\_\_author\_\_ = "Zach Kaufman"  
  
from graphviz import GraphViz  
import math  
import graphmaker  
  
  
# Searcher class to use the different search algorithms  
def inList(element, listTwo):  
 for items in listTwo:  
 if element.nodeName == items.nodeName:  
 return bool(True)  
 return bool(False)  
  
#sorts a list of searchNodes  
def sortList(unsortedList):  
 n = len(unsortedList)  
 for i in range(n):  
 for j in range(n - i - 1):  
 if unsortedList[j].nodeName > unsortedList[j + 1].nodeName:  
 unsortedList[j], unsortedList[j + 1] = unsortedList[j + 1], unsortedList[j]  
  
  
class Searcher:  
 # Goal node that we are currently searching for  
 goal = []  
  
 # Start Node that we will start are search on  
 start = None  
  
 # Ongoing list of the Nodes we open  
 openList = []  
  
 # Name of File that we load the map from  
 fileName = ""  
  
 # list of the nodes that we loaded from the file  
 mapList = None  
  
 # class search method variable  
 searchMethod = ""  
  
 # what kind of h-function I will use  
 hMode = ""  
  
 # whether we are using verbose or not  
 verboseType = bool(False)  
  
 # the vizgraph that we are using to display the routes for the fuctions  
 vizGraph = None  
  
 # the number of nodes inside of the map  
 numberOfNodes = 0  
  
 # all the nodes in the map  
 nodesinMap = []  
  
 # number of nodes that we search through out an algorithm  
 numberofSearchNodes = 0  
  
 # the final path that we see to get to the end goal[s]  
 path = []  
  
 # all the nodes that we explore through out a search algorithm  
 nodeExpansion = []  
  
 # easy to print goals list  
 printGoals = []  
  
 # currentNode that we are exploring  
 currentNode = None  
  
 # frontier max size  
 frontierMaxSize = 0  
  
 # the max depth that we go into  
 maxDepth = 0  
  
 # the node that a search algorithm ends on  
 endNode = None  
  
 # the total cost of the path we found with the search algorithm  
 pathCost = 0  
  
 # Creates a new searcher class  
 # Takes in file and reads the lines from it and assigns it to MapList  
 def \_\_init\_\_(self, searchtype, filen, htype, verbosemode):  
 self.fileName = filen  
 self.searchMethod = searchtype  
 self.hMode = htype  
 self.verboseType = verbosemode  
 with open(self.fileName) as f:  
 map = f.readlines()  
 # Strips the elements of maplist for any blank space  
 self.mapList = [x.strip() for x in map]  
 print("File loaded!")  
  
  
 # Simple Search Node class  
 # Class imbeded inside the Searcher Class  
 class SearchNode:  
 # Creates a new search node with the Nodename,  
 # Path cost and the children of that given node  
 def \_\_init\_\_(self, nodeName, pathCost, depth, parent):  
 self.nodeName = nodeName  
 self.value = pathCost  
 self.nodeChildren = []  
 self.depth = depth  
 self.parentNode = parent  
  
 # runs the search algorithms  
 # checks to see which algorithm we should be running  
 def go(self):  
 if self.searchMethod.upper() == "BREADTH":  
 self.breadthSearch()  
 elif self.searchMethod.upper() == "DEPTH":  
 self.depthSearch()  
 elif self.searchMethod.upper() == "BEST":  
 self.bestSearch()  
 elif self.searchMethod.upper() == "A\*":  
 self.aStarSearch()  
 else:  
 print("Not a valid Search Method")  
  
 # Counts the number of nodes in a given map  
 def countNodes(self):  
 for line in self.mapList:  
 # replaces everything with spaces and then creates a list split by commas  
 line = line.replace('\'', '').replace('[', '').replace(']', '').replace(' ', '').strip('()')  
 node = line.split(',')  
 # adds to the nodes map if hasn't been added already  
 if node[0] not in self.nodesinMap:  
 self.nodesinMap.append(node[0])  
 if node[1] not in self.nodesinMap:  
 self.nodesinMap.append(node[1])  
 count = 0  
 # counts the number of nodes in the map  
 for i in self.nodesinMap:  
 count = count + 1  
 self.numberOfNodes = count  
  
