Adrian Monreal Lab 7

Dr. Fuentes asked us to complete 5 functions, this lab edited the maze lab that we just did so for the first function we ask for use input how many walls they would like to remove if its greater than the number of cells it displays that there is more than one path to end, if the user input is equal to the number of cells minus 1 than there is one path but if theres less than that then there’s no guaranteed path to the end.

1. Modify your maze-building program to allow for both cases mentioned above.   
   \*\*\*Your program should display n, the number of cells,   
   \*\*\*and ask the user for m, the number of walls to remove,   
   \*\*\*then display a message indicating one of the following:  
   (a) A path from source to destination is not guaranteed to exist (when m < n − 1)  
    (b) The is a unique path from source to destination (when m = n − 1)  
   (c) There is at least one path from source to destination (when m > n − 1)
2. Write a method to build the adjacency list representation of your maze.  
    Cells in the maze should be represented by vertices in the graph.   
    If two cells u and v are contiguous and there is no wall separating them,   
    then there must be an edge from u to v in the graph.   
    The example below shows a maze and the corresponding graph representation.
3. 3. Implement the following algorithms to solve the maze you created,   
   assuming the starting position is bottom-left corner and the goal position is the top-right corner.  
   (a) Breadth-first search.  
   (b) Depth-first search using a stack. This is identical to breadth-first search but the queue is replaced by a stack.  
   (c) Depth-first search using recursion.

*"""  
Adrian Monreal  
lab 7  
CS2302 Olac Fuentes  
  
"""*import dsf  
import matplotlib.pyplot as plt  
import numpy as np  
import random  
  
import time  
  
  
  
"""  
# Starting point for program to build and draw a maze  
# Modify program using disjoint set forest to ensure there is exactly one  
# simple path joining any two cells  
# Programmed by Olac Fuentes  
"""  
  
class Queue:  
 def \_\_init\_\_(self):  
 self.items = []  
  
  
 def isEmpty(self):  
 return self.items == []  
  
  
 def enqueue(self, item):  
 self.items.insert(0, item)  
  
  
 def dequeue(self):  
 return self.items.pop()  
  
  
 def size(self):  
 return len(self.items)  
  
  
class Stack:  
 def \_\_init\_\_(self):  
 self.items = []  
  
 def isEmpty(self):  
 return self.items == []  
  
 def push(self, item):  
 self.items.append(item)  
  
 def pop(self):  
 return self.items.pop()  
  
 def peek(self):  
 return self.items[len(self.items) - 1]  
  
 def size(self):  
 return len(self.items)  
  
def draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=False):  
 fig, ax = plt.subplots()  
 for w in walls:  
 if w[1]-w[0] ==1: #vertical wall  
 x0 = (w[1]%maze\_cols)  
 x1 = x0  
 y0 = (w[1]//maze\_cols)  
 y1 = y0+1  
 else:#horizontal wall  
 x0 = (w[0]%maze\_cols)  
 x1 = x0+1  
 y0 = (w[1]//maze\_cols)  
 y1 = y0  
 ax.plot([x0,x1],[y0,y1],linewidth=1,color='k')  
 sx = maze\_cols  
 sy = maze\_rows  
 ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k')  
 if cell\_nums:  
 for r in range(maze\_rows):  
 for c in range(maze\_cols):  
 cell = c + r\*maze\_cols  
 ax.text((c+.5),(r+.5), str(cell), size=10,  
 ha="center", va="center")  
 ax.axis('off')  
 ax.set\_aspect(1.0)  
 plt.show()  
 fig.savefig('maze.png')  
  
 #M #n  
def wall\_list(maze\_rows , maze\_cols):  
 # Creates a list with all the walls in the maze  
 w =[]  
 for r in range(maze\_rows):  
 for c in range(maze\_cols):  
 cell = c + r\*maze\_cols  
 if c!=maze\_cols-1:  
 w.append([cell,cell+1])  
 if r!=maze\_rows-1:  
 w.append([cell,cell+maze\_cols])  
 return w  
  
plt.close("all")  
  
def dsfMaze(rows,columns,wallList):  
 cells = rows \* columns  
 s = dsf.DisjointSetForest(cells)  
 while dsf.NumSets(s)> 1:  
 adjIndex = 0  
 curr = random.randint(0,len(wallList)-1)  
 wall = wallList[curr]  
 if dsf.find(s,wall[0]) != dsf.find(s,wall[1]):  
 dsf.union(s, wall[0], wall[1])  
 wallList.pop(curr)  
 return wallList  
"""  
1. Modify your maze-building program to allow for both cases mentioned above.   
\*\*\*Your program should display n, the number of cells,   
\*\*\*and ask the user for m, the number of walls to remove,   
\*\*\*then display a message indicating one of the following:  
(a) A path from source to destination is not guaranteed to exist (when m < n − 1)  
 (b) The is a unique path from source to destination (when m = n − 1)  
(c) There is at least one path from source to destination (when m > n − 1)  
"""  
  
