Michael Gonzalez

CS 2302 Data Structures

Olac Fuentes

05/01/2019

Lab 7 – Graphs

For lab 6, I wrote a program that created a maze where each cell was reachable from any other cell and there was a unique path from the start to the destination. This time we have to modify this code and be able to do the following. 1. Modify the maze-building program to ask the user how many walls to remove from the maze and to handle 3 cases where (a) A path from source to destination is not guaranteed to exist (when m < n -1), (b) The is a unique path from source to destination (when m = n-1), or (c) There is at least one path from source to destination (when m > n -1). 2. Write a method to build the adjacency list representation of your maze. 3. Implement the following algorithms to solve the maze you created, assuming the starting position is bottom-left corner and the goal position is the top-right corner. (a) Breadth-first search. (b) Depth-first search using a stack. This is identical to breadth-first search. but the queue is replaced by a stack. (c) Depth-first using recursion.

My solution for part 1 is simple, The changes I made to allow for user interaction are as follow. Print to the console “"n, the number of cells:",num\_cells” &"Type m, the number of walls to remove:”” and had them type whatever integer they wished and depending on their choice it would take them to an if statement. If m < n-1 it printed "A path from source to destination is not guaranteed to exist (m < n -1)" and ran the method Choice1, or if m > n-1 it printed “here is at least one path from source to destination (m > n-1)" and ran the method Choice2, lastly if it wasn’t choice 1 or 2 it meant that m = n-1 and ran the method Choice 3.

The methods Choice1, Choice2, and Choice3 are essentially the same method and are only name differently for documentation purposes. Each method takes in a disjoint set forest S, a “wall” array and a “m” variable which is the chosen number of walls to delete or num\_cells for choice 3. First the method will create an empty list G, this will hold our Graphs adjacency list, then the method runs a while loop with the condition, while m > 0, this while loop is used to create the maze by removing a random wall m times or in choice 3 where it keeps removing walls until all cells are in 1 set. In the loop it creates and stores a random integer d, and goes into an if statement where it makes the comparison, dsf.find(S,walls[d][0]) != dsf.find(S,walls[d][1]), this uses a method from the disjoint set forest file to find the parent node of the specified wall, this is done just like in Lab 6-Maze. If the condition returns true then it unites the two walls and pops the walls from the list walls, then it takes that popped wall and adds it to the G list. Since a popped wall means there is an edge connecting point A to B we add both points to it. Next, out the if statement it decreases m by one and the while loop runs until there is no more walls to remove. This may or may not create a solvable maze for if the 2 end points are not in a single set there will be no path to take. Once out the while loop it draws the updated DSF and creates the Maze with the draw\_maze function provided in class. Next it goes trough the 3 search methods and plots the solution for the maze.

For Breadth\_First\_search, I followed the pseudocode given in class and came to this method. This method takes in a graph G represented by an adjacency list and v an integer of where to start. In the method it creates two arrays, one called visited filled with false boolean values of size G and prev filled with -1 integers of size G. then it creates an empty Queue and enqueues (v) makes visited[v] true and goes into a while loop. The While loop has the condition not Q.empty, which means to keep going as long as the queue is not empty. In the while loop the code dequeues an element and stores it in u, next it goes into a for loop, that goes for t in G[u] which is every vertex that is connected to the current vertex. It changes that vertex’s visited value to true and updates prev as well, prev[t] = u and enqueues the current vertex in G[u]. Once the queue is empty it returns prev an in it the connection to that vertex’s previous vertex. The running time for this method is usually around

For Depth\_First\_search, I followed the pseudocode given in class and came to this method. This method takes in a graph G represented by an adjacency list and v an integer of where to start. In the method it creates two arrays, one called visited filled with false boolean values of size G and prev filled with -1 integers of size G. then it creates an empty Stack and Stacks (v) makes visited[v] true and goes into a while loop. The While loop has the condition not len(stack)<=0, which means to keep going as long as the stack is not empty. In the while loop the code pops an element and stores it in u, next it goes into a for loop, that goes for t in G[u] which is every vertex that is connected to the current vertex. It changes that vertex’s visited value to true and updates prev as well, prev[t] = u, and stacks the current vertex in G[u]. Once the stack is empty it returns prev an in it the connection to that vertex’s previous vertex. The running time for this method is usually around