 # breadth search algorithm  
 def breadthSearch(self):  
 # first prints out what search we are preforming  
 print("BREADTH search: from ", self.start.nodeName, "to", self.goal[0].nodeName)  
 # sets the current node to the start node and sets the max depth and frontier size  
 self.currentNode = self.start  
 self.maxDepth = self.currentNode.depth  
 self.frontierMaxSize = len(self.openList)  
 # loops through while goal list still has nodes inside of it  
 while self.goal:  
 # checks to see if current depth is greater than max depth  
 if self.currentNode.depth > self.maxDepth:  
 self.maxDepth = self.currentNode.depth  
 self.numberofSearchNodes = self.numberofSearchNodes + 1  
 if self.verboseType:  
 print("\nExploring Node: ", self.currentNode.nodeName)  
 # checks to see if the current node is in side the expansion list  
 if self.currentNode.nodeName not in self.nodeExpansion:  
 self.nodeExpansion.append(self.currentNode)  
 # gets the successors of the current node  
 self.getSuccessor(self.currentNode)  
 # adds the successor to the open list  
 self.insertToOpen("END", self.currentNode.nodeChildren, self.currentNode)  
 # sets the new currentnode to the node of the beginning of the open list  
 self.currentNode = self.openList[0]  
 # checks to see if the currentNode is in the goal list  
 if inList(self.currentNode, self.goal):  
 if self.currentNode.depth > self.maxDepth:  
 self.maxDepth = self.currentNode.depth  
 pathNode = self.currentNode  
 # adds the goal node to the pathNode  
 self.path.append(self.currentNode.nodeName)  
 while pathNode.nodeName != self.start.nodeName:  
 self.path.insert(0, pathNode.parentNode.nodeName)  
 pathNode = pathNode.parentNode  
 self.endNode = self.currentNode  
 self.pathCost = self.endNode.value  
 self.numberofSearchNodes = self.numberofSearchNodes + 1  
 self.nodeExpansion.append(self.endNode)  
 # prints out the node stats for current goal node that we found  
 self.printSearchStats()  
 self.nodeExpansion.remove(self.endNode)  
 removeNode = None  
 for nodes in self.goal:  
 if nodes.nodeName == self.currentNode.nodeName:  
 removeNode = nodes  
 self.goal.remove(removeNode)  
  
 # depth search algorithm  
 def depthSearch(self):  
 # first prints out what search we are preforming  
 print("DEPTH search: from ", self.start.nodeName, "to", self.goal[0].nodeName)  
 # sets the current node to the start node and sets the max depth and frontier size  
 self.currentNode = self.start  
 self.maxDepth = self.currentNode.depth  
 self.frontierMaxSize = len(self.openList)  
 # loops through while goal list still has nodes inside of it  
 while self.goal:  
 # checks to see if current depth is greater than max depth  
 if self.currentNode.depth > self.maxDepth:  
 self.maxDepth = self.currentNode.depth  
 self.numberofSearchNodes = self.numberofSearchNodes + 1  
 if self.verboseType:  
 print("\nExploring Node: ", self.currentNode.nodeName)  
 # checks to see if the current node is in side the expansion list  
 if self.currentNode.nodeName not in self.nodeExpansion:  
 self.nodeExpansion.append(self.currentNode)  
 # gets the successors of the current node  
 self.getSuccessor(self.currentNode)  
 # adds the successor to the open list  
 self.insertToOpen("FRONT", self.currentNode.nodeChildren, self.currentNode)  
 # sets the new currentnode to the node of the beginning of the open list  
 self.currentNode = self.openList[0]  
 # checks to see if the currentNode is in the goal list  
 if inList(self.currentNode, self.goal):  
 if self.currentNode.depth > self.maxDepth:  
 self.maxDepth = self.currentNode.depth  
 pathNode = self.currentNode  
 # adds the goal node to the pathNode  
 self.path.append(self.currentNode.nodeName)  
 while pathNode.nodeName != self.start.nodeName:  
 self.path.insert(0, pathNode.parentNode.nodeName)  
 pathNode = pathNode.parentNode  
 self.endNode = self.currentNode  
 self.pathCost = self.endNode.value  
 self.numberofSearchNodes = self.numberofSearchNodes + 1  
 self.nodeExpansion.append(self.endNode)  
 # prints out the node stats for current goal node that we found  
 self.printSearchStats()  
 self.nodeExpansion.remove(self.endNode)  
 removeNode = None  
 for nodes in self.goal:  
 if nodes.nodeName == self.currentNode.nodeName:  
 removeNode = nodes  
 self.goal.remove(removeNode)  
  
