**Introduction**

The assignment’s purpose was to teach how to use Breadth-First Search, Depth-First Search and recursive Depth-First Search to produce a solution to a randomized maze.

**Proposed Solution Design and Implementation**

Using Breadth-First Search, the function should produce a function somewhat as effectively as Depth-First Search. The functions defer as Breadth-First Search uses a Queue

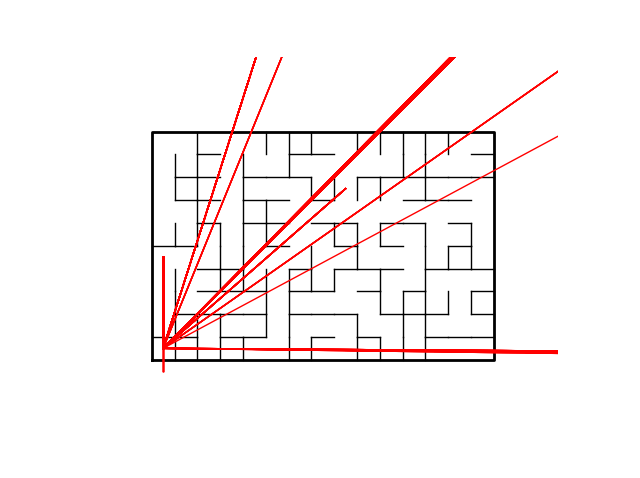
To iteratively produce the successful Depth-First Search function, it is simply the same method as the Breadth-First Search, except the Depth-First Search uses a Stack. In Python, a stack module could not be found, and with a bit of research, it was realized that native-Python lists can be manipulated as stacks. This is done by using using the pop() and insert() method. The insert method should always be set to the first index, such that, insert(0,k), where k is an arbitrary character created by the programmer.

The Depth-First Search with recursion was a more difficult challenge to approach. Following the pseudo-code given during the lecture, it was possible to try and replicate it in Python, however, adding every element to the array was difficult and was not able to be completed.

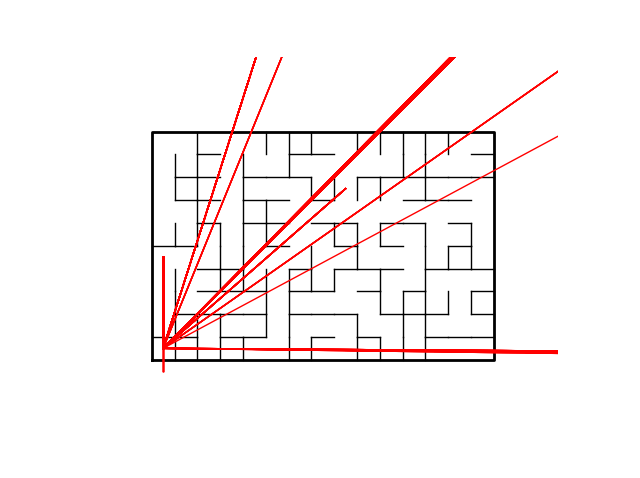
**Experimental Results**

Breadth First Search:

A random number was given, this result is to show how the search and/or the adjacency list were flawed.



Depth-First Search:



**Conclusions**

Having completed this assignment somewhat poorly, many things can be learned from this lesson. Firstly, there should be more studying done with disjoint set forests as well as Graph Theory. Secondly, the use of Python’s list manipulation still needs to be perfected. In conclusion, the use of Python’s lists and disjoint set forests can be improved indefinitely.

**Appendix**

"""

Course: CS 2302

Author: Jacob Montenegro

Lab: 7

Instructor: Dr. Olac Fuentes

T.A.: Anindita Nath, Maliheh Zargaran

Date of Last Modification: 4/29/2019

Program's Purpose: Using Disjoint Set Forests and Graph Theory, produce a randomized

maze and complete it with depth-first search, breadth-first search

and depth-first search using recursion.

"""

# Implementation of disjoint set forest

# Programmed by Olac Fuentes

# Last modified March 28, 2019

import matplotlib.pyplot as plt

import numpy as np

from scipy import interpolate

import random, math, queue

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,draw=True):

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

#draws the maze and path

def draw\_path(walls,r,c,prev,v):

fig, ax = plt.subplots()

for w in walls:

if w[1]-w[0] ==1: #vertical wall

x0 = (w[1]%c)

x1 = x0

y0 = (w[1]//c)

y1 = y0+1

else:#horizontal wall

x0 = (w[0]%c)

x1 = x0+1

y0 = (w[1]//c)

y1 = y0

ax.plot([x0,x1],[y0,y1],linewidth=1,color='k')

sx = c

sy = r

ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k')

draw\_path1(walls,r,c,prev,prev[v],ax)

ax.axis('off')

ax.set\_aspect(1.0)

def draw\_path1(walls,r,c,prev,v,ax):

#ensures that the recursive call doens't become an infinte loop

if r == 1000:

return

if prev[v] != -1 :

draw\_path1(walls,r+1,c,prev,prev[v],ax)

#.5 is added to make sure the path is drawn in the center of each cell.