For Depth\_First\_search\_Recursion, I followed the pseudocode given in class and came to this method. This method takes in a graph G represented by an adjacency list and v an integer of where to start. Outside the method there are two arrays, one called visited filled with false boolean values of size G and prev filled with -1 integers of size G we will be updating these arrays trough the method using pass by reference. then it makes visited[v] true and goes into a for loop. The for loop has the condition that goes for t in G[u] which is every vertex that is connected to the current vertex. Then it goes into an if statement, if visited[t] has not been visited then it changes it to true, updates prev, prev[t] = u and calls Depth\_First\_search\_Recursion(G,t). At this point the code does a recursive call to keep traversing trough the vertexs and keeps going until all calls have been complete and once, they are the ending prev and visited arrays are now what they should be. The running time for this method is usually around

Conclusion

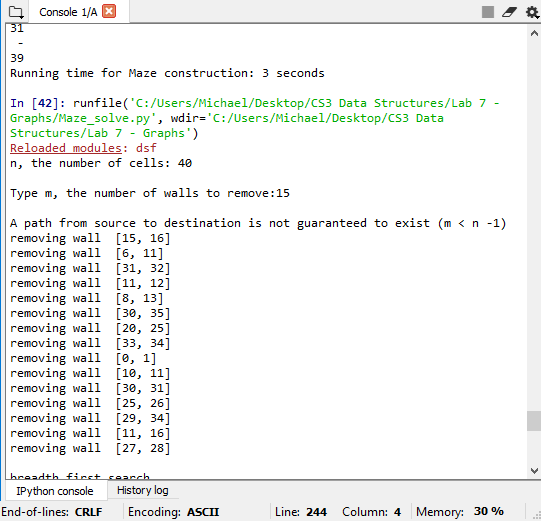
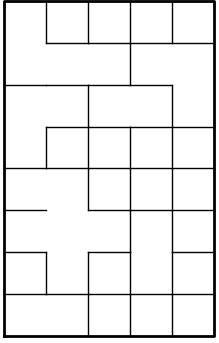
In conclusion, I learned how to design a maze and grow to solve it using different algorithms, and how to calculate its running time. I have now a better appreciation for the application of graphs and how they can be used to represent data. I have become more comfortable with coding in python than in lab 6 and I believe that I will be able to learn more from the last lab and hopefully make the grade in this class.

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.

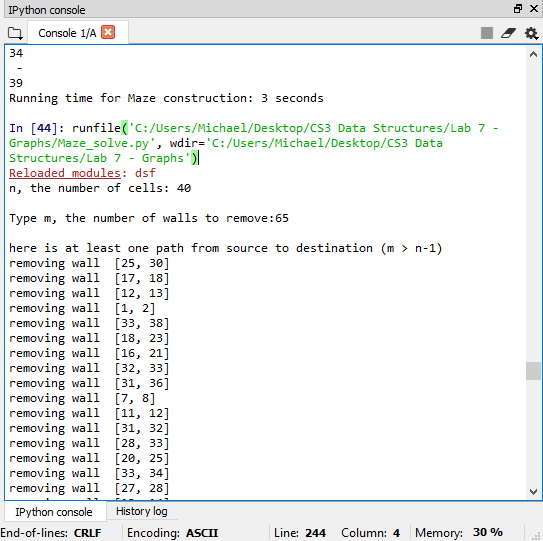
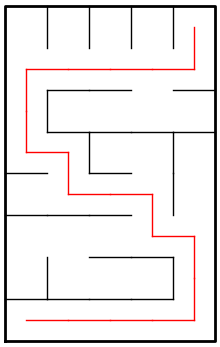
– Michael Gonzalez

