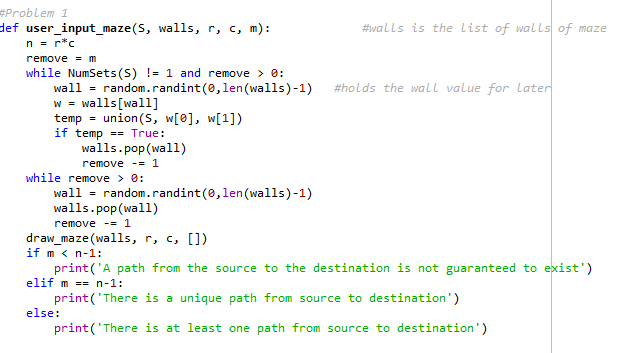
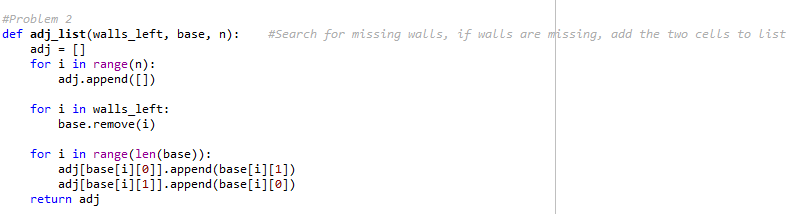
Lab 7

The purpose of this lab is to practice with graphs more, which includes determining whether or not a solution to the maze exists, and whether or not there are multiple solutions. Then we have to create an adjacency list based off that maze. And lastly, we have to be able to traverse the maze and reach the end using Breadth-first search, Depth-First search, and Depth-first search using recursion. And lastly, we have to show the paths taken by the searches.

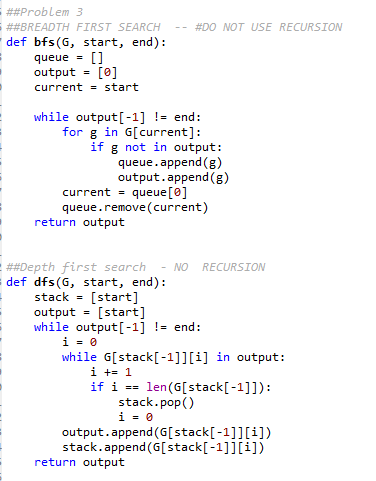
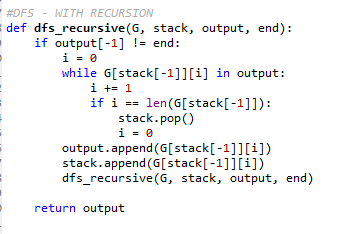
For my first problem, I edited the code so that m is equal to the number of cells there are, and n being a user-input number that removes n walls. It removes the walls given by the user then determines if there is a unique path, potentially more than one path, or potentially no paths.

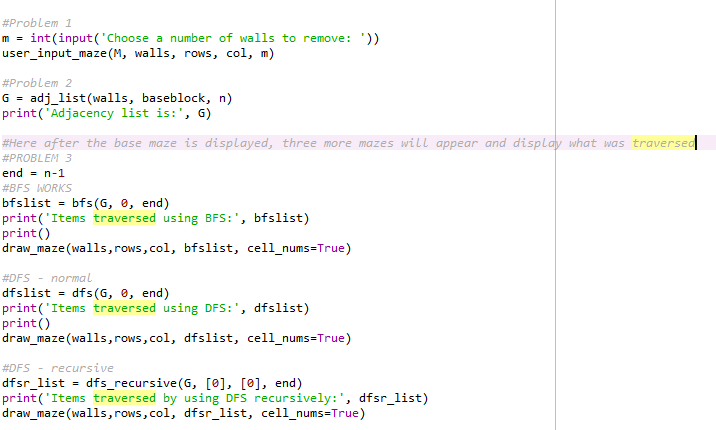


For my second problem, I created a method that searches for missing walls, using the list of current walls within the program, and the list of initial walls before they were removed. I then find the two cells that the missing walls separated, and fit them into the adjacency list.

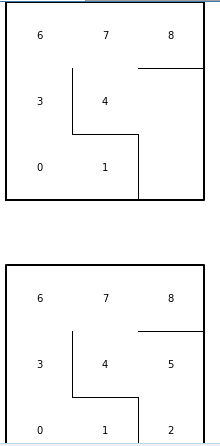


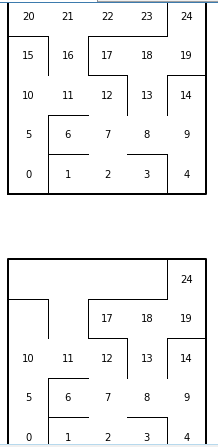
Third problem was an issue for me. We needed to properly traverse the graph using BFS and DFS. A big problem I faced was that I was trying to store all the values, without using a queue or a stack. I was also trying to create a clear-cut path to the end rather than store all the values traversed by BFS and DFS. Once I implemented a Queue/Stack and visited lists, it let me stay better organized while trying to traverse the graph. And as for DFS with recursion, I replaced my while loop with an if statement, and called back on the method to go through a loop, until I touched the end of the maze.

