**Emmanuel Alvarez**

**80567137**

**Introduction**

The purpose of this lab is to create a maze using a disjoin set forest structure as la lab 6, but this implementing breadth first search, depth first search and depth first search using a recursive method, this with the purpose of create a path from the first cell to the last cell of the maze.

**Proposed solution and design implementation**

First of all, I created an array containing a graph representing it as an adjacency list. To create this method, I first compared the array that has the walls between two cells and the array that has cells where there is not a wall between them, if they had a same value, I removed it from the second array. Then, with the resulting array, I went through the whole array and create the adjacency list by adding to each index the nodes where this index is connected to. When I had the adjacency list, I made the other methods to create an array containing the path to go from the first cell to the final cell of the maze depending of the search algorithm.

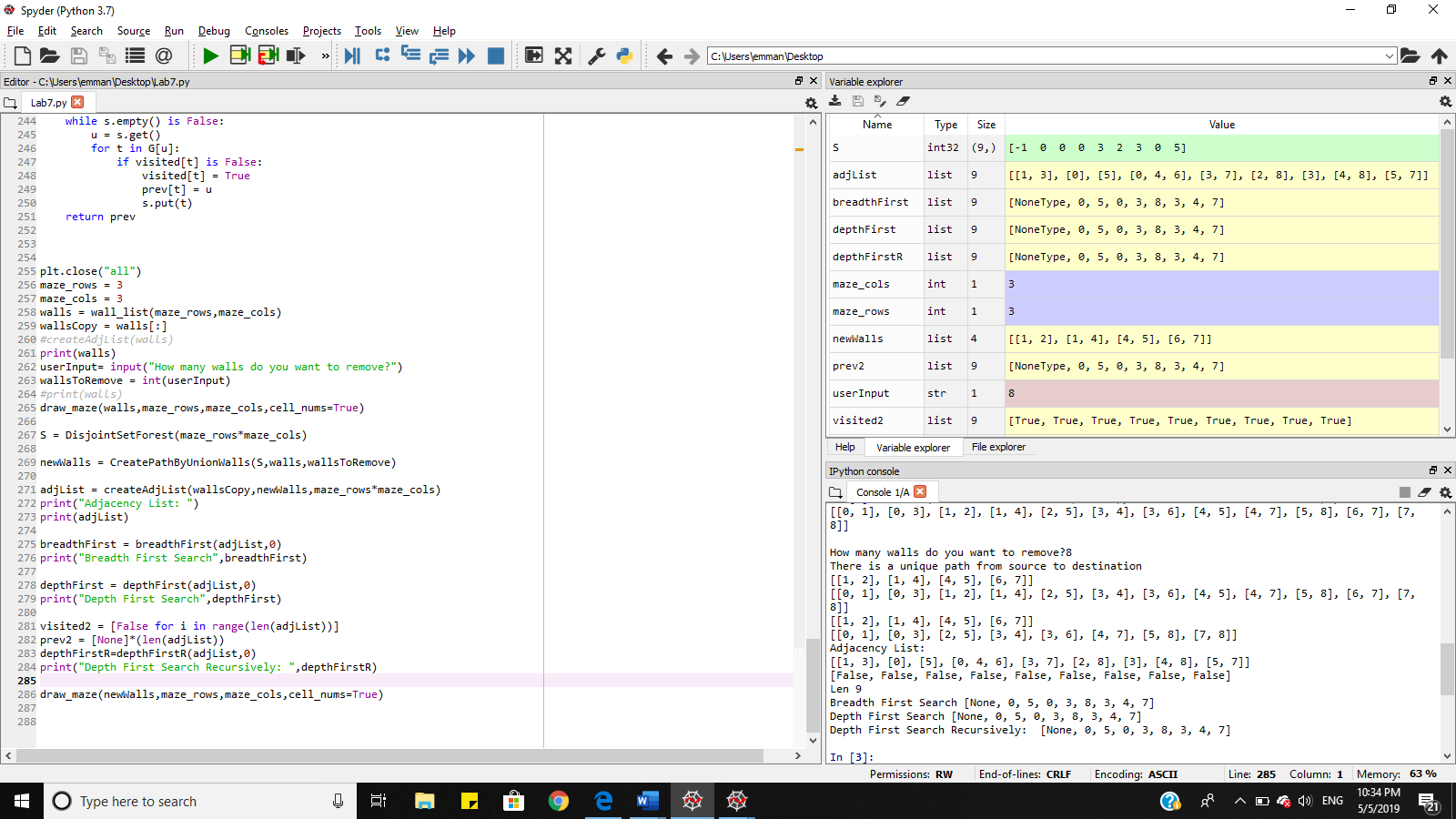
For the breadth first search algorithm, I used a queue. First I added the first cell to the queue, then I dequeue it and enqueue the nodes that it was connected to if there are nodes. This repeated this process until the queue is empty. At this point I had already created the prev array. Each index of the prev array represents a node, and the number that contains this index represents the previous node that you have to visit the current index depending of the search algorithm.

