Lab 7 asked us to expand upon lab 6, which had us building mazes. Now we modify the program to remove m walls where m is a user input, and display an output describing the maze based upon m. Next build an adjacency list representation of the maze we create and implement breadth-first search and depth first search with a stack and recursively. Then show the solution to the maze in final drawing.

Approaching the input of m and wall removal, this was a trivial matter, I edited my while loop from the previous program to now run until m, the walls the user wants removed, was 0. I would only remove 1 from m if a wall was able to be removed, and if m was greater than n-1, where n is the number of cells, then begin to remove walls at random. I would display alongside this a statement saying if there was a path from any one source to the destination, or if there may not be one dependent on m’s relation to n-1. If m is less than n-1 then the path is not guaranteed, if it is equal to n-1 then there will be a unique path and if m is greater than or equal to n then there may be more than one path from a source cell to a destination.

Next, we had to build an adjacency list representation of the new maze, this was not too difficult I first created an empty list of lists that was n cells long where n is the complete size of the maze. I then took advantage of how we create the maze and when a wall is removed, I make the cells that the wall is separating now be adjacent to each other in the adjacency list, that’s all I did.

Onto the breadth first algorithm, I created a queue that started with zero and an array that was the same size as the adjacency list. Then while the queue was not empty I would see if the depth of the next value was input into the list already or not, if not then I add it to the queue and list the node that points to it and it’s distance from the zero node and pop the first item in the queue to move onto the next item in the list, since breadth first is not done in a particular order and follows however items enter the queue it fills out paths fairly quickly. Following the breadth first algorithm is the depth first algorithm which is very similar to the breadth first in our case since all route from one cell to another have equal weight. Because of that we only prioritize cells with lower values that were most recently entered into a stack by inputting the higher value items into the stack before the smaller values and popping the top of the stack which has the most recent minimum value available. Doing it recursively was similar only changing how it handled the Adjacency list by removing items and stopping when there were no more items left and concatenating D, the depth list of cells in the end.

Showing the solution wasn’t too hard I only modified the function used to draw the maze and after the maze it would draw a line from the n-1 cell down to the 0 cell by tracing the depth lists created from the previous algorithms and drawing a line from the current node it’s at to the previous node until the 0 node was reached.

Testing wasn’t too bad on the first part of this, the only trouble I ran into was creating incomplete mazes when they were supposed to be complete which was a simple error. I ran into this error as I would decrease m regardless if I removed a wall or not which is wrong, it should only be decreased when a wall is removed so I simply put it inside the if statement checking if the wall removal was valid.

Building the adjacency list I got first try, since the walls indices stated which cells it separated, 0 and 1 or 58 and 59, I had them point to each other in the correct index spot in the adjacency list, so index 58 would point to 59 and 59 to 58, or 1 to 0 and 0 to 1.

The breadth first took some time and analysis of the maze created to verify if it was properly working. The main issues I had coding were int eh while loop as I would get index out of bounds errors often and in my initial draft of the function, I had a counter variable which would be used to set the distance from 0 which was incorrect of me to do. Instead I updated distance by adding one from the previous nodes distance which then solved my error of seeing nodes with 80 distance from 0 even though it was only 12 spaces away in reality and only got that number as it was visited much later in the queue.

Depth first took me more time to code as my initial start was somewhat clouded as I attempted to make it far to similar to the breadth first but then I realized I can’t rely on indices as much and must be confident that the items are entered in order form greatest recent value to least recent value where they are recent in terms of the most recent or current node that it currently looking at, say it is at node 17 and next to it is 27, 18 and 7, it would prioritize 7, then 18 and finally 27 and visit them in that order, getting the code to do that was my largest hurdle. Solving that issue was only a matter of time as I remembered the min and max functions built into python and implemented that to append the maximum values first then append the minimum values into the stack.

Drawing the solution didn’t take too long, it took more trial and error than anything to find how to properly align the lines in the center of each cell and follow the path back to the zero cell correctly and accurately. In my test runs I often ran into overlapping lines which I noticed from seeing missing lines and thicker lines than normal, to solve this I created 4 cases for the lines to follow, a line to the left, to the right a line up and a line down, these lines are made depending which node is the previous node. If the node is directly below the current then it goes down, if it’s above it went up and so on, the values for drawing such lines were found through pure trial and error no hard math, just staring at the image and trying something new until it looked correct and tried to perfect it from there, eventually reaching the function I have now.

Above are the runtimes for breadth first and depth first algorithms taken from various sized mazes, the even number runs are depth first and odd number runs are breadth first. Runs 1 and 2 were done on a 5x5 maze, 3 and 4 on a 10x10 maze, 45 and 6 on a 15x15 and 7 and 8th runs on a 20x20 maze to adequately increase the size of each runtime. In the long run it can be see that breadth first is faster on average while depth first is only slightly slower on very larger sizes but on smaller sizes it’s hardly noticeable.

This lab taught me better manipulation of stacks and queues as well as ways to interpret graphs from more abstract forms like a maze and create adequate solutions to solve and create adjacency lists from them.