 # Best Search Algorithm  
 def bestSearch(self):  
 # first prints out what search we are preforming  
 print("BEST search: from ", self.start.nodeName, "to", self.goal[0].nodeName)  
 # sets the current node to the start node and sets the max depth and frontier size  
 self.currentNode = self.start  
 self.maxDepth = self.currentNode.depth  
 self.frontierMaxSize = 1  
 # loops through while goal list still has nodes inside of it  
 while self.goal:  
 # checks to see if current depth is greater than max depth  
 if self.currentNode.depth > self.maxDepth:  
 self.maxDepth = self.currentNode.depth  
 self.numberofSearchNodes = self.numberofSearchNodes + 1  
 if self.verboseType:  
 print("\nExploring Node: ", self.currentNode.nodeName)  
 # checks to see if the current node is in side the expansion list  
 if self.currentNode.nodeName not in self.nodeExpansion:  
 self.nodeExpansion.append(self.currentNode)  
 # gets the successors of the current node  
 self.getSuccessor(self.currentNode)  
 # adds the successor to the open list  
 self.insertToOpen("INORDER", self.currentNode.nodeChildren, self.currentNode)  
 # sets the new currentnode to the node of the beginning of the open list  
 self.currentNode = self.openList[0]  
 # checks to see if the currentNode is in the goal list  
 if inList(self.currentNode, self.goal):  
 if self.currentNode.depth > self.maxDepth:  
 self.maxDepth = self.currentNode.depth  
 pathNode = self.currentNode  
 # adds the goal node to the pathNode  
 self.path.append(self.currentNode.nodeName)  
 while pathNode.nodeName != self.start.nodeName:  
 self.path.insert(0, pathNode.parentNode.nodeName)  
 pathNode = pathNode.parentNode  
 self.endNode = self.currentNode  
 self.pathCost = self.endNode.value  
 self.numberofSearchNodes = self.numberofSearchNodes + 1  
 self.nodeExpansion.append(self.endNode)  
 # prints out the node stats for current goal node that we found  
 self.printSearchStats()  
 self.nodeExpansion.remove(self.endNode)  
 removeNode = None  
 for nodes in self.goal:  
 if nodes.nodeName == self.currentNode.nodeName:  
 removeNode = nodes  
 self.goal.remove(removeNode)  
  
 # A\* search Algorithm  
 def aStarSearch(self):  
 # first prints out what search we are preforming  
 print("A\* search: from ", self.start.nodeName, "to", self.goal[0].nodeName)  
 # sets the current node to the start node and sets the max depth and frontier size  
 self.currentNode = self.start  
 self.maxDepth = self.currentNode.depth  
 self.frontierMaxSize = 1  
 # loops through while goal list still has nodes inside of it  
 while self.goal:  
 # checks to see if current depth is greater than max depth  
 if self.currentNode.depth > self.maxDepth:  
 self.maxDepth = self.currentNode.depth  
 self.numberofSearchNodes = self.numberofSearchNodes + 1  
 if self.verboseType:  
 print("\nExploring Node: ", self.currentNode.nodeName)  
 # checks to see if the current node is in side the expansion list  
 if self.currentNode.nodeName not in self.nodeExpansion:  
 self.nodeExpansion.append(self.currentNode)  
 # gets the successors of the current node  
 self.getSuccessor(self.currentNode)  
 # adds the successor to the open list  
 self.insertToOpen("H-FUNCTION", self.currentNode.nodeChildren, self.currentNode)  
 # sets the new currentnode to the node of the beginning of the open list  
 self.currentNode = self.openList[0]  
 # checks to see if the currentNode is in the goal list  
 if inList(self.currentNode, self.goal):  
 if self.currentNode.depth > self.maxDepth:  
 self.maxDepth = self.currentNode.depth  
 pathNode = self.currentNode  
 # adds the goal node to the pathNode  
 self.path.append(self.currentNode.nodeName)  
 while pathNode.nodeName != self.start.nodeName:  
 self.path.insert(0, pathNode.parentNode.nodeName)  
 pathNode = pathNode.parentNode  
 self.endNode = self.currentNode  
 self.pathCost = self.endNode.value  
 self.numberofSearchNodes = self.numberofSearchNodes + 1  
 self.nodeExpansion.append(self.endNode)  
 # prints out the node stats for current goal node that we found  
 self.printSearchStats()  
 self.nodeExpansion.remove(self.endNode)  
 removeNode = None  
 for nodes in self.goal:  
 if nodes.nodeName == self.currentNode.nodeName:  
 removeNode = nodes  
 self.goal.remove(removeNode)  
  
