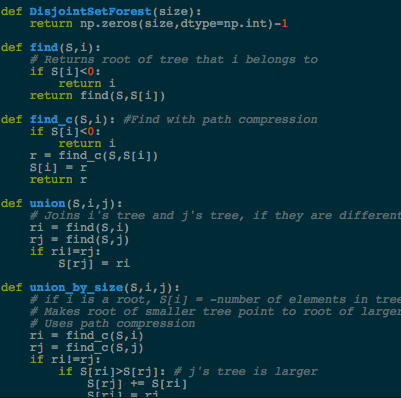
**Lab # 7 Report**

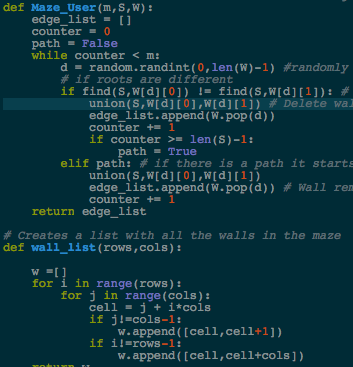
**Introduction:**

﻿ For this lab assignment we were asked to ask the user to input the number of walls that will be removed. we have to build an adjajency list representation of them maze and we have to represent them in three different ways: breadth, first search, depth first search, and a recursive version of depth first search. The main purpose for this lab is to understand how different graphs work.

**Pre-methods:**

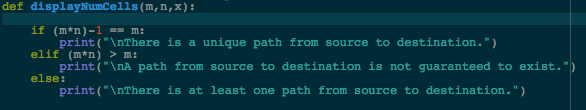
In order to have the methods for the lab completed I needed to call constructors for disjoint set forest. For disjoint set forest I used *union()* and *find()* for some methods. I also used *draw\_maze\_path()* which is in charge of drawing the maze with the use of the matplot library. I needed a method *Maze\_User()* in order to build the maze with an edge list.





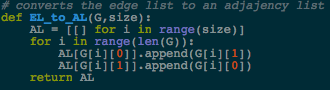
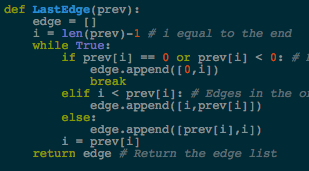
**First part:**

The first task was to create a method, Which I called *DisplayNumCells(),* that help us to know whether if the number walls that want to be removed is greater, equal of lower than number of rows times the number of columns to see if it has a unique path, the path is not guaranteed to exist, or if there is at least one path from source to destination.



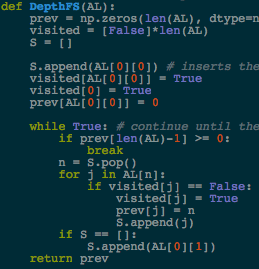
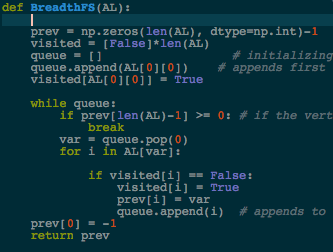
**Second part:**

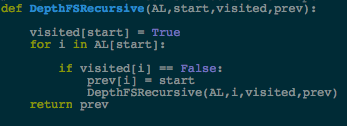
We needed to build an adjajency list representation on the maze, so I did a method *EL\_to\_AL()* which is in charge of converting the edge list representation of the maze to one represented in adjajancy list. After that we needed another method that adds an edge between *u* and *v* in the graph only if v and u are contiguous and there is no wall separating them, so I did a method *LastEdge()* that completes that part of the process by appending the value with a while loop and appending the element depending on its value, and just finally returning the edge list.



**Third part:**

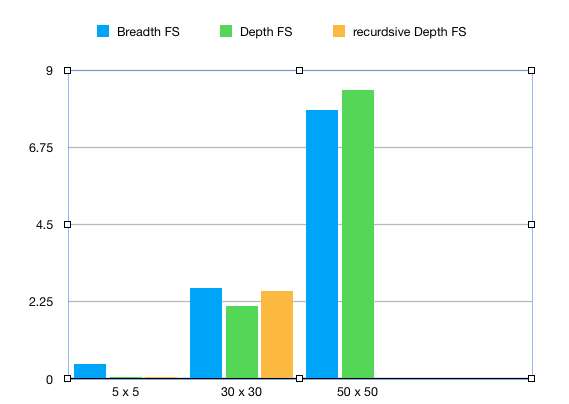
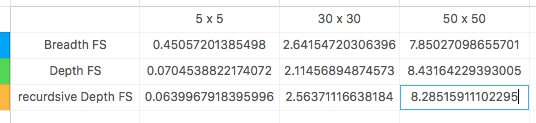
I had to implement the maze in three different representations; Breadth First Search, Depth First search and its recursive version. What I did for Breadth FS was to create a queue and be appending the vertex that has not be visited. For Depth FS it was pretty similar but here we needed to use a stack. Finally, for the recursive version of the Depth FS, if the item is false then the previous vertex is now the origin of the path and recursively we call the method by changing the origin for the index of the loop.