Experiments/Results

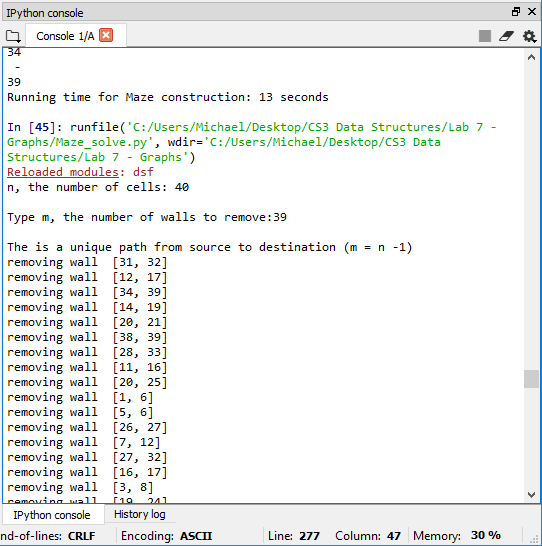
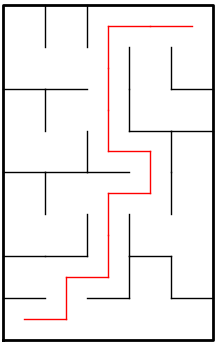
1a. (when m < n -1): EX: 8\*5 m=15 EXAMPLE MAZE:



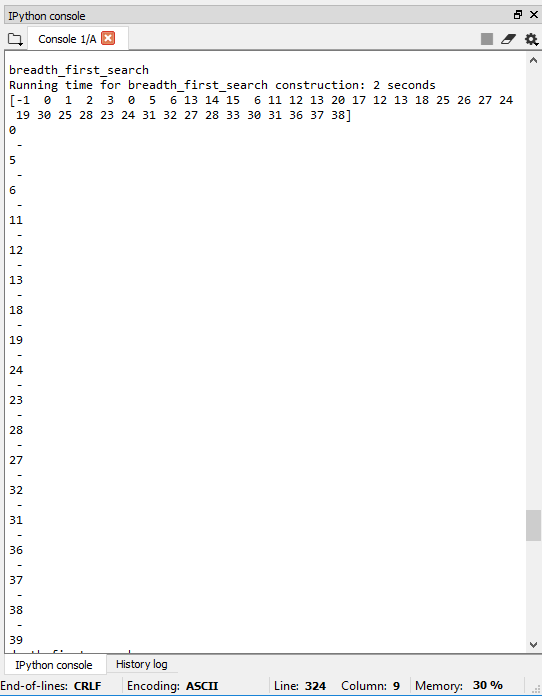
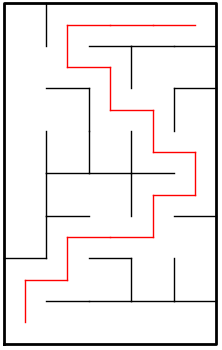
1b. (when m > n -1): EX: 8\*5 m=50 EXAMPLE MAZE:



1c. (when m = n-1) : EX: 8\*5 m=39 EXAMPLE MAZE:

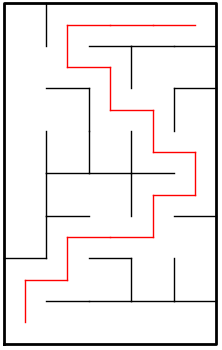


3a. Breadth\_First\_Search:



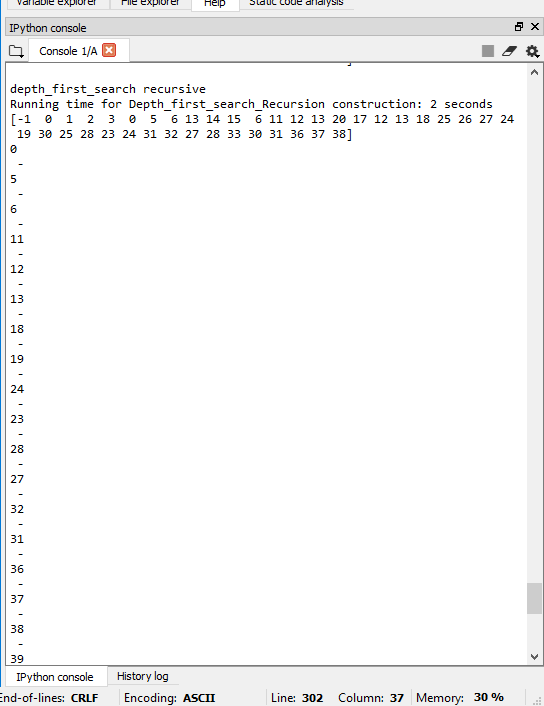
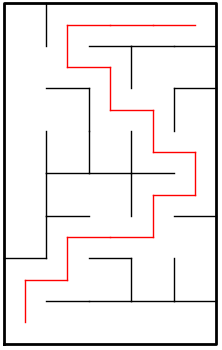
Running time is: 2 Seconds