Abstract ---------

# -\*- coding: utf-8 -\*-

"""

Course 2302(Data Structures)

Instructor: Olac Fuentes

Teaching Assistant: Anithdita Nath, Mali Zargaran

Lab 5

Last Edited on 4/30/2019

@author: Seth

"""

import matplotlib.pyplot as plt

import numpy as np

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\_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 draw\_maze\_solved(walls,maze\_rows,maze\_cols,D):

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')

cur = (maze\_rows\*maze\_cols)-1

# print(cur)

while (D[cur])[0] != -1:

# print((D[cur])[0])

if cur-(D[cur])[0] == maze\_cols: #vertical line

# print('making vertical')

x0 = (cur%maze\_cols)+.5

x1 = x0

y0 = (cur//maze\_cols)-.5

y1 = y0+1

elif (D[cur])[0]-1 == cur:#horizontal line to the right

# print('line to the right')

x0 = ((D[cur])[0]%maze\_cols)-.5

x1 = x0+1

y0 = (cur//maze\_cols)+.5

y1 = y0

elif (D[cur])[0]-maze\_cols == cur:#vertical upwards

# print('line upwards')

x0 = ((D[cur])[0]%maze\_cols)+.5

x1 = x0

y0 = (cur//maze\_cols)+.5

y1 = y0+1

else:#regular horizontal

# print('making horizontal')

x0 = ((D[cur])[0]%maze\_cols)+.5

x1 = x0+1

y0 = (cur//maze\_cols)+.5

y1 = y0

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

# print(cur)

cur = D[cur][0]

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 build\_AL(AL, i, walls):

#adds nodes walls[i][0] and walls[i][1] to AL and returns the new AL

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

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

return AL

def breadth\_first(AL):

#array containing distance is formatted by prev. node and then distance from 0 node

Q = []

Q.append(0)#forsure starting point

current = 0#Our first value in the q

Depth = [ [] for i in range(len(AL))]

Depth[current].append(-1)#no previous node

Depth[current].append(0)#distance of 0

while len(Q) != 0:

for i in range(len(AL[current])):

if len(Depth[(AL[current])[i]]) == 0:#if the next values have yet to be visited then add to Q

Q.append((AL[current])[i])

Depth[(AL[current])[i]].append(current)#lists prev node

Depth[(AL[current])[i]].append((Depth[current])[-1]+1)#distance is 1 more than the lasts

current = Q.pop()

#update Q, current value, and distance value

return Depth

def depth\_first(AL):

#array containing distance is formatted by prev. node and then distance from 0 node

S = []

S.append(0)#forsure starting point

current = 0#Our first value in the q

Depth = [ [] for i in range(len(AL))]

Depth[current].append(-1)#no previous node

Depth[current].append(0)#distance of 0

while len(S) != 0:

for i in range(len(AL[current])):

if len(Depth[max(AL[current])]) == 0:#if the next values have yet to be visited then add to S

# print(AL[current])

S.append(max(AL[current]))

Depth[max(AL[current])].append(current)#lists prev node

Depth[max(AL[current])].append((Depth[current])[-1]+1)#distance is 1 more than the lasts

AL[current].remove(max(AL[current]))

elif len(Depth[min(AL[current])]) == 0:

S.append(min(AL[current]))

Depth[min(AL[current])].append(current)#lists prev node

Depth[min(AL[current])].append((Depth[current])[-1]+1)#distance is 1 more than the lasts

AL[current].remove(min(AL[current]))

# print(S)

current = S.pop()

#update Q, current value, and distance value

return Depth

#creates value for the 0 node, the starting node being 0 at distance 0

plt.close("all")

maze\_rows = 20

maze\_cols = 20

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

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

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

#constructs maze via union by size and compression

r = maze\_rows\*maze\_cols

print('The maze has', maze\_rows\*maze\_cols, 'cells, how many walls would you like to remove?')

r = input()

remove = int(r)

#printing output based upon user input

if remove > maze\_rows\*maze\_cols:

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

elif remove < maze\_rows\*maze\_cols-1:

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

else:

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

AL = [ [] for i in range(maze\_rows\*maze\_cols) ] #creates an AL that has the num cells in the maze, each cell will be a node

while remove > 0: #Remove walls until there is no more walls to remove

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

if len(dsfToSetList(S)) == 1:#once there is a complete maze, start removing walls at random

AL = build\_AL(AL, d, walls)

walls.pop(d)

remove-=1

elif find\_c(S, (walls[d])[0]) != find\_c(S, (walls[d])[1]):#if points are already in same set do not remove

union\_by\_size(S, (walls[d])[0], (walls[d])[1])

AL = build\_AL(AL, d, walls)

walls.pop(d)

remove -=1

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

print(AL)

print('##########################################################')

#print(walls)

start = time.time()

D = breadth\_first(AL)

end = time.time()

print('Time for Breadth:', end - start)

S = [[] for i in range(len(AL))]

start = time.time()

S = depth\_first(AL)

end = time.time()

print('Time for depth:', end - start)

#print('Using depth first')

#print(S)

print('####################################################################')

#print('using breadth first')

#print(D)

draw\_maze\_solved(walls,maze\_rows,maze\_cols, D) #post\_maze

“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.”

- Seth Abel Flores