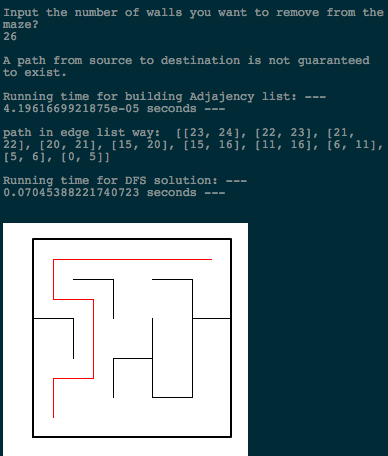
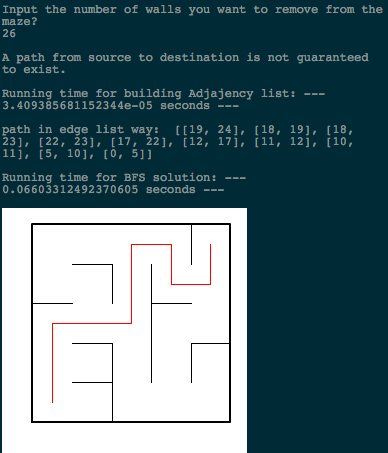


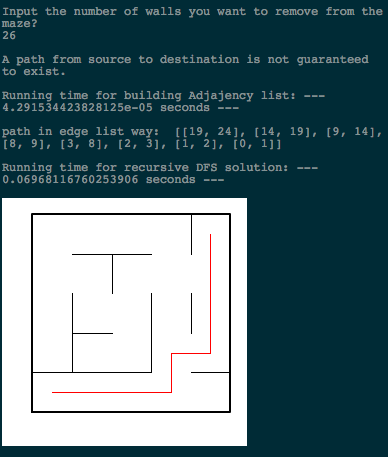


**Conclusion:**

In my opinion, this lab assignment got a good level of difficulty but it is because I’m still learning on how multiple algorithms work, but it help me a lot because now I know how those two algorithms work and I feel more comfortable when I have to work with graphs.







﻿# DAVIS, DAVID A 80610756

# For this lab assignment we were asked to ask the user to input the number of

# walls that will be removed. we have to build an adjajency list representation

# of them maze and we have to represent them in three different ways: breadth,

# first search, depth first search, and a recursive version of depth first search.

# The main purpose for this lab is to understand how different graphs work.

import matplotlib.pyplot as plt

import numpy as np

import random

import time

# ---------------------------- PRE-METHODS -------------------

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 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):

# 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

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

# 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

else:

S[ri] += S[rj]

S[rj] = ri

def removeComp(s, walls, numSets): #remove random parts of the wall

while numSets>1:

w = random.choice(walls) #random wall selection

i = walls.index(w) #position of wall

if find(s, w[0]) != find(s, w[1]): ##if root of w[0] is not the same as w[1]

walls.pop(i) #wall removal

union\_by\_size(s, w[0], w[1]) #wall union after the removal

numSets -=1

return w

def draw\_maze\_path(walls,path,maze\_rows,maze\_cols,cell\_nums=False):

fig, ax = plt.subplots()

for p in path:

if p[1]-p[0] != 1:

# Vertical Path

px0 = (p[1]%maze\_cols)+.5

px1 = px0

py0 = (p[1]//maze\_cols)-.5

py1 = py0+1

else:

# Horizontal Path

px0 = (p[0]%maze\_cols)+.5

px1 = px0+1

py0 = (p[1]//maze\_cols)+.5

py1 = py0

ax.plot([px0,px1],[py0,py1],linewidth=1,color='r')

for w in walls:

if w[1]-w[0] == 1: # Vertical Wall position

x0 = (w[1]%maze\_cols)

x1 = x0

y0 = (w[1]//maze\_cols)

y1 = y0+1

else: # Horizontal Wall postion

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('on')

ax.set\_aspect(1.0)

ax.axis('off')

ax.set\_aspect(1.0)

