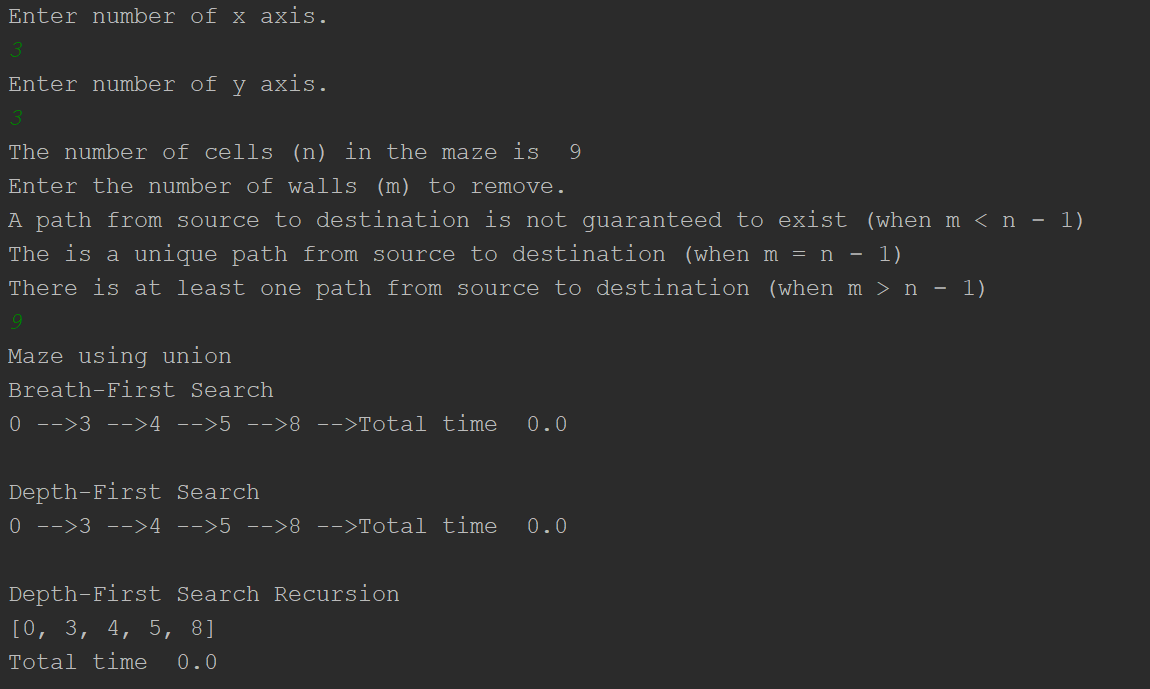
Lab 7 is about modifying the maze building program that we did for lab 6. The modifications included to display the number of cells (n) and ask the user for the number of walls to be removed (m). After that is done you tell the user that depending if ‘m’ is greater, equal, or less than ‘n’ then the possible solutions to the maze is more than one, only one path, and a path to the destination is not guaranteed accordingly. After we had to solve the maze starting at the bottom left corner of the maze to the top corner of the maze. We accomplish this by using the breadth-first search, depth-first search, and depth-first search using recursion given an adjacency list as input.

