentNode.up.value != '#':  
 availableNodes.append(currentNode.up)  
 if currentNode.down is not None and currentNode.down not in availableNodes and currentNode.down not in visitedNodes and currentNode.down.value != '#':  
 availableNodes.append(currentNode.down)  
 return self.path, self.fullPath  
  
 def BFS(self):  
 # Fill the correct path in self.path  
 # self.fullPath should contain the order of visited nodes  
 def isValidPath(current, visited):  
 isValid = False  
 if current.up == visited or current.down == visited or current.right == visited or current.left == visited:  
 isValid = True  
 return isValid  
  
 self.path.clear()  
 self.fullPath.clear()  
  
 visitedNodes = []  
 availableNodes = queue.Queue()  
 availableNodes.put(self.startNode)  
 while availableNodes:  
 currentNode = availableNodes.get()  
 visitedNodes.append(currentNode)  
 self.fullPath.append(currentNode.id)  
 if currentNode.id == self.endNode.id:  
 current = currentNode  
 self.path.append(current.id)  
 visitedNodes.reverse()  
 for visited in visitedNodes:  
 if isValidPath(current, visited):  
 self.path.append(visited.id)  
 current = visited  
 self.path.reverse()  
 return self.path, self.fullPath  
  
 if currentNode.up is not None and currentNode.up not in visitedNodes and currentNode.up.value != '#':  
 availableNodes.put(currentNode.up)  
 if currentNode.down is not None and currentNode.down not in visitedNodes and currentNode.down.value != '#':  
 availableNodes.put(currentNode.down)  
 if currentNode.left is not None and currentNode.left not in visitedNodes and currentNode.left.value != '#':  
 availableNodes.put(currentNode.left)  
 if currentNode.right is not None and currentNode.right not in visitedNodes and currentNode.right.value != '#':  
 availableNodes.put(currentNode.right)  
 return self.path, self.fullPath  
  
 def AStarEuclideanHeuristic(self):  
 # Cost for a step is calculated based on edge cost of node  
 # and use Euclidean Heuristic for evaluating the heuristic value  
 # Fill the correct path in self.path  
 # self.fullPath should contain the order of visited nodes  
 return self.path, self.fullPath, self.totalCost  
  
  
  
def main():  
 searchAlgo = SearchAlgorithms('S,.,.,#,.,.,. .,#,.,.,.,#,. .,#,.,.,.,.,. .,.,#,#,.,.,. #,.,#,E,.,#,.')  
 searchAlgo.Create2DMaze(searchAlgo.mazeStr)  
 searchAlgo.DefineNodesPositions()  
 searchAlgo.PrintMaze(searchAlgo.maze2D)  
 path, fullPath = searchAlgo.DFS()  
 print('\*\*DFS\*\*\nPath is: ' + str(path) + '\nFull Path is: ' + str(fullPath) + '\n\n')  
  
 #######################################################################################  
  
 searchAlgo = SearchAlgorithms('S,.,.,#,.,.,. .,#,.,.,.,#,. .,#,.,.,.,.,. .,.,#,#,.,.,. #,.,#,E,.,#,.')  
 searchAlgo.DefineNodesPositions()  
 path, fullPath = searchAlgo.BFS()  
 print('\*\*BFS\*\*\nPath is: ' + str(path) + '\nFull Path is: ' + str(fullPath) + '\n\n')  
 #######################################################################################  
  
 searchAlgo = SearchAlgorithms('S,.,.,#,.,.,. .,#,.,.,.,#,. .,#,.,.,.,.,. .,.,#,#,.,.,. #,.,#,E,.,#,.', [0, 15, 2, 100, 60, 35, 30, 3  
 , 100, 2, 15, 60, 100, 30, 2  
 , 100, 2, 2, 2, 40, 30, 2, 2  
 , 100, 100, 3, 15, 30, 100, 2  
 , 100, 0, 2, 100, 30])  
 path, fullPath, TotalCost = searchAlgo.AStarEuclideanHeuristic()  
 print('\*\*ASTAR with Euclidean Heuristic \*\*\nPath is: ' + str(path) + '\nFull Path is: ' + str(  
 fullPath) + '\nTotal Cost: ' + str(TotalCost) + '\n\n')  
  
 #######################################################################################  
  
main()