# ------------------------- METHODS FOR LAB 7 -----------------------

# This method build the maze with the use of edge list

def Maze\_User(m,S,W):

edge\_list = []

counter = 0

path = False

while counter < m:

d = random.randint(0,len(W)-1) #randomly selects a wall in the maze

# if roots are different

if find(S,W[d][0]) != find(S,W[d][1]): # if roots are different then join sets

union(S,W[d][0],W[d][1]) # Delete wall

edge\_list.append(W.pop(d))

counter += 1

if counter >= len(S)-1:

path = True

elif path: # if there is a path it starts removing any walls

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

edge\_list.append(W.pop(d)) # Wall removal

counter += 1

return edge\_list

# Creates a list with all the walls in the maze

def wall\_list(rows,cols):

w =[]

for i in range(rows):

for j in range(cols):

cell = j + i\*cols

if j!=cols-1:

w.append([cell,cell+1])

if i!=rows-1:

w.append([cell,cell+cols])

return w

# converts the edge list to an adjajency list

def EL\_to\_AL(G,size):

AL = [[] for i in range(size)]

for i in range(len(G)):

AL[G[i][0]].append(G[i][1])

AL[G[i][1]].append(G[i][0])

return AL

# Presents the maze in Breadth First Search way

def BreadthFS(AL):

prev = np.zeros(len(AL), dtype=np.int)-1

visited = [False]\*len(AL)

queue = [] # initializing the queue

queue.append(AL[0][0]) # appends first element to the queue

visited[AL[0][0]] = True

while queue:

if prev[len(AL)-1] >= 0: # if the vertex needed is found then breaks

break

var = queue.pop(0)

for i in AL[var]:

if visited[i] == False:

visited[i] = True

prev[i] = var

queue.append(i) # appends to te queue if the elemtn has not been visited

prev[0] = -1

return prev

# Presents the maze in Depth First Search way

def DepthFS(AL):

prev = np.zeros(len(AL), dtype=np.int)-1

visited = [False]\*len(AL)

S = []

S.append(AL[0][0]) # inserts the first element to the queue

visited[AL[0][0]] = True

visited[0] = True

prev[AL[0][0]] = 0

while True: # continue until the vertex has been reached

if prev[len(AL)-1] >= 0:

break

n = S.pop()

for j in AL[n]:

if visited[j] == False:

visited[j] = True

prev[j] = n

S.append(j)

if S == []:

S.append(AL[0][1])

return prev

# This method is a recursive version of Depth First Search

def DepthFSRecursive(AL,start,visited,prev):

visited[start] = True

for i in AL[start]:

if visited[i] == False:

prev[i] = start

DepthFSRecursive(AL,i,visited,prev)

return prev

def LastEdge(prev):

edge = []

i = len(prev)-1 # i equal to the end

while True:

if prev[i] == 0 or prev[i] < 0: # Base case when prev is equal to 0 or less

edge.append([0,i])

break

elif i < prev[i]: # Edges in the order of small,big

edge.append([i,prev[i]])

else:

edge.append([prev[i],i])

i = prev[i]

return edge # Return the edge list

# This method displays if there is a unique path, a path from source of to destination

# and if there is at least one pth from source if destination

def displayNumCells(m,n,x):

if (m\*n)-1 == m:

print("\nThere is a unique path from source to destination.")

elif (m\*n) > m:

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

else:

print("\nThere is at least one path from source to destination.")

plt.close("all")

maze\_rows = 50

maze\_cols = 50

x = input('Input the number of walls you want to remove from the maze?\n')

x = int(x)

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

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

displayNumCells(maze\_rows, maze\_cols, x)

print()

EL = Maze\_User(x,S,walls)

time1 = time.time()

AL = EL\_to\_AL(EL,maze\_cols\*maze\_rows) # convert edge list to AL list

print('Running time for building Adjajency list: '"--- %s seconds ---" % (time.time() - time1))

print()

# DISPLAYS THE MAZE WITH DEPTH FIRST SEARCH

#time1 = time.time()

#prev = DepthFS(AL)

#path = (LastEdge(prev))

#print('path in edge list way: ',path)

#draw\_maze\_path(walls,path,maze\_rows,maze\_cols)

#print()

#print('Running time for DFS solution: '"--- %s seconds ---" % (time.time() - time1))

#print()

# DISPLAYS THE MAZE WITH USING BREADTH FIRST SEARCH

#time1 = time.time()

#prev = BreadthFS(AL) # Function ends when goal has been reached to shorten time

#path = (LastEdge(prev))

#print("path in edge list way: ",path)

#draw\_maze\_path(walls,path,maze\_rows,maze\_cols)

#print()

print('Running time for BFS solution: '"--- %s seconds ---" % (time.time() - time1))

# DISPLAYS THE MAZE WITH A RECURSIVE VERSION OF DEPTH FIRST SEARCH

time1 = time.time()

visited = [False]\*len(AL)

p = np.zeros(len(AL),dtype=int)-1

prev = DepthFSRecursive(AL,0,visited,p)

path = (LastEdge(prev))

print("path in edge list way: ",path)

draw\_maze\_path(walls,path,maze\_rows,maze\_cols)

print()

print('Running time for recursive DFS solution: '"--- %s seconds ---" % (time.time() - time1))

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.

* David A. Davis