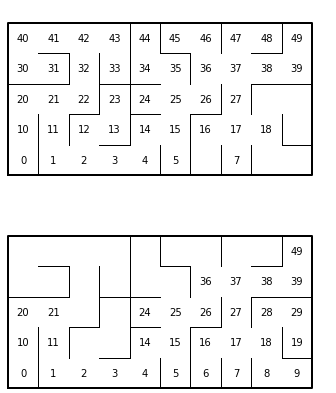
Calls made for the program

In conclusion, I need to better understand the pseudo-code involved with DFS and BFS. While I got the assignment done, I feel like it could be much cleaner than it is now, as well as much more efficient. I also want to return to this and figure out how to get the clear-cut path to the end without having the side paths taken by the searches pop up on display. But from what I did learn, I feel like I can handle these search algorithms much better than before I started.









**APPENDIX:**

#Patrick Brannan

#Last Edited 5/4/2019

#Purpose of program is work on traversing the graph using BFS and DFS, while

# also creating a Adj. List.

import matplotlib.pyplot as plt

import numpy as np

import random

import time

def draw\_maze(walls,maze\_rows,maze\_cols,path, 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: #Changed to only draw the numbers along the path

for r in range(maze\_rows):

for c in range(maze\_cols):

cell = c + r\*maze\_cols

if cell in path:

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, S):

# 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 DisjointSetForest(size):

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

def find(S,i):

if S[i]<0:

return i

return find(S,S[i])

def find\_c(S,i):

if S[i]<0:

return i

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

S[i] = r

return r

def union(S,i,j): #joins set if different

ri = find(S,i)

rj = find(S,j)

if ri!=rj:

S[rj] = ri

return True

else:

return False

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

return True

else:

return False

def NumSets(S):

count =0

for i in range(len(S)):

if S[i]<0:

count += 1

return count

#Problem 1

def user\_input\_maze(S, walls, r, c, m): #walls is the list of walls of maze

n = r\*c

remove = m

while NumSets(S) != 1 and remove > 0:

wall = random.randint(0,len(walls)-1) #holds the wall value for later

w = walls[wall]

temp = union(S, w[0], w[1])

if temp == True:

walls.pop(wall)

remove -= 1

while remove > 0:

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

walls.pop(wall)

remove -= 1

draw\_maze(walls, r, c, [])

if m < n-1:

print('A path from the source to the destination is not guaranteed to exist')

elif m == n-1:

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

else:

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

#Problem 2

def adj\_list(walls\_left, base, n): #Search for missing walls, if walls are missing, add the two cells to list

adj = []

for i in range(n):

adj.append([])

for i in walls\_left:

base.remove(i)

for i in range(len(base)):

adj[base[i][0]].append(base[i][1])

adj[base[i][1]].append(base[i][0])

return adj

##Problem 3

##BREADTH FIRST SEARCH -- #DO NOT USE RECURSION

def bfs(G, start, end):

queue = []

output = [0]

current = start

while output[-1] != end:

for g in G[current]:

if g not in output:

queue.append(g)

output.append(g)

current = queue[0]

queue.remove(current)

return output

##Depth first search - NO RECURSION

def dfs(G, start, end):

stack = [start]

output = [start]

while output[-1] != end:

i = 0

while G[stack[-1]][i] in output:

i += 1

if i == len(G[stack[-1]]):

stack.pop()

i = 0

output.append(G[stack[-1]][i])

stack.append(G[stack[-1]][i])

return output

#DFS - WITH RECURSION

def dfs\_recursive(G, stack, output, end):

if output[-1] != end:

i = 0

while G[stack[-1]][i] in output:

i += 1

if i == len(G[stack[-1]]):

stack.pop()

i = 0

output.append(G[stack[-1]][i])

stack.append(G[stack[-1]][i])

dfs\_recursive(G, stack, output, end)

return output

plt.close("all")

rows = 5

col = 10

n = rows\*col

print('CREATING MAZES WITH DIMENSIONS', rows, 'X', col)

#Start of randomized maze

M = DisjointSetForest(rows\*col)

walls = wall\_list(rows,col,M)

baseblock = wall\_list(rows, col, M)

#Problem 1

m = int(input('Choose a number of walls to remove: '))

user\_input\_maze(M, walls, rows, col, m)

#Problem 2

G = adj\_list(walls, baseblock, n)

print('Adjacency list is:', G)

#Here after the base maze is displayed, three more mazes will appear and display what was traversed

#PROBLEM 3

end = n-1

#BFS WORKS

start = time.time()

bfslist = bfs(G, 0, end)

print('Items traversed using BFS:', bfslist)

print()

draw\_maze(walls,rows,col, bfslist, cell\_nums=True)

bfstime = time.time() - start

#DFS - normal

start = time.time()

dfslist = dfs(G, 0, end)

print('Items traversed using DFS:', dfslist)

print()

draw\_maze(walls,rows,col, dfslist, cell\_nums=True)

dfstime = time.time() - start

#DFS - recursive

start = time.time()

dfsr\_list = dfs\_recursive(G, [0], [0], end)

print('Items traversed by using DFS recursively:', dfsr\_list)

draw\_maze(walls,rows,col, dfsr\_list, cell\_nums=True)

dfsrtime = time.time() - start

print('BFS time with', n, 'size is:',bfstime)

print('DFS time with', n, 'size is', dfstime)

print('DFS with recurstion time with', n, 'size is', dfsrtime)

***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 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.***

***-Patrick Brannan***