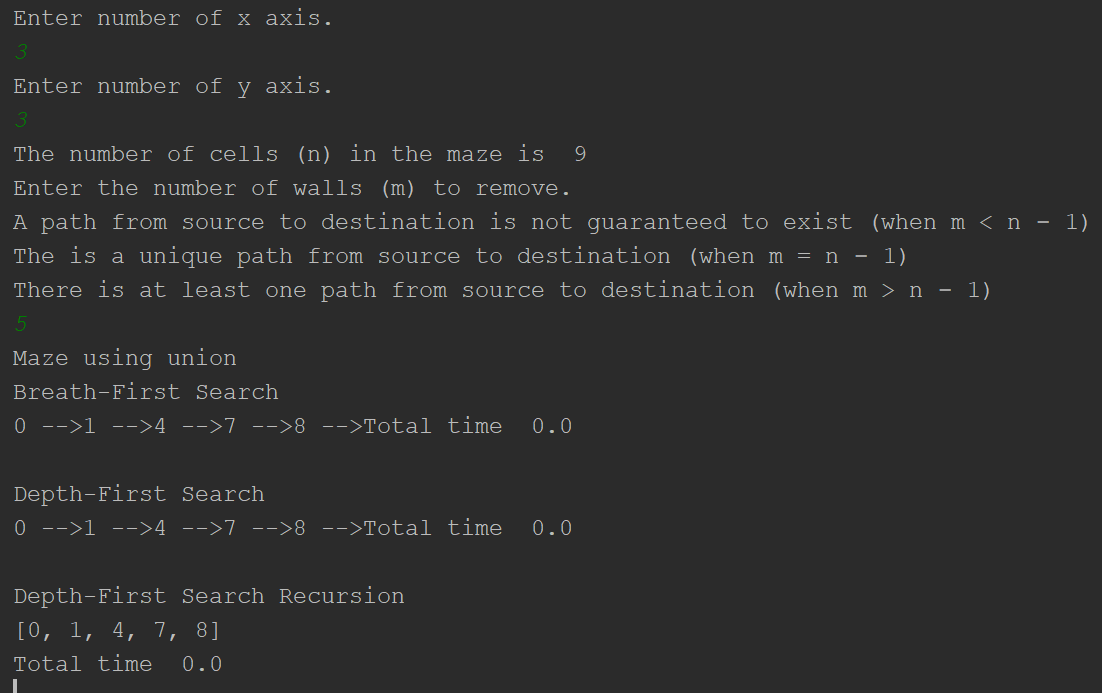
Since we had the maze creation algorithm already it was really simple to modify. All i had to do was ask the user of the size inputs to the maze and print out the ‘n’ number of cells. After I ask the user to pass an integer ‘m’ that will be the number of walls removed and warning them of how m compares to n can affect the solutions. Next just the same as when we created our maze in lab 6 we create our maze using a disjoint set forest but now every time we add a connection to the disjoint set forest we add the connection to the adjacency list too. The way I did this is by creating a class called Graph. Graph contains the number of vertex in the maze in the variable ‘vertices’ and the adjacency list in the variable ‘graph’. The way you add the connections is by using the method add\_edge() that takes 3 variables G, V1, and V2 that is the Graph vertice one and vertice two. This way we can access the variable graph that is an array size of the number of vertices. Then using V1 as our index to the array we store V2 to show the connection. The inputs I pass into the method is my object Graph and Choice[0] and Choice[1] that represent my y and x axis in my maze.

