CS 2302

Lab 7 Report

Author: David Ayala

ID: 80617838

Instructor: Olac Fuentes

TA: Anindita Nath

**Introduction**

The purpose of this lab was to 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. 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.

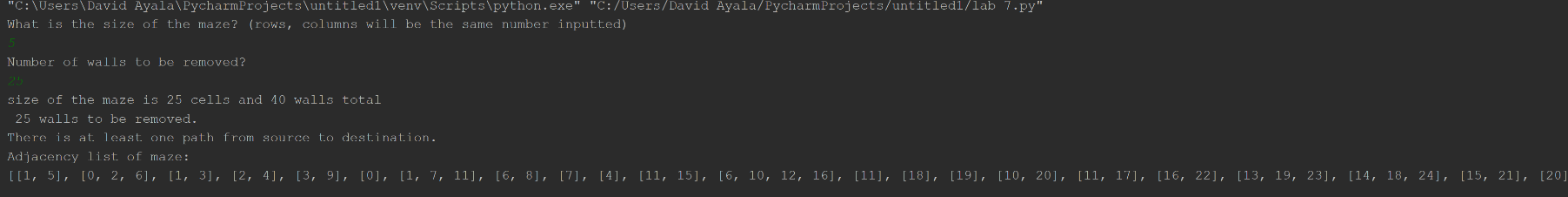
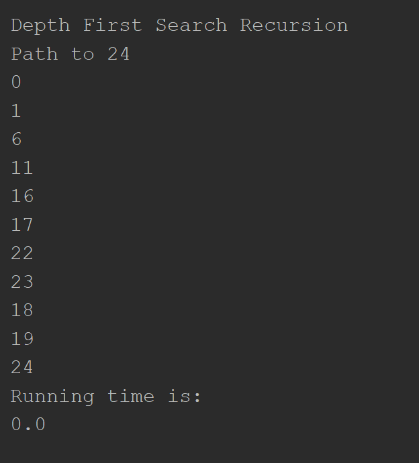
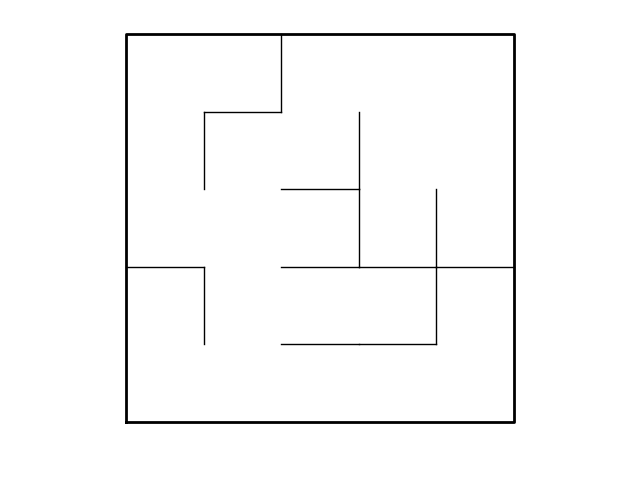
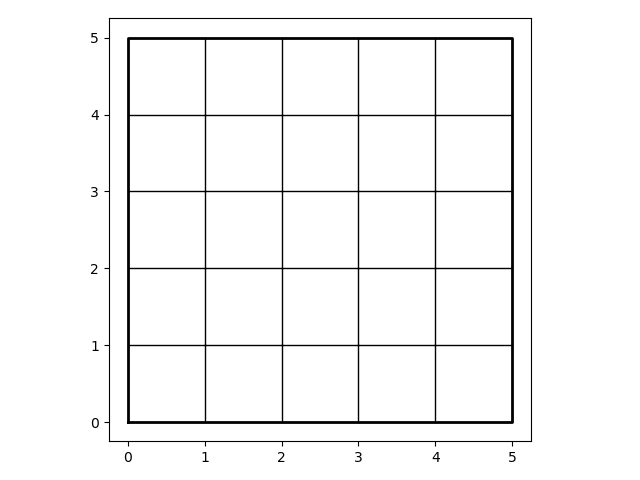
**Proposed Solution & Design Implementation**

