Lab 7 Report

Introduction:

In this lab we were tasked to use depth first search and breadth first search for the lab 6 graph.

Proposed Solution:

You need to create a queue to go through the breadth first search algorithm. The algorithm goes through the graph and adds it to the queue. Make a stack to go through the depth first search. Both algorithms are O(|V|+|E|). The breadth first search algorithm will get the neighbors of the vertex and add them to the queue. It will eventually pop that element. A stack will grab a vertex and add to the stack until you are out of vertices.

Conclusion:

This lab teaches us how to use the algorithm design techniques. Makes us think critically.

Appendix:

import matplotlib.pyplot as plt

import numpy as np

import random

import queue

def DisjointSetForest(size):

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

def breadth\_first\_search(G, v):

visited = np.arange(len(G)) \* False

prev = np.arange(len(G)) \* -1

Q = []

Q.enqueue(v)

while len(Q)>= 1:

u = Q.pop()

for t in range(len(G[u])):

if visited[t] != True:

visited[t] == True

prev[t] = u

Q.enqueue(t)

return prev

def depth\_first\_search(G,source):

visited = np.arange(len(G)) \* False

visited[source] = True

for t in G[source]:

if visited[t] != True:

prev[t] = source

return prev

def depth\_first\_search\_r(G,source):

visited = np.arange(len(G)) \* False

visited[source] = True

for t in G[source]:

if visited[t] != True:

prev[t] = source

depth\_first\_search(G,t)

def find(S,i):

# Returns root of tree that i belongs to

if S[i]<0:

return i

return find(S,S[i])

def union(S,i,j):

ri = find(S,i)

rj = find(S,j)

if ri!=rj:

S[rj] = ri

def find\_c(S,i):

if S[i]<0:

return i

else:

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

S[i] = r

return r

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[rj] = rj

else:

S[ri] += S[rj]

S[rj] = ri

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

plt.close("all")

maze\_rows = 10

maze\_cols = 15

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

print(walls)

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

remove= input("How many walls would you like to remove")

if int(remove) > len(walls):

print("There is at least 1 path from your source to your destination")

if int(remove) < len(walls):

print("There is not a path from source to destination guaranteed")

if int(remove) == len(walls):

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

S = DisjointSetForest(len(walls))

num\_sets = len(S)

while num\_sets>1:

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

print('removing wall ',walls[d])

walls.pop(d)

num\_sets = num\_sets-1

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

Academic Agreement:

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.

Andres Arellanes