3b. Depth\_first\_search:



Running time is: 2 Seconds

3c. Depth\_First\_Search\_Recursion:



Running time is: 2 Seconds

Appendix – code

#Author: Michael Gonzalez

#Course: CS 2302 Data Structures

#Lab 7

#TA: Anindita Nath & Eduuardo Lara

#Purpose:the purpose of this lab is to modify our maze code to handle 3 cases

#1.

#(a) A path from source to destination is not guaranteed to exist (when m < n -1)

#(b) The is a unique path from source to destination (when m = n-1)

#(c) There is at least one path from source to destination (when m > n -1).

#2.Write a method to build the adjacency list representation of your maze.

#3.Implement the following algorithms to solve the maze you created, assuming the starting position is

#bottom-left corner and the goal position is the top-right corner.

#(a) Breadth-first search.

#(b) Depth-first search using a stack. This is identical to breadth-first search. but the queue is replaced

#by a stack.

#(c) Depth-first using recursion.

# Last modified April,28 2019

import matplotlib.pyplot as plt

import numpy as np

import random

import time

import queue

import dsf

def adj\_list\_to\_edge\_list(G):

g = []

for source in range(len(G)):

g.append([source,G[source]])

return g

def remove\_all\_v\_edges(G,v):

for i in range(len(G[v])):

G[v][i] = -1

return G

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)

return ax

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 CountSets(S):

num\_sets = 0

for i in range(len(S)):

if S[i]<0:

num\_sets += 1

dsf.find\_c(S,i)

return num\_sets

###################################################

# LAB START

#created a method to draw a series off lines if there is a solution

def draw\_path(ax,prev,v,xe,ye):

if prev[v] != -1:

if v == prev[v]+maze\_cols:

xi= xe

yi=ye-1

draw\_path(ax,prev,prev[v],xi,yi)

ax.plot([xi,xe],[yi,ye],linewidth=1,color='r')

if v == prev[v]-maze\_cols:

xi= xe

yi=ye+1

draw\_path(ax,prev,prev[v],xi,yi)

ax.plot([xi,xe],[yi,ye],linewidth=1,color='r')

if v == prev[v]+1:

xi= xe-1

yi=ye

draw\_path(ax,prev,prev[v],xi,yi)

ax.plot([xi,xe],[yi,ye],linewidth=1,color='r')

if v == prev[v]-1:

xi= xe+1

yi=ye

draw\_path(ax,prev,prev[v],xi,yi)

ax.plot([xi,xe],[yi,ye],linewidth=1,color='r')

# method for if the user choses m < n-1

def Choice\_1(S,walls,m):

G = []

for i in range(maze\_cols\*maze\_rows):

G.append([])

while m > 0:

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

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

if dsf.find(S,walls[d][0]) != dsf.find(S,walls[d][1]):

# dsf.union(S,walls[d][0],walls[d][1])

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

poppedWall = walls.pop(d)

G[poppedWall[0]].append(poppedWall[1])

G[poppedWall[1]].append(poppedWall[0])

m = m-1

dsf.draw\_dsf(S)

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

print()

print("breadth\_first\_search")

Path = breadth\_first\_search(G,0)

draw\_path(pic,Path,num\_cells-1,maze\_cols-.5,maze\_rows-.5)

print(Path)

printPath(Path,num\_cells-1)

return G

# method for if the user choses m > n-1

def Choice\_2(S,walls,m):

G = []

for i in range(maze\_cols\*maze\_rows):

G.append([])

while m > 0:

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

if dsf.find(S,walls[d][0]) != dsf.find(S,walls[d][1]):

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

# dsf.union(S,walls[d][0],walls[d][1])

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

poppedWall = walls.pop(d)

G[poppedWall[0]].append(poppedWall[1])

G[poppedWall[1]].append(poppedWall[0])

m = m-1

dsf.draw\_dsf(S)