  
 # Print the search states for a given algorithm and map  
 def printSearchStats(self):  
 self.countNodes()  
 print("\n\n-------------------------------------------------")  
 print("SEARCH SUMMARY STATS:")  
 print("Search Type: ", self.searchMethod, " Map File: ",  
 self.fileName, "Total Nodes in Graph: ", self.numberOfNodes)  
 print("Start Node: ", self.start.nodeName,  
 "; Goal Node(s): ", end="[")  
 for node in self.printGoals:  
 print(node.nodeName, end=',')  
 print("]")  
 print("Search total of ", self.numberofSearchNodes,  
 " out of total of ", self.numberOfNodes, " in graph")  
 print("Ended at Node: ", self.endNode.nodeName, " with path cost: ", self.pathCost)  
 print("Path (", len(self.path), "): ", self.path)  
 print("Frontier size: Average= ", len(self.openList) / self.frontierMaxSize  
 , "Max size =", self.frontierMaxSize)  
 print("Depth of Search: Average=", len(self.openList) / self.maxDepth,  
 ";Max Depth = ", self.maxDepth)  
 print("Order of Node Expansion:", end=" [")  
 for nodes in self.nodeExpansion:  
 print(nodes.nodeName, end =",")  
 print("]")  
  
 # Sets the start of the search function and the end of the search function  
 # Creates a the Graph Vizual and adds the starting node to the open list  
 def setStartGoal(self, startNode, goalNode):  
 # Sets the start Node  
 self.start = self.SearchNode(startNode, 0, 0, None)  
 # my insertToOpen function takes a list so I made a dummylist just to keep it consistent  
 # and add my starting node to my open list  
 startList = [self.start]  
 goals = goalNode.split(',')  
 for goal in goals:  
 self.goal.append(self.SearchNode(goal, None, None, None))  
 self.printGoals = self.goal  
 self.insertToOpen("FRONT", startList, '')  
 # Calls the function to create the graph vizual with startnode and goal node  
 self.createGraphViz(startNode, goalNode)  
 # sets the goal node that we are searching for  
 print("Start = ", startNode, "Goal = ", goalNode)  
  
 # Creates the starting graph vizual and sets the  
 # start and goal nodes on the graph  
 def createGraphViz(self, startLabel, goalLabel):  
 self.vizGraph = GraphViz()  
 self.vizGraph.loadGraphFromFile(self.fileName)  
 self.vizGraph.plot()  
 self.vizGraph.markStart(startLabel)  
 goals = goalLabel.split(',')  
 for goal in goals:  
 self.vizGraph.markGoal(goal)  
  
 # Takes in a list of node and adds them to open list  
 # Checks to see if a duplicate node is already in the list  
 # Checks to see if list is empty  
 # adds list, at the front, and the end, and "in order"  
 # depending on value of the node given  
 def insertToOpen(self, addStyle, addToList, parentNode):  
 # gets rid of the node that we are currently exploring  
 if self.verboseType and parentNode != '':  
 print("Inserting new Children:", end=" ")  
 index = 0  
 for children in parentNode.nodeChildren:  
 if not inList(children, self.nodeExpansion):  
 print(children.nodeName, end=", ")  
 index = index + 1  
 if self.frontierMaxSize < index:  
 self.frontierMaxSize = index  
 if inList(parentNode, self.openList):  
 self.openList.remove(parentNode)  
  
 index = 0  
 if parentNode != '':  
 for children in parentNode.nodeChildren:  
 if not inList(children, self.nodeExpansion):  
 index = index + 1  
 if self.frontierMaxSize < index:  
 self.frontierMaxSize = index  
  