For the depth first search algorithm I did the same procedure of the breadth search algorithm but this time using a stack instead of the queue.

For the depth first search algorithm using recursion, I created a method with the graph and the source cell as parameters. Then if an element of the graph is not visited, then I added it to the prev array and so on until the all the elements of the graph were visited.

**Experimental results**

In order to get a better idea of what I was adding to the prev array and the elements that I had to add to the adjacency list, I printed out different outputs as show below:



**Big O notation**

Breath first search: O(|V| + |E|)

Depth first Search: O(|V| + |E|)

Depth first search recursively: O(|V|+|E|)

**Appendix**

"""

Name: Emmanuel Alvarez

Professor: Olac Fuentes

Last Day modified: 05-05-2019

The purpose of this lab is create a maze and realize a breadth first search,

depth first search and depth first search recursively from the cell 0 to the

last cell of the maze.

"""

import matplotlib.pyplot as plt

import numpy as np

import random

from scipy import interpolate

import time

import queue

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 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

def returnIndexOfWall(walls,i,j): #Returns the index of walls with specific cells

for k in range(len(walls)):

#print(walls[k])

if i in walls[k] and j in walls[k]:

return k

#####################################################################################################

def DisjointSetForest(size):

return np.zeros(size,dtype=np.int)-1

def dsfToSetList(S):

#Returns aa list containing the sets encoded in S

sets = [ [] for i in range(len(S)) ]

for i in range(len(S)):

sets[find(S,i)].append(i)

sets = [x for x in sets if x != []]

return sets

def find(S,i):

# Returns root of tree that i belongs to

if S[i]<0:

return i

return find(S,S[i])

def find\_c(S,i): #Find with path compression

if S[i]<0:

return i

r = find\_c(S,S[i])

S[i] = r

return r

def union(S,i,j,walls):

# Joins i's tree and j's tree, if they are different

ri = find(S,i)

rj = find(S,j)

if ri!=rj:

S[rj] = ri

index = returnIndexOfWall(walls,i,j) #gets the index of cells

walls.pop(index) #removes the wall of specific cells

#wallsToRemove-=1

def unionSameSet(S,i,j,walls):

index = returnIndexOfWall(walls,i,j) #gets the index of cells

walls.pop(index) #removes the wall of specific cells

def union\_c(S,i,j):

# Joins i's tree and j's tree, if they are different

# Uses path compression

ri = find\_c(S,i)

rj = find\_c(S,j)

if ri!=rj:

S[rj] = ri

def union\_by\_size(S,i,j,walls):

# if i is a root, S[i] = -number of elements in tree (set)

# Makes root of smaller tree point to root of larger tree

# Uses path compression

ri = find\_c(S,i)

rj = find\_c(S,j)

if ri!=rj:

if S[ri]>S[rj]: # j's tree is larger

S[rj] += S[ri]

S[ri] = rj

index = returnIndexOfWall(walls,i,j) #gets the index of the cells

walls.pop(index)# removes the wall of those cells

else:

S[ri] += S[rj]

S[rj] = ri

index = returnIndexOfWall(walls,i,j)

walls.pop(index)

def draw\_dsf(S):

scale = 30

fig, ax = plt.subplots()

for i in range(len(S)):

if S[i]<0: # i is a root

ax.plot([i\*scale,i\*scale],[0,scale],linewidth=1,color='k')

ax.plot([i\*scale-1,i\*scale,i\*scale+1],[scale-2,scale,scale-2],linewidth=1,color='k')

else:

x = np.linspace(i\*scale,S[i]\*scale)

x0 = np.linspace(i\*scale,S[i]\*scale,num=5)

diff = np.abs(S[i]-i)

if diff == 1: #i and S[i] are neighbors; draw straight line

y0 = [0,0,0,0,0]

else: #i and S[i] are not neighbors; draw arc

y0 = [0,-6\*diff,-8\*diff,-6\*diff,0]

f = interpolate.interp1d(x0, y0, kind='cubic')

y = f(x)

ax.plot(x,y,linewidth=1,color='k')