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

print()

print("breadth\_first\_search")

Path = breadth\_first\_search(G,0)

draw\_path(pic,Path,num\_cells-1,maze\_cols-.5,maze\_rows-.5)

print(Path)

printPath(Path,num\_cells-1)

return G

# method for if the user choses m = n-1

def Choice\_3(S,walls,num\_cells):

G = []

for i in range(num\_cells):

G.append([])

while CountSets(S) > 1:

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

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

if dsf.find(S,walls[d][0]) != dsf.find(S,walls[d][1]):

# dsf.union(S,walls[d][0],walls[d][1])

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

poppedWall = walls.pop(d)

G[poppedWall[0]].append(poppedWall[1])

G[poppedWall[1]].append(poppedWall[0])

dsf.draw\_dsf(S)

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

print()

print("breadth\_first\_search")

Path = breadth\_first\_search(G,0)

draw\_path(pic,Path,num\_cells-1,maze\_cols-.5,maze\_rows-.5)

print(Path)

printPath(Path,num\_cells-1)

return G

#method used to print path as an array

def printPath(prev,v):

if prev[v] != -1:

printPath(prev,prev[v])

print(" - ")

print(v)

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

visited =np.zeros(len(G),dtype=bool)

prev = np.zeros(len(G),dtype=int)-1

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

def Depth\_first\_search\_Stack(G,v):

visited =np.zeros(len(G),dtype=bool)

prev = np.zeros(len(G),dtype=int)-1

stack = []

stack.append(v)

visited[v] = True

while not len(stack)<=0:

u = stack.pop()

for t in G[u]:

if not visited[t]:

visited[t] = True

prev[t] = u

stack.append(t)

return prev

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

visited[source] = True

for t in G[source]:

if not visited[t]:

prev[t] = source

depth\_first\_search\_Rec(G,t)

if \_\_name\_\_ == "\_\_main\_\_":

#########################################

# User Interface #

#########################################

start = time.time()

maze\_rows = 5

maze\_cols = 5

num\_cells = maze\_rows\*maze\_cols

plt.close("all")

#Maze creation

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

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

S = dsf.DisjointSetForest(num\_cells)

dsf.draw\_dsf(S)

print("n, the number of cells:",num\_cells)

x =input("Type m, the number of walls to remove:")

print()

#Choice m < n-1

if int(x) < num\_cells-1:

print("A path from source to destination is not guaranteed to exist (m < n -1)")

Gr = Choice\_1(S,walls,int(x))

print("depth\_first\_search Stack")

path3 = Depth\_first\_search\_Stack(Gr,0)

printPath(path3,num\_cells-1)

print(path3)

print()

visited =np.zeros(len(Gr),dtype=bool)

prev = np.zeros(len(Gr),dtype=int)-1

print("depth\_first\_search recursive")

depth\_first\_search\_Rec(Gr,0)

print(prev)

printPath(prev,num\_cells-1)

#Choice m > n-1

elif int(x) > num\_cells-1:

print("here is at least one path from source to destination (m > n-1)")

Gr = Choice\_2(S,walls,int(x))

print("depth\_first\_search Stack")

path3 = Depth\_first\_search\_Stack(Gr,0)

print(path3)

printPath(path3,num\_cells-1)

print()

visited =np.zeros(len(Gr),dtype=bool)

prev = np.zeros(len(Gr),dtype=int)-1

print("depth\_first\_search recursive")

depth\_first\_search\_Rec(Gr,0)

print(prev)

printPath(prev,num\_cells-1)

#Choice m = n-1

else:

print("The is a unique path from source to destination (m = n -1)")

Gr = Choice\_3(S,walls,num\_cells)

print("depth\_first\_search Stack")

path3 = Depth\_first\_search\_Stack(Gr,0)

printPath(path3,num\_cells-1)

print(path3)

print()

visited =np.zeros(len(Gr),dtype=bool)

prev = np.zeros(len(Gr),dtype=int)-1

print("depth\_first\_search recursive")

depth\_first\_search\_Rec(Gr,0)

print(prev)

printPath(prev,num\_cells-1)

elapsed\_time = time.time()-start

print("Running time for Maze construction:", round(elapsed\_time),"seconds")