 # Checks to see if we are inserting children at the front of the open list  
 if addStyle == "FRONT":  
 # iterates through elements of the addToList  
 counter = len(addToList) - 1  
 while counter >= 0:  
 # Checks to make sure that there isn't already a given node in the list  
 if inList(addToList[counter], self.openList) and not inList(addToList[counter], self.nodeExpansion):  
 removeNode = None  
 for node in self.openList:  
 if node.nodeName == addToList[counter].nodeName:  
 removeNode = node  
 self.openList.remove(removeNode)  
 if not inList(addToList[counter], self.nodeExpansion):  
 self.openList.insert(0, addToList[counter])  
 counter = counter - 1  
 if self.verboseType:  
 self.showOpenList()  
 # checks to see if we are inserting children at the end of the open list  
 elif addStyle == "END":  
 # iterates through elements of the addToList  
 for child in addToList:  
 # Checks to make sure that there isn't already a given node in the list  
 if not inList(child, self.openList) and not inList(child, self.nodeExpansion):  
 self.openList.append(child)  
  
  
 elif addStyle == "H-FUNCTION":  
 for child in addToList:  
 # Checks to make sure that there isn't already a given node in the list  
 if inList(child, self.openList) and not inList(child, self.nodeExpansion):  
 removeNode = None  
 for node in self.openList:  
 if node.nodeName == child.nodeName:  
 removeNode = node  
 removeNodeHSLD = self.gethSLD(removeNode.nodeName)  
 childHSLD = self.gethSLD(child.nodeName)  
 index = 0  
 while index < len(removeNodeHSLD):  
 if (removeNodeHSLD[index] + int(removeNode.value)  
 > childHSLD[index] + int(child.value)):  
 self.openList.remove(removeNode)  
 break  
 index = index + 1  
  
 if not inList(child, self.openList) and not inList(child, self.nodeExpansion):  
 counter = 0  
 # Iterates through open list to check the different values of the list  
 for nodes in self.openList:  
 # checks to see if the value of the given node is less than any of the nodes in  
 # the open list and then places it in it's correct spot  
 childHSLD = self.gethSLD(child.nodeName)  
 nodeHSLD = self.gethSLD(nodes.nodeName)  
 smallestValue = self.findSmallest(childHSLD, nodeHSLD)  
 if (smallestValue in childHSLD and not inList(child, self.openList)  
 and not inList(child, self.nodeExpansion)):  
 self.openList.insert(counter, child)  
 break  
 elif (smallestValue in nodeHSLD and not inList(nodes, self.openList)  
 and not inList(nodes, self.nodeExpansion)):  
 self.openList.insert(counter, nodes)  
 break  
 counter = counter + 1  
 # Checks to see if the node was placed in open list,  
 # if not placed at the end of the open list  
 if not inList(child, self.openList) and not inList(child, self.nodeExpansion):  
 self.openList.append(child)  
  
 # if not inserting at end or front then inserting children in order of open list  
 else:  
 # iterates through elements of the addToList  
 for child in addToList:  
 # Checks to make sure that there isn't already a given node in the list  
 if inList(child, self.openList) and not inList(child, self.nodeExpansion):  
 removeNode = None  
 for node in self.openList:  
 if node.nodeName == child.nodeName:  
 removeNode = node  
 if (removeNode.value > child.value):  
 self.openList.remove(removeNode)  
 if not inList(child, self.openList) and not inList(child, self.nodeExpansion):  
 counter = 0  
 # Iterates through open list to check the different values of the list  
 for nodes in self.openList:  
 # checks to see if the value of the given node is less than any of the nodes in  
 # the open list and then places it in it's correct spot  
 if child.value < nodes.value:  
 self.openList.insert(counter, child)  
 break  
 counter = counter + 1  
 # Checks to see if the node was placed in open list,  
 # if not placed at the end of the open list  
 if not inList(child, self.openList) and not inList(child, self.nodeExpansion):  
 self.openList.append(child)  
 if self.verboseType:  
 self.showOpenList()  
  