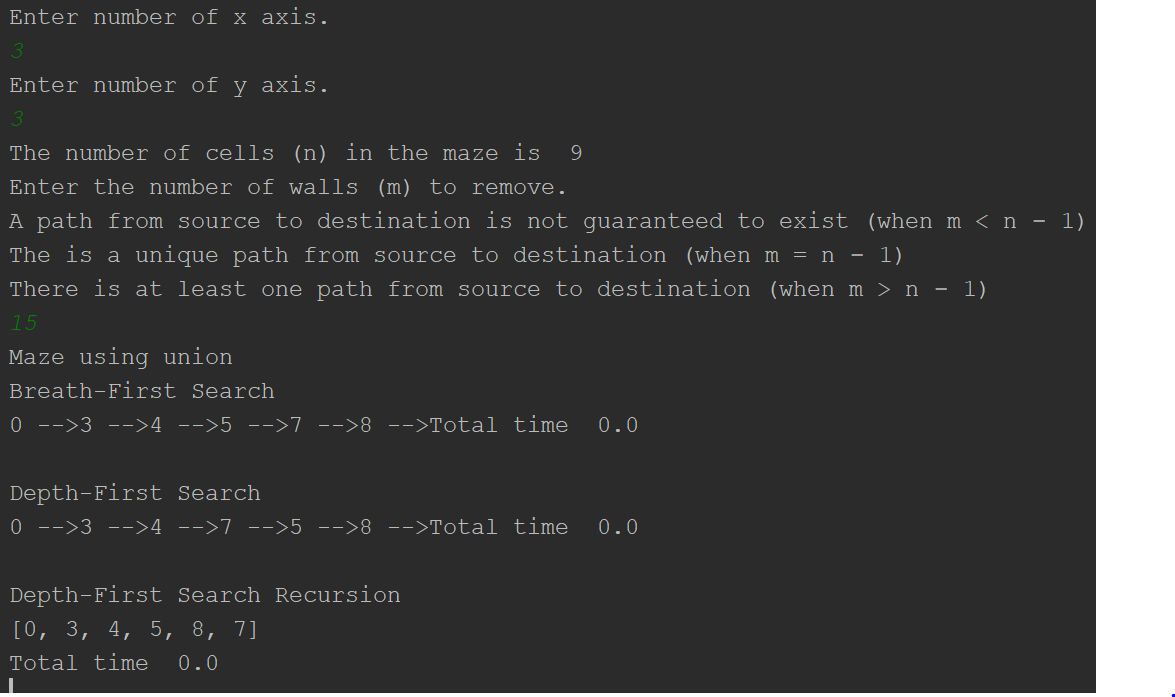
Now that we have created our adjacency list and maze we can start searching for possible solutions from the bottom left to the top right corner. The first way to solve it is by using breadth-first search. This is an algorithm to traverse a graph. The way it works is that we have a list that is made up of booleans stating if we have visited the vertiec yet or not. If we have we set it as true if not we leave it at false. We then have a queue that stores all the vertices that we have not visited so that we can add to the visited ones. For this I made the function BFS that take G and start. G is the graph and start is my initial veritice in this case it will always be 0 as that will be my starting place in the maze. Inside the function i initialize the variable visited with all be false at the start with the exception of our starting vertice and my queue with the vericie start. Then in a while loop stating that while the queue still has elements we pop them and save them as our new starting place we change the state of the visited list to True. after inside the while loop we have a for loop that will traverse all the vertices that are connected and if they have not been visited yet we add them to our queue. This goes on till all the variables in the queue are popped. At the end I print out the results of all the vertices that are connected to the starting vertices.

The next algorithm is depth-first search. Depth-first search explores possible vertices from the starting vertex down each branch before backtracking. It also uses a stack instead of a queue. In my program I made the function DFS() that takes G and start. G is the adjacency list and start is my initial vertex. The way it works is by having a list call visited and a stack. We initialize the visited with an empty list and the stack with only the starting vertex. Then using a while loop we pop the first thing in the stack and add it to the visited list. Then using a for loop to traverse the vertex at thee position of start we add the vertex that are connected to it. At the end we print out the visited list.

The last algorithm would be depth-first search using recursion. Just like regular depth-first search it also explores possible vertices from the starting vertex down each branch before backtracking. But instead of using a for loop it uses a recursive call. In my program I made DFSR() method that takes three inputs G start and visited that is initialize with None. The way it works is also similar. It too has a visited list but this visited list is imputed as an input for the function as to keep track in the recursive call but it does not have a stack. Insted after the first run that initializes visitors to an empty list we add our starting vertex and enter into a for loop. Inside the for loop we traverse all the vertex in the adjacency list at position start and if they are not in the visited list then we call our recursive call with the Graph, the next vertex, and the new visited list. In the end we return the visited list after all vertex that are connected have been visited.







