Jon Munoz

Lab 7

CS2302

Professor: Olac Fuentes

TA: Anindita Nath

**INTRODUCTION**

For this lab we had to make an adjacency list of the maze that we had created for lab 6. Once we made this adjacency list we had to create the graph representation and implement breadth first search and depth first search in order to solve our maze. With the depth first search we had to do it two different ways, one with recursion and the other with stacks. Once we had the solution we would then print the path that was the solution.

**PROPOSED SOLUTION DESIGN AND IMPLEMENTATION**

For making the adjacency list had had originally struggled to figure out how to do this. I first tried to treat the walls list like a 2D array and would traverse using two for loops. When I tried this however I was not getting the desired results. I would get errors and I found this to be rather difficult. I soon saw a pattern in the walls list that allowed me to just traverse the list in its entirety with one for loop. What I did within this method was create an empty list and then append empty list to it for each cell in the maze. Once I had the list of list what I did was create a for loop to go to the length of the walls list before any modification was done to it and check each index i to see if the index at i for the original walls list was in the modified walls list after the maze had been made. If an index was not in the modified walls but was in the original walls then I appended ﻿AL[originalWall[i][0]].append(originalWall[i][1]) and ﻿AL[originalWall[i][1]].append(originalWall[i][0]). This would make each index in AJ, the adjacency list, have its corresponding adjacent cells. Doing this would also add the duplicate edges as well which is necessary for the BFS and DFS methods. Once I made this method I moved on to the DFS and BFS methods which was simply translating the pseudocode that we were given to actual code. For the part where we had to make the graph representation all that I did was simply modify the code that was given to us to draw a circular graph. For the part that actually printed the path in the maze I could not get it. I tried to store the values of the printPath method into a list and print those lines only but I could not figure it out. I tried if pathList[i] was in G (G being the adjacency list representation) but this would not work since it is a list of list.

**EXPERIMENTAL RESULTS**

A close up of a black background

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

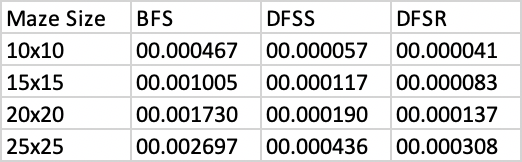
Description automatically generatedBelow is the results of running my code with a 5 by 5 maze with first having ten walls removed, the second has twenty four walls removed, and the last has thirty walls removed.

A picture containing black

Description automatically generatedA screenshot of a cell phone

Description automatically generated

When ten walls are removed there is no path, so the program just prints the destination. Below are the running times for the algorithms to find a path to the end. BFS is depth first search, DFSS is depth first search with stacks, and DFSR is depth first search with recursion. The times are in seconds/milliseconds:



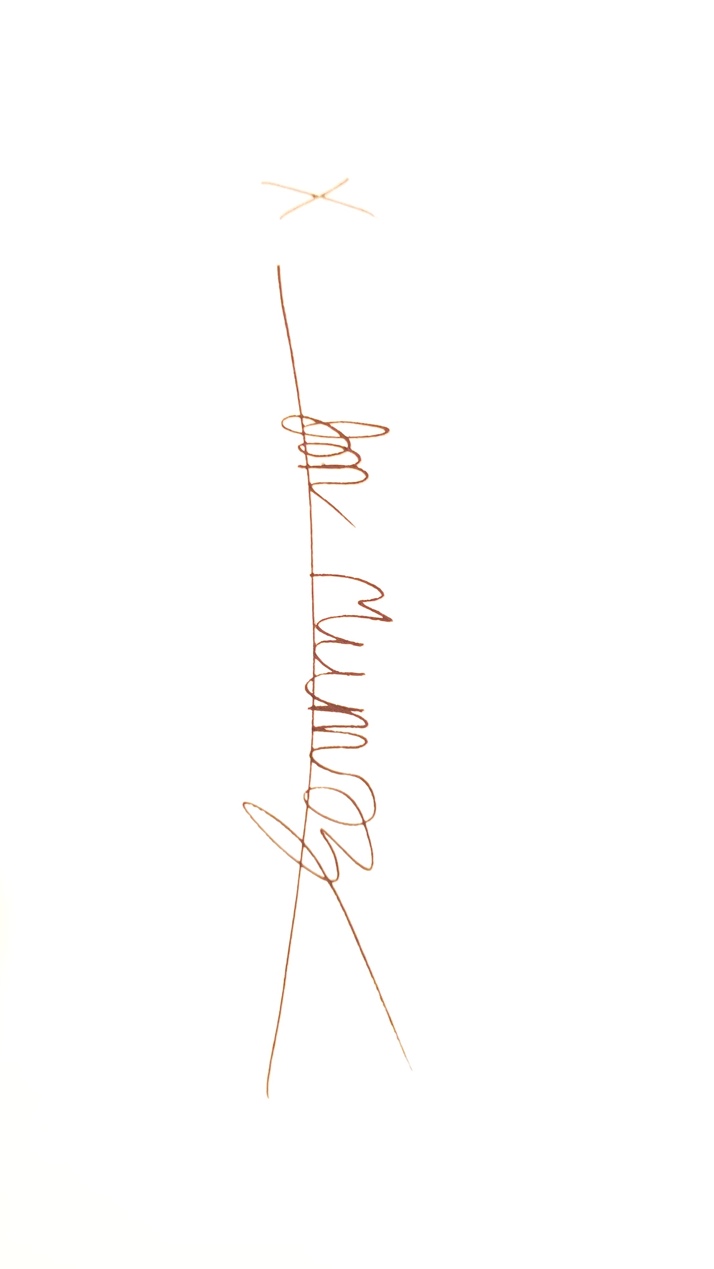
**CONCLUSION**

At the end of this lab I learned about how to make an adjacency list and how to work with BFS and DFS. One important aspect about DFS an BFS that I learned is that duplicate edges but be in the adjacency list in order for them to work, I did not know this originally but after the lab I now know.