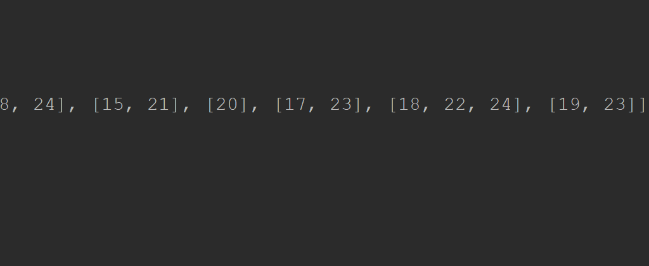
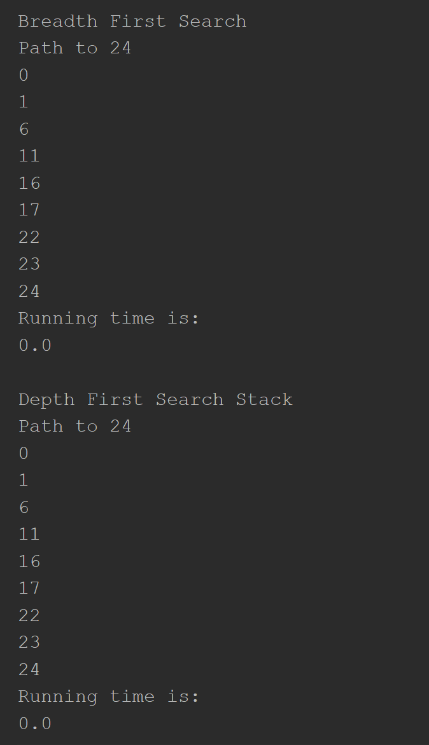
Since all I had to do was modify lab 6 to make lab 7, that meant that I have to the following cases remove less than n − 1 walls, some cells will not be reachable from the start cell. If you remove more than n − 1 walls (notice that after removing n − 1 walls, all remaining walls separate cells that are reachable from each other by adjusting how the code worked. Then when it came to the adjacency list all I had to do was looking through a previous quiz that we had done and find a graph to adjacency list function that we had done before and slightly modify it. Once that was done I had to do Breadth-first search, Depth-first search using a stack, Depth-first search using recursion. I remember that professor Fuentes had gone over pseudo code about them before and I used that a basis for the code. I couldn’t figure out what was causing the Running time not to work.

**Experimental Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **Size** | **Breadth-first search** | **Depth-first search using recursion** | **Depth-first search using recursion** |
| **5 by 5 (25)** | 0.0 | 0.0 | 0.0 |
| **10 by 10 (100)** | 0.0 | 0.0 | 0.0 |
| **15 by 15 (225)** | 0.0 | 0.0 | 0.0 |
| **20 by 20 (400)** | 0.0 | 0.0 | 0.0 |

**Not Sure what went wrong with the Running times but if you see in the code, I tried to do it.**

****



**Conclusion**

This lab allowed us to learn how to go from a graph to adjacency list and how to do Breadth-first search, Depth-first search using a stack, Depth-first search using recursion. This lab also was great way to brush up for the next exam and how to make mazes/removing walls from them to make them solvable.

**Appendix**

# Course:CS 2302 MW 1:30-2:50, Author:David Ayala  
# Assignment:Lab #7, Instructor: Olac Fuentes  
# Teaching Assistant: Maliheh Zargaran, Date of last Modification: 5/1/2019  
# Purpose of program: 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. 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  
import matplotlib.pyplot as plt  
import queue  
import numpy as np  
import random  
import time  
  
  
  
def DisjointSetForest(size):  
 return np.zeros(size, dtype=np.int) - 1  
  
  
def find(S, i):  
 # Returns root of tree that i belongs to  
 if S[i] < 0:  
 return i  
 return find(S, S[i])  
  
def numberOfSets(S):  
 counter = 0  
 for i in S:  
 if i < 0:  
 counter += 1  
 return counter  
  
def union(S, i, j):  
 # Joins i's tree and j's tree, if they are different  
 ri = find(S, i)  
 rj = find(S, j)  
 if ri != rj: # Do nothing if i and j belong to the same set  
 S[rj] = ri # Make j's root point to i's root  
  
  
def find\_c(S, i):  
 if S[i] < 0:  
 return i  
 r = find\_c(S, S[i])  
 S[i] = r  
 return S[i]  
  
  
def union\_by\_size(S, i, j):  
 ri = find\_c(S, i)  
 rj = find\_c(S, j)  
 if ri != rj:  
 if S[ri] > S[rj]:  
 S[rj] += S[ri]  
 S[ri] = rj  
 else:  
 S[ri] += S[rj]  
 S[rj] = ri  
  
def userInput():  
 print("Number of walls to be removed?")  
 temp = input()  
 m = int(temp)  
 return m  
  
def wall\_list(maze\_rows, maze\_cols):  
 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  
  