ax.plot([x0[2]+2\*np.sign(i-S[i]),x0[2],x0[2]+2\*np.sign(i-S[i])],[y0[2]-1,y0[2],y0[2]+1],linewidth=1,color='k')

ax.text(i\*scale,0, str(i), size=20,ha="center", va="center",

bbox=dict(facecolor='w',boxstyle="circle"))

ax.axis('off')

ax.set\_aspect(1.0)

def CreatePathByUnion(S,walls):

while OneSetInForest(S) != 1: #while there are more than one set

d = random.randint(0,len(walls)-1) # Creates a random number in order to remove a random wall

union(S,walls[d][0],walls[d][1],walls) #joins the cells if they are in different sets

return walls

def CreatePathByUnionSize(S,walls):

while OneSetInForest(S) != 1:

d = random.randint(0,len(walls)-1)

union(S,walls[d][0],walls[d][1],walls)

return walls

def OneSetInForest(S):#checks how many sets are in the forest

counter = 0

for i in range(len(S)):

if S[i] == -1:

counter+=1

return counter

def CreatePathByUnionWalls(S,walls,wallsToRemove):

numOfWalls = len(walls)

if(wallsToRemove<len(S)-1):

print("A path from source to destination is not guaranteed to exist.")

elif(wallsToRemove== len(S)-1):

print("There is a unique path from source to destination")

else:

print("There is at least one path from source to destination")

while len(walls) != numOfWalls - wallsToRemove:# will join the sets even when they are on the same set

d = random.randint(0,len(walls)-1) # Creates a random number in order to remove a random wall

unionSameSet(S,walls[d][0],walls[d][1],walls)

return walls

while len(walls) != numOfWalls-wallsToRemove:

d = random.randint(0,len(walls)-1) # Creates a random number in order to remove a random wall

union(S,walls[d][0],walls[d][1],walls) #joins the cells if they are in different sets

return walls

def createAdjList(walls,removeWalls,cells): #creates the adjacency list

adjList = [[] for i in range(cells)]

print(removeWalls)

print(walls)

for w in range(len(removeWalls)):# creates a array where there is not a wall between two cells

if removeWalls[w] in walls:

index = returnIndexOfWall(walls,removeWalls[w][0],removeWalls[w][1])

walls.pop(index)

print(removeWalls)

print(walls)

for i in range(len(walls)):

firstCell = walls[i][0]

secondCell = walls[i][1]

if firstCell not in adjList[secondCell]: #if the number is not in the index of adjacency list, then append it

adjList[secondCell].append(firstCell)

if secondCell not in adjList[firstCell]:

adjList[firstCell].append(secondCell)

return adjList

def breadthFirst(G, v):

Q = queue.Queue(100)

visited = [False for i in range(len(G))]

print(visited)

prev = [None]\*(len(G))

print("Len",len(prev))

Q.put(v)

visited[v] = True

while Q.empty() is False:

u = Q.get()

for t in G[u]:

if visited[t] is False:

visited[t] = True

prev[t] = u

Q.put(t)

return prev

def depthFirstR(G,source):

visited2[source] = True

for t in G[source]:

if visited2[t] is False:

prev2[t] = source

depthFirstR(G,t)

return prev2

def depthFirst(G, v):

s = queue.LifoQueue(100)

visited = [False for i in range(len(G))]

prev = [None]\*(len(G))

s.put(v)

visited[v] = True

while s.empty() is False:

u = s.get()

for t in G[u]:

if visited[t] is False:

visited[t] = True

prev[t] = u

s.put(t)

return prev

plt.close("all")

maze\_rows = 3

maze\_cols = 3

walls = wall\_list(maze\_rows,maze\_cols)

wallsCopy = walls[:]

#createAdjList(walls)

print(walls)

userInput= input("How many walls do you want to remove?")

wallsToRemove = int(userInput)

#print(walls)

draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True)

S = DisjointSetForest(maze\_rows\*maze\_cols)

newWalls = CreatePathByUnionWalls(S,walls,wallsToRemove)

adjList = createAdjList(wallsCopy,newWalls,maze\_rows\*maze\_cols)

print("Adjacency List: ")

print(adjList)

breadthFirst = breadthFirst(adjList,0)

print("Breadth First Search",breadthFirst)

depthFirst = depthFirst(adjList,0)

print("Depth First Search",depthFirst)

visited2 = [False for i in range(len(adjList))]

prev2 = [None]\*(len(adjList))

depthFirstR=depthFirstR(adjList,0)

print("Depth First Search Recursively: ",depthFirstR)

draw\_maze(newWalls,maze\_rows,maze\_cols,cell\_nums=True)

“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.”