**APPENDIX**

**SOURCE CODE**

|  |
| --- |
| #Jon Munoz |
|  | #CS2302 Data Structures |
|  | #Lab 7 |
|  | #Instructor:Olac Fuentes |
|  | #TA:Anindita Nath |
|  | #Last Modified 4/29/19 |
|  |  |
|  | import queue |
|  | import matplotlib.pyplot as plt |
|  | import numpy as np |
|  | import random |
|  | import datetime |
|  |  |
|  |  |
|  | #this method is the method that prints the path to a desired cell |
|  | def printPath(prev, v): |
|  | if prev[v] != -1:#keep going untill prev[v] is -1 since that will be the beginning |
|  | printPath(prev, prev[v])#make recursive call |
|  | print(" --- ", end = ' ')#print the path |
|  | print(v, end = ' ')#print the beginning cell |
|  |  |
|  | #BFS method |
|  | def breadth\_first\_search(G, v): |
|  | visited = [False for i in range(len(G))]#make a list filled with False |
|  | prev = [-1 for i in range(len(G))]#make a list filled with -1 |
|  | Q = queue.Queue()#create an empty queue |
|  | Q.put(v)#add v into the queue |
|  | visited[v] = True#make the visited index at v True |
|  | while Q.empty() == False:#while loop will keep going until the queue is not empty |
|  | u = Q.get()#extract the first item in the queue |
|  | for t in G[u]:#go through G |
|  | if visited[t] == False:#if visited at t is False then make it true and change prev at t to u |
|  | visited[t] = True |
|  | prev[t] = u |
|  | Q.put(t)#put t into the queue |
|  | return prev#return the prev list |
|  |  |
|  | #DFS using stack method |
|  | def DFSStacks(G, v): |
|  | visited = [False for i in range(len(G))]#make a list filled with False |
|  | prev = [-1 for i in range(len(G))]#make a list filled with -1 |
|  | S = []#create an empty list/stack |
|  | S.append(v)#add v to the stack |
|  | visited[v] = True#set visited at v to True |
|  | while len(S) != 0:#keep popping until S is empty |
|  | u = S.pop() |
|  | for t in G[u]: |
|  | if visited[t] == False:#if visited at t is False then make it true and change prev at t to u |
|  | visited[t] = True |
|  | prev[t] = u |
|  | S.append(t)#add t |
|  | return prev#return prev list |
|  |  |
|  | #DFS recursive method |
|  | def DFSRecursive(G, source): |
|  | global visited#make visited global so its not constantly reset |
|  | global prev#make prev global so its not constantly reset |
|  | visited[source] = True#set visited at source to True |
|  | for t in G[source]:#go through G |
|  | if visited[t] == False:#if visited at t is False then make it true |
|  | prev[t] = source#make prev at t equal to source |
|  | DFSRecursive(G, t)#recursive call |
|  | return prev#return prev list |
|  |  |
|  | 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") |
|  | #this was my attempt to print the path |
|  | # for i in range(n): |
|  | # for r in range(maze\_rows): |
|  | # for c in range(maze\_cols): |
|  | # cell = c + r\*maze\_cols |
|  | # coords.append([c+.5,r+.5]) |
|  | # for i in range(n): |
|  | # for j in range(len(L)): |
|  | # if L[i] in G[j]: |
|  | # for dest in G[i]: |
|  | # print("HERE") |
|  | # ax.plot([coords[i][0],coords[dest][0]],[coords[i][1],coords[dest][1]], |
|  | # linewidth=1,color='red') |
|  | 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 DisjointSetForest(size): |
|  | return np.zeros(size,dtype=np.int)-1 |
|  |  |
|  | #finds the root of a given number i |
|  | def find(S,i): |
|  | if S[i]<0:#if S[i] < zero then youre at the root return i |
|  | return i |
|  | return find(S,S[i])#recursive call if S[i] is not less that 0 |
|  |  |
|  | #finds the root of a given number i using path compression |
|  | def findC(S,i): |
|  | if S[i] < 0:#if S[i] < zero then youre at the root return i |
|  | return i |
|  | root = findC(S,S[i])#recursively get to the root |
|  | S[i] = root#directly set the root of i to the root value |
|  | return root |
|  |  |
|  | #combines the two given values using their size to choose who get combined |
|  | def unionbySize(S,i,j): |
|  | ri = findC(S,i)#find the root of the first value |
|  | rj = findC(S,j)#find the root of the second value |
|  | if ri!=rj:#if the roots are not the same then you complete the union |
|  | if S[ri] > S[rj]:#if S[ri] is greater than S[rj] then you combine S[rj] to S[ri] |
|  | S[rj] += S[ri] |
|  | S[ri] = rj |
|  | else:#rif S[rj] is greater than S[ri] then you combine S[ri] to S[rj] |
|  | S[ri]+=S[rj] |
|  | S[rj] = ri |
|  |  |
|  | #counts the number of sets in the given disjoint set forest |
|  | def NumSets(S): |
|  | count = 0#counter |
|  | for i in range(len(S)):#traverse the entire DSF |
|  | if S[i] < 0:#if an value is less than 0 then add one to the counter |
|  | count += 1 |
|  | return count |
|  |  |
|  | #union the two given values |
|  | 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: # Do nothing if i and j belong to the same set |
|  | S[rj] = ri |
|  |  |
|  | def draw\_graph(G,maze\_rows,maze\_cols): |
|  | fig, ax = plt.subplots() |
|  | n = len(G) |
|  | r = 30 |
|  | coords =[] |
|  | for i in range(n): |
|  | 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",bbox=dict(facecolor='w',boxstyle="circle")) |
|  | coords.append([c+.5,r+.5]) |
|  | for i in range(n): |
|  | for dest in G[i]: |
|  | ax.plot([coords[i][0],coords[dest][0]],[coords[i][1],coords[dest][1]], |
|  | linewidth=1,color='k') |
|  |  |
|  | #THis method converts the walls list to an Adjacency Listin order to do graph functions |
|  | def ToaJ(walls, originalWall, numCells): |
|  | AL = []#Adjacency List |
|  | for k in range(numCells):#fill the list with empty cells |
|  | AL.append([]) |
|  | for i in range(len(originalWall)):#go through the entire original walls list |
|  | if originalWall[i] not in walls:#if a wall is in the original walls list but not the list once the graph is made then the cells are connected |