ax.plot((prev[v]+.5,.5),(prev[v+1]+.5,.5),linewidth=1,color='r')

else:

#moves to next cell if negative 1, this ensures that a path is drawn.

draw\_path1(walls,r+1,c,prev,prev[random.randint(0,len(prev)-1)],ax)

#checks if items in same set

def SameSet(S,i,j):

return find(S,i) == find(S,j)

#sameSet with path compression

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

return find\_c(S,i) == find\_c(S,j)

def numSets(S):

c = 0

for s in S:

if s < 0:

c += 1

return c

def draw\_graph(G):

fig, ax = plt.subplots()

n = len(G)

r = 30

coords =[]

for i in range(n):

theta = 2\*math.pi\*i/n+.001 # Add small constant to avoid drawing horizontal lines, which matplotlib doesn't do very well

coords.append([-r\*np.cos(theta),r\*np.sin(theta)])

for i in range(n-1):

for dest in G[i]:

ax.plot([coords[i][0],coords[dest][0]],[coords[i][1],coords[dest][1]],

linewidth=1,color='k')

for i in range(n):

ax.text(coords[i][0],coords[i][1],str(i), size=10,ha="center", va="center",

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

ax.set\_aspect(1.0)

ax.axis('off')

def Start(m,n):

#computes whether the user's input will create a maze with a path to the end or not.

if m < n - 1:

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

if m == n - 1:

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

if m > n - 1:

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

def AdjacencyList(origin,walls):

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

AL = []

for i in range(len(walls)):

#print(len(AL[i]))

AL.append([])

for j in range(len(walls[i])):

#the up,down,left, and right variables were created to check if two cells

#are contiguous, as well as to check if these cells have a wall seperating them or not

up = i+1

left = j-1

right = j+1

down = i-1

if not origin[up][j] in walls[i] and (origin[up-1][j] == origin[i][j]):

AL[i].append(origin[up][j])

if not origin[down][j] in walls[i] and (origin[down+1][j] == origin[i][j]):

AL[i].append(origin[down][j])

if not origin[i][left] in walls[i] and (origin[i][left+1] == origin[i][j]):

AL[i].append(origin[i][left])

if right< len(walls[i]) and not origin[i][right] in walls[i] and (origin[i][right-1] == origin[i][j]):

AL[i].append(origin[i][right])

return AL

#This function was taken directly from the pseudo code given to us

def BreadthFirst(G,v):

visited = [False]\*len(G)

prev = [-1]\*len(G)

Q = queue.Queue()

Q.put(v)

visited[v] = True

while not Q.empty():

u = Q.get()

for t in G[u]:

if not visited[t]:

visited[t] = True

prev[t] = u

Q.put(t)

return prev

#This function was taken directly from the pseudo code given to us

def DepthFirst(G,v):

visited = [False]\*len(G)

prev = [-1]\*len(G)

#A list is used here in place of a stack since when I searched online for useable stack

#modules in Python, I found none, however I did find that a simple list could be used as a stack

#simply insert every new value into the front, then pop values from the end.

S = []

S.append(v)

visited[v] = True

while len(S)>0:

u = S.pop()

print(u)

for t in G[u]:

if not visited[t]:

visited[t] = True

prev[t] = u

S.append(t)

return prev

#This function was taken directly from the pseudo code given to us

def Depth\_recursion(G,source):

global visited

global prev

visited[source] = True

print(visited)

for t in G[source]:

if not visited[t]:

prev[t] = source

Depth\_recursion(G,t)

def Standard():

global visited

global prev

plt.close("all")

maze\_rows = 10

maze\_cols = 15

#creates a list of all the walls

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

#creates sets of disjoint set forests

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

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

#produces the number of cells in the maze

n = maze\_rows\*maze\_cols

m = int(input('Enter a number\n'))

#returns whether the user's input will create a maze with a path to the end or not

Start(m,n)

#creates a list of all the original walls

origin = []

for i in range(len(walls)):

for j in range(len(walls[i])):

origin.append([])

origin[i].append(walls[i][j])

origin = origin[:len(origin)//2]

i = 0

#removes the number of walls m provided by the user, such that the number of sets always stays at 1

while m > 0:

i+=1

if i % 100000 == 0:

break

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

if not SameSet(S,walls[d][0],walls[d][1]):

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

walls.pop(d)

m-=1

i=0

#removes the rest of the walls (assuming that the user's input exceeded the number of cells in the maze)

while m > 0:

i+=1

if i % 100000 == 0:

break

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

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

walls.pop(d)

m-=1

#returns an adjacency list

L = AdjacencyList(origin,walls)

visited = [False]\*len(L)

prev = [-1]\*len(L)

#draws the new maze

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

prev = BreadthFirst(L,0)

prev = DepthFirst(L,0)

Depth\_recursion(L,0)

#draws the new maze with the path

draw\_path(walls,maze\_rows,maze\_cols,prev,0)

Standard()

**I certify that this project is entirely my own work. I wrote, debugged, and tested the code 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. Signed, Jacob M. Montenegro.**