 # finds the smallest value in the 2 given list  
 def findSmallest(self, listOne, listTwo):  
 smallest = 100000  
 for listOneNode in listOne:  
 for listTwoNode in listTwo:  
 if listOneNode < listTwoNode:  
 if listOneNode < smallest:  
 smallest = listOneNode  
 elif listTwoNode < listOneNode:  
 if listTwoNode < smallest:  
 smallest = listTwoNode  
 return smallest  
  
 # takes in a SearchNode and a list of tupples and checks to see  
 # if the SearchNode name is in the list  
 # gives all the children, Successors that are connect, to the parent node that  
 # is given and returns a printed list in Alphabetical order of the children nodes  
 # also assigns those children nodes to the parent node  
 def getSuccessor(self, parentNode):  
 addNode = None  
 # Loops through the map list and makes an easy list to read  
 for line in self.mapList:  
 line = line.replace('\'', '').replace('[', '').replace(']', '').replace(' ', '').strip('()')  
 node = line.split(',') # clean comma-separated values. that is a list now  
 # checks to see which elements in the new list are connect to the parent node  
 if parentNode.nodeName in node:  
 # checks to see if parent node is the first or second element  
 # then assigns whichever it is not to be the children  
 if node[0] == parentNode.nodeName:  
 addNode = self.SearchNode(node[1], int(node[2]) + parentNode.value,  
 parentNode.depth + 1, parentNode)  
 elif node[1] == parentNode.nodeName:  
 addNode = self.SearchNode(node[0], int(node[2]) + parentNode.value,  
 parentNode.depth + 1, parentNode)  
 parentNode.nodeChildren.append(addNode) # adds new children node to parent  
 # sorts the list parentNode's children list in alphabetical order  
 sortList(parentNode.nodeChildren)  
 # prints out the list of children  
  
 # Bubble sorts a list of SearchNodes  
  
 # Calculates the hSLD and gives it back to user  
 # can calculate for multiple goals  
 # returns a list of hSLD for different goals  
 def gethSLD(self, giveNodeName):  
 givenNodeCoordinates = None  
 goalNodeCoordinates = []  
 listOfhSLD = []  
 for line in self.mapList:  
 line = line.replace('\'', '').replace('[', '').replace(']', '').replace(' ', '').strip('()')  
 node = line.split(',')  
 for goals in self.goal:  
 if goals.nodeName in node:  
 if node[0] == goals.nodeName:  
 goalNodeCoordinates.append((node[3], node[4]))  
 elif node[1] == goals.nodeName:  
 goalNodeCoordinates.append((node[5], node[6]))  
 if giveNodeName in node:  
 if node[0] == giveNodeName:  
 givenNodeCoordinates = (node[3], node[4])  
 elif node[1] == giveNodeName:  
 givenNodeCoordinates = (node[5], node[6])  
 for goalCoordiantes in goalNodeCoordinates:  
 hSLD = math.sqrt((int(goalCoordiantes[0]) - int(givenNodeCoordinates[0])) \*\* 2  
 + (int(goalCoordiantes[1]) - int(givenNodeCoordinates[1])) \*\* 2)  
 hSLD = round(hSLD, 2)  
 if hSLD not in listOfhSLD:  
 listOfhSLD.append(hSLD)  
 return listOfhSLD  
  
 # returns the current open list  
 def showOpenList(self):  
 print("")  
 for nodes in self.openList:  
 huerticsTB = []  
 for heuristics in self.gethSLD(nodes.nodeName):  
 huerticsTB.append(round(heuristics + int(nodes.value), 2))  
 print("[", nodes.nodeName, ", ", nodes.depth, ", ", nodes.value,  
 ", ", self.gethSLD(nodes.nodeName), ", ", huerticsTB, end = "],")  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 z = Searcher('A\*', 'random230.txt', 'hSLD', bool(False))  
 z.setStartGoal("R", "G,W,Q")  
 z.getSuccessor(z.start)  
 z.go()