|  | AL[originalWall[i][0]].append(originalWall[i][1])#add to adjacency list |
|  | AL[originalWall[i][1]].append(originalWall[i][0])#add to adjacency list |
|  | return AL#return the adjacency list |
|  |  |
|  | plt.close("all") |
|  | ans1 = input("How many rows would you like? ")#this line gets the user input for the desired number of rows |
|  | print("Number of rows:", ans1) |
|  | rows = int(ans1)#parse the input to an int |
|  | ans2 = input("How many columns would you like? ")#this line gets the user input for the desired number of columns |
|  | print("Number of columns:", ans2) |
|  | columns = int(ans2)#parse the input to an int |
|  |  |
|  | maze\_rows = rows#set maze\_rows to the value of rows |
|  | maze\_cols = columns#set maze\_cols to the value of columns |
|  |  |
|  | n = rows \* columns#used to tell how many cells there are |
|  |  |
|  | print() |
|  | print("There are ", n, "cells in the maze") |
|  | m = input("How many walls do you want to remove?") |
|  | m = int(m) |
|  |  |
|  | original = wall\_list(maze\_rows,maze\_cols)#store the original walls list into a variable for comparisons later |
|  | walls = wall\_list(maze\_rows,maze\_cols) |
|  |  |
|  | draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True) |
|  |  |
|  | S = DisjointSetForest(maze\_rows \* maze\_cols)#create a DSF with the size of the value of rows times the value of columns |
|  |  |
|  | if m < n - 1: |
|  | print("A path from source to destination is not guaranteed to exist") |
|  | while m != 0:#keep doing the below code until there is only one set |
|  | w = random.choice(walls)#set w to a random list within the bigger list |
|  | ind = walls.index(w)#get the index of the chosen list w |
|  | if findC(S,w[0]) != findC(S,w[1]):#find the parents of the number in index 0 and 1 of the list using path compression and if they are different union them |
|  | walls.pop(ind)#pop the wall connecting the two numbers since we want to "union" them |
|  | unionbySize(S,w[0],w[1])#use unionbySize method to connect the two values within the DSF |
|  | m -= 1#one set is removed so subtract 1 from the numberOfSets value |
|  | draw\_maze(walls,maze\_rows,maze\_cols)#draw the new maze |
|  |  |
|  | if m == n - 1: |
|  | print("The is a unique path from source to destination") |
|  | while m != 0:#keep doing the below code until there is only one set |
|  | w = random.choice(walls)#set w to a random list within the bigger list |
|  | ind = walls.index(w)#get the index of the chosen list w |
|  | if findC(S,w[0]) != findC(S,w[1]):#find the parents of the number in index 0 and 1 of the list using path compression and if they are different union them |
|  | walls.pop(ind)#pop the wall connecting the two numbers since we want to "union" them |
|  | unionbySize(S,w[0],w[1])#use unionbySize method to connect the two values within the DSF |
|  | m -= 1#one set is removed so subtract 1 from the numberOfSets value |
|  | draw\_maze(walls,maze\_rows,maze\_cols)#draw the new mazee |
|  |  |
|  | if m > n - 1: |
|  | print("There is at least one path from source to destination") |
|  | numberOfSets = NumSets(S) |
|  | while m != 0:#keep doing the below code until there is only one set |
|  | w = random.choice(walls)#set w to a random list within the bigger list |
|  | ind = walls.index(w)#get the index of the chosen list w |
|  | if findC(S,w[0]) != findC(S,w[1]):#find the parents of the number in index 0 and 1 of the list using path compression and if they are different union them |
|  | walls.pop(ind)#pop the wall connecting the two numbers since we want to "union" them |
|  | unionbySize(S,w[0],w[1])#use unionbySize method to connect the two values within the DSF |
|  | numberOfSets -= 1 |
|  | m -= 1#one set is removed so subtract 1 from the numberOfSets value |
|  | if numberOfSets == 1 and m > 0: |
|  | walls.pop(ind) |
|  | m -= 1 |
|  | # draw\_maze(walls,maze\_rows,maze\_cols)#draw the new maze |
|  |  |
|  | numCells = maze\_rows \* maze\_cols#numCells variable helps when doing the printpath call |
|  |  |
|  |  |
|  | #below are the calls to the differant search methods and rhe paths that they produce |
|  | print("------------------------------------") |
|  | print("ADJACENCY LIST") |
|  | G = ToaJ(walls, original, numCells) |
|  | visited = [False for i in range(len(G))] |
|  | prev = [-1 for i in range(len(G))] |
|  | print(ToaJ(walls, original, numCells)) |
|  | print("BFS") |
|  | start = datetime.datetime.now()#start time of maze construction |
|  | K = breadth\_first\_search(G, 0) |
|  | end = datetime.datetime.now()#end time of creating the maze |
|  | elapsed = end - start#total time |
|  | print(elapsed) |
|  | print(K) |
|  | printPath(K, numCells - 1) |
|  | print() |
|  | print("------------------------------------") |
|  | print("DFS with Stack") |
|  | start = datetime.datetime.now()#start time of maze construction |
|  | J = DFSStacks(G,0) |
|  | end = datetime.datetime.now()#end time of creating the maze |
|  | elapsed = end - start#total time |
|  | print(elapsed) |
|  | print(J) |
|  | printPath(J, numCells - 1) |
|  | print() |
|  | print("------------------------------------") |
|  | print("DFS Recursive") |
|  | start = datetime.datetime.now()#start time of maze construction |
|  | H = DFSRecursive(G,0) |
|  | end = datetime.datetime.now()#end time of creating the maze |
|  | elapsed = end - start#total time |
|  | print(elapsed) |
|  | print(H) |
|  |  |
|  | printPath(H, numCells - 1) |
|  | draw\_graph(G,maze\_rows, maze\_cols) |



“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 provide inappropriate assistance to any student in the class.”