# Lab 7

# Programmed by Anthony Herrera

# Last modified April, 28, 2019

import matplotlib.pyplot as plt

import numpy as np

from scipy import interpolate

import random

import time

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):

# 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\_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):

# 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 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 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 NumberOfSets(S):# will count the number of -1 that show diffrent sets

count = 0

for x in S:

if x == -1:

count += 1

return count

class Graph: # creates a graph object with vertexes

def \_\_init\_\_(self,vertices):

self.vertices = vertices

self.graph = []

for v in range(vertices): # append unvisited vertexes

self.graph.append([])

def add\_edge(G,v1,v2): # add edges to the graph

G.graph[v1].append(v2)

def BFS(G,start):

visited = [False] \* (len(G.graph)) # unvisited array

queue = []

queue.append(start) # Queue off unvisited vertexes

visited[start] = True

while queue:

start = queue.pop(0)

print(start,'-->', end='')

for i in G.graph[start]: # add unvisited vertex to queue

if visited[i] == False:

queue.append(i)

visited[i] = True

def DFS(G,start):

visited = [] # creates array for vested vertexes

stack = [start] # stack that hold vertexes to be added

while stack:

start = stack.pop()

visited.append(start)

print(start,'-->', end='')

for i in G.graph[start]: # append unvisited vertexes

stack.append(i)

def DFSR(G,start,visited=None):

if visited == None: # creates array of vertexes that have been visited

visited = []

visited.append(start) # appends vertex to visited list

for i in G.graph[start]: # traverse and add unvisited vertexes

if i not in visited:

DFSR(G,i,visited)

return visited

plt.close("all")

maze\_rows = int(input('Enter number of x axis.\n'))

maze\_cols = int(input('Enter number of y axis.\n'))

graph = Graph(maze\_rows\*maze\_cols)

print('The number of cells (n) in the maze is ',maze\_cols\*maze\_rows)

wall\_remover = int(input('Enter the number of walls (m) to remove.\nA path from source to destination is not guaranteed to exist (when m < n − 1)\nThe is a unique path from source to destination (when m = n − 1)\nThere is at least one path from source to destination (when m > n − 1)\n'))

wall\_remover += 1

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

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

disjoint\_set\_forest = DisjointSetForest(maze\_rows\*maze\_cols)

print('Maze using union')

timer\_0 = time.time()

while wall\_remover > 0: # will create the maze using stander union of sets

choice = random.choice(walls) # selects random wall from list

index = walls.index(choice) # return index of wall

if find(disjoint\_set\_forest,choice[0]) != find(disjoint\_set\_forest,choice[1]):

walls.pop(index) # deletes the wall selected

union(disjoint\_set\_forest,choice[0],choice[1]) # add the wall to a set using union

#print('V1=',choice[0],'V2=',choice[1])

add\_edge(graph,choice[0],choice[1])

wall\_remover -= 1 # decreases the number of sets by one

timer\_1 = time.time()

#print('Total time ', timer\_1-timer\_0)

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

#print(graph.graph)

print('Breath-First Search')

timer\_0 = time.time()

BFS(graph,0)

timer\_1 = time.time()

print('Total time ', timer\_1-timer\_0)

print('\nDepth-First Search')

timer\_0 = time.time()

DFS(graph,0)

timer\_1 = time.time()

print('Total time ', timer\_1-timer\_0)

print('\nDepth-First Search Recursion')

timer\_0 = time.time()

print(DFSR(graph,0))

timer\_1 = time.time()

print('Total time ', timer\_1-timer\_0)

plt.show()

Academic dishonesty includes but is not limited to cheating, plagiarism and collusion. Cheating may involve copying from or providing information to another student, possessing unauthorized materials during a test, or falsifying data (for example program outputs) in laboratory reports. Plagiarism occurs when someone represents the work or ideas of another person as his/her own. Collusion involves collaborating with another person to commit an academically dishonest act. Professors are required to - and will - report academic dishonesty and any other violation of the Standards of Conduct to the Dean of Students.

* Anthony Herrera