def Maze2(rows,columns,wallList):  
 n = rows \* columns #number of cells  
 print("there are ", n, "cells ")  
 m = input("How many walls would you like to remove")  
 m = int(m)  
 s = dsf.DisjointSetForest(n)  
 #a  
 if m<n-1:  
 print("A path from source to destination is not guaranteed to exist (when m < n − 1)")  
 #b  
 if m == n-1:  
 print("The is a unique path from source to destination (when m = n − 1)")  
 #c  
 if m>n-1:  
 print("There is at least one path from source to destination (when m > n − 1)")  
 if m > len(wallList) :  
 while m > 0:  
  
 x = random.randint(0, len(wallList) - 1)  
 wallList.pop(x)  
 m -= 1  
 return wallList  
 while m> 0:  
 curr = random.randint(0,len(wallList)-1)  
 wall = wallList[curr]  
 if dsf.find(s,wall[0]) != dsf.find(s,wall[1]):  
 dsf.union(s, wall[0], wall[1])  
 wallList.pop(curr)  
 m-=1  
 return wallList  
  
"--------------------------------------------------------------------------------------"  
  
  
"""  
2. Write a method to build the adjacency list representation of your maze.  
 Cells in the maze should be represented by vertices in the graph.   
 If two cells u and v are contiguous and there is no wall separating them,   
 then there must be an edge from u to v in the graph.   
 The example below shows a maze and the corresponding graph representation.  
 """  
def adjListdsfMaze(rows,columns,wallList):  
 cells = rows \* columns  
 s = dsf.DisjointSetForest(cells)  
 G = [[] for i in range(len(cells))]  
 while dsf.NumSets(s)> 1:  
 curr = random.randint(0,len(wallList)-1)  
 wall = wallList[curr]  
 if dsf.find(s,wall[0]) != dsf.find(s,wall[1]):  
 dsf.union(s, wall[0], wall[1])  
 newEntry = wallList.pop(curr)  
 G[newEntry[0]].append(newEntry[1])  
 return G  
"------------------------------------------------------------------------"  
  
"""  
3. Implement the following algorithms to solve the maze you created,   
assuming the starting position is bottom-left corner and the goal position is the top-right corner.  
(a) Breadth-first search.  
(b) Depth-first search using a stack. This is identical to breadth-first search but the queue is replaced by a stack.  
(c) Depth-first search using recursion.  
"""  
#a  
def breadthFirstsearch(G,v ):  
 visited = np.zeros(len(G),dtype=bool)  
 prev = np.zeros(len(G),dtype=int)-1  
 q = Queue()  
 q.enqueue(v)  
 visited[v] = True  
 while not q.isEmpty():  
 u = q.dequeue()  
 for t in G[u]:  
 if not visited[G]:  
 visited[t] = True  
 prev[t] = u  
 q.enqueue(t)  
 return prev  
#b  
def depthFirstSearchS(G,v):  
 s = Stack()  
 s.push(v)#source  
 visited = []  
  
  
  
  
  
"\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_"  
  
  
maze\_rows = 10  
  
maze\_cols = 15  
  
  
walls = wall\_list(maze\_rows,maze\_cols)  
  
draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True)  
  
new = dsfMaze(maze\_rows,maze\_cols,walls)  
  
""""  
start = time.time()  
draw\_maze(new,maze\_rows,maze\_cols,cell\_nums=False)  
end = time.time()  
print(end - start)  
"""  
print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")  
F1 = Maze2(maze\_rows,maze\_cols,walls)  
draw\_maze(F1,maze\_rows,maze\_cols,cell\_nums=False)  
print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")  
F2 = adjListdsfMaze(maze\_rows,maze\_cols,F1)  
print(F2)  
print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")  
F3a = breadthFirstsearch(F2, 0)  
print(F3a)  
print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")  
""""  
  
F3b = Maze2(maze\_rows,maze\_cols,walls)  
draw\_maze(F3,maze\_rows,maze\_cols,cell\_nums=False)  
  
"""  
print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")  
""""  
F3c = Maze2(maze\_rows,maze\_cols,walls)  
draw\_maze(F3,maze\_rows,maze\_cols,cell\_nums=False)  
  
"""  
print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

“I certify that this project is entirely my own work. I

wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.”