def makeMaze(size, walls):  
 wallsToBeRemoved = userInput()  
 totalSize = size \* size  
 Sets = DisjointSetForest(totalSize)  
 print('size of the maze is '+ str(totalSize) +' cells and ' + str(len(walls)) + ' walls total')  
 print(' '+ str(wallsToBeRemoved) + ' walls to be removed.')  
 if wallsToBeRemoved > totalSize - 1:  
 sets = numberOfSets(Sets)  
 if wallsToBeRemoved > len(walls):  
 return []  
 while wallsToBeRemoved > 0:  
 temp = random.randint(0, len(walls) - 1)  
 if sets == 1:  
 walls.pop(temp)  
 wallsToBeRemoved -= 1  
 elif find\_c(Sets, walls[temp][0]) != find\_c(Sets, walls[temp][1]):  
 union\_by\_size(Sets, walls[temp][0], walls[temp][1])  
 walls.pop(temp)  
 wallsToBeRemoved -= 1  
 sets -= 1  
 print('There is at least one path from source to destination.')  
 return walls  
  
 elif wallsToBeRemoved < totalSize - 1:  
 print('A path from source to destination is not guaranteed to exist.')  
 elif wallsToBeRemoved == totalSize - 1:  
 print('There is a unique path from source to destination.')  
 else:  
 print('please run again')  
 while wallsToBeRemoved > 0:  
 # Select a random wall w =[c1,c2]  
 w = random.randint(0, len(walls) - 1)  
 # If cells c1 and c2 belong to different sets...  
 if find\_c(Sets, walls[w][0]) != find\_c(Sets, walls[w][1]):  
 # remove w and join c1’s set and c2’s set  
 union\_by\_size(Sets, walls[w][0], walls[w][1])  
 walls.pop(w)  
 wallsToBeRemoved -= 1  
 return walls  
  
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)  
  
def mazeToAdjacencyList(walls, size, walls2):  
 adjacencyList = []  
 for i in range(size \* size):  
 adjacencyList.append([])  
 for wall in walls2:  
 if wall not in walls:  
 adjacencyList[wall[0]].append(wall[1])  
 adjacencyList[wall[1]].append(wall[0])  
 return adjacencyList  
  
  
start = time.time()  
def DepthFirstSearchStack(G, vertices):  
 previous = len(G) \* [-1]  
 list = []  
 visited = len(G) \* [False]  
 visited[vertices] = True  
 list.append(vertices)  
 while len(list) > 0:  
 temp = list.pop()  
 for i in G[temp]:  
 if not visited[i]:  
 visited[i] = True  
 previous[i] = temp  
 list.append(i)  
 return previous  
end = time.time()  
  
start2 = time.time()  
def DepthFirstSearchRecursion(G, vertices):  
 visited[vertices] = True  
 for i in G[vertices]:  
 if visited[i] is False:  
 previous[i] = vertices  
 DepthFirstSearchRecursion(G, i)  
 return previous  
end2 = time.time()  
  
  
start3 = time.time()  
def BreadthFirstSearch(G, vertices):  
 visited = len(G) \* [False]  
 visited[vertices] = True  
 previous = len(G)\*[-1]  
 temp = queue.Queue()  
 temp.put(vertices)  
 while not temp.empty():  
 temp2 = temp.get()  
 for i in G[temp2]:  
 if not visited[i]:  
 visited[i] = True  
 previous[i] = temp2  
 temp.put(i)  
 return previous  
end3 = time.time()  
  
  
def printingPath(previous, vertices):  
 if previous[vertices] != -1:  
 printingPath(previous, previous[vertices])  
 print(vertices)  
  
plt.close("all")  
  
print('What is the size of the maze? (rows, columns will be the same number inputted) ')  
size = input()  
size = int(size)  
  
originalWalls = wall\_list(size, size)  
walls = wall\_list(size, size)  
maze = makeMaze(size, walls)  
draw\_maze(originalWalls, size, size, cell\_nums=True)  
draw\_maze(maze, size, size, cell\_nums=False)  
AdjacencyList = mazeToAdjacencyList(walls, size, originalWalls)  
previous = len(AdjacencyList) \* [-1]  
visited = len(AdjacencyList) \* [False]  
BFS = BreadthFirstSearch(AdjacencyList, 0)  
DFSS = DepthFirstSearchStack(AdjacencyList, 0)  
DFSR = DepthFirstSearchRecursion(AdjacencyList, 0)  
mazeSize = size \* size - 1  
  
print('Adjacency list of maze:')  
print(AdjacencyList)  
  
print()  
  
print('Breadth First Search')  
print('Path to ' + str(mazeSize) + '')  
printingPath(BFS, mazeSize)  
print('Running time is:')  
print(end3 - start3)  
  
print()  
  
print('Depth First Search Stack')  
print('Path to ' + str(mazeSize) + '')  
printingPath(DFSS, mazeSize)  
print('Running time is:')  
print(end - start)  
  
print()  
  
print('Depth First Search Recursion')  
print('Path to ' + str(mazeSize) + '')  
printingPath(DFSR, mazeSize)  
print('Running time is:')  
print(end2 - start2)  
  
plt.show()