**Introduction**

The object of this assignment is to create a maze using disjoint set forest and give the path from left-bottom to right-top. In order to get the path, we are using breadth-first search, depth-first search using stack, and depth-first search using recursion. At first, we need to build the appropriate maze, when two cells don’t belong to the same set, it means there is no path from those two cells. Thus the implementation is to unionize when two cells are not in the same set and remove the wall. Walls are defined in an array and by removing an appropriate wall, the maze will be drawn with single a path. In addition, we are comparing standard union and union with compressed. At the beginning, the program asks whether the user wants to use standard union or union with compression. After having the maze, we run each ways to get the path for the user.

**Part-1:** display maze info depending on cell numbers and number of walls to be removed

First, let user know the number of cells and lets user input the number of walls to be removed. If the input number is less than (cells -1), display that destination is not guaranteed, if input number is same as (cells-1), display there is a unique path, otherwise, there is at least one path.

**Part-2:** Build adjacency list of the maze

The way I reached out for this part is to assume that the maze has all the edges and none of the edges are removed yet. Thus, g (variable for list) will have all the elements when created. In adjacency list, we don’t include the edge for the ones which wall exist. So, I have a for loop which traverse through each walls and remove edges. After everything is done, return the adjacency list.

**Part-3:** get the path

For the breadth-first search, at first I declare a boolean variable which will check if the the vertex is visited already and prev which contains information to get the path. For this breadth-first search, I used queue to implement it. As long as queue is not empty, dequeue element, u, and traverse the list from index u. If the element is not visited yet, set to visited and get the index to prev and enqueue the following vertex. After the execution, prev will have the path information.

**Part-3-b:** Depth-first search with stack

This part is very similar to breadth first search, but using stack instead of queue. Therefore, it will be reversed which means it will reach the deep and come back to the source.

**Part-3-c:** depth-first search with recursion

This part is similar to depth-first search but in recursive instead of using stack. Since we are using recursion, we need global variable for prev and visited variable. Every vertex visited will be checked as visited and if the next vertex is not checked, mark the index at next vertex to current vertex.

Output:

There are 25 cells.

How many walls would you like to remove? 10

Type 1 to use standard union, type 2 to use union with path compressions. 2

A path from source to destination is not guaranteed to exist.

Adjacency List: [[1], [0, 6], [3, 7], [2, 4, 8], [3], [10], [1, 7, 11], [2, 6, 12], [3, 9], [8], [5, 11, 15], [6, 10], [7, 13], [12, 14, 18], [13], [10], [17], [16, 18, 22], [13, 17, 19], [18], [21], [20, 22], [17, 21, 23], [22, 24], [23]]

Using BFS: 0-1-6-7-1

2-13-18-17-22-23-24

Runtime of BFS: 0.0 s.

Using DFS stack: 0-1-6-7-12-13-18-17-22-23-24

Runtime of DFS - stack ver.: 15.5896 s.

Using DFS recursion: 0-1-6-7-12-13-18-17-22-23-24

Runtime of DFS - recursion ver.: 15.6399 s.

A close up of a device

Description automatically generated

Output:

There are 49 cells.

How many walls would you like to remove? 48

Type 1 to use standard union, type 2 to use union with path compressions. 2

There is a unique path from source to destination.

Adjacency List: [[1, 7], [0, 2], [1, 3], [2, 4, 10], [3, 5], [4, 6], [5], [0, 8], [7], [10], [3, 9, 11, 17], [10, 12, 18], [11, 13], [12], [15], [14, 22], [17, 23], [10, 16, 24], [11, 19], [18], [27], [22, 28], [15, 21, 23], [16, 22, 30], [17, 25], [24, 26, 32], [25, 27, 33], [20, 26, 34], [21, 35], [36], [23, 31, 37], [30], [25], [26], [27], [28, 36], [29, 35, 43], [30, 38, 44], [37, 45], [46], [41, 47], [40, 48], [43], [36, 42], [37], [38, 46], [39, 45, 47], [40, 46], [41]]

Using BFS: 0-1-2-3-10-17-16-23-30-37-38-45-46-47-40-41-48

Runtime of BFS: 0.0 s.

Using DFS stack: 0-1-2-3-10-17-16-23-30-37-38-45-46-47-40-41-48

Runtime of DFS - stack ver.: 0.0 s.

Using DFS recursion: 0-1-2-3-10-17-16-23-30-37-38-45-46-47-40-41-48

Runtime of DFS - recursion ver.: 0.0 s.

A close up of a keyboard

Description automatically generated

"""""""""""""""""""""""""""""""""""""""

Lab 7 - Graphs

04/29/2019 - Ken M. Amamori

CS2302 MW 10:30 - 11:50

Professor: Olac Fuentes

TA: Anindita Nath, Maliheh Zargaran

IA: Eduardo Lara PL: Erick Macik

"""""""""""""""""""""""""""""""""""""""

import matplotlib.pyplot as plt

import numpy as np

import random

from scipy import interpolate

import time

import sys

import math

import queue

#print the path from top right corner to 0

def printPath(prev, v):

if prev[v]!=-1:

printPath(prev, prev[v])

print("-", end='')

print(v, end='')

#solving the maze using DFS rec

def DFS\_rec(g, s):

global visited\_rec #global var

global prev\_rec #global var

visited\_rec[s] = True

for t in g[s]:

if not visited\_rec[t]:

visited\_rec[t] = True

prev\_rec[t] = s

DFS\_rec(g, t)

#solving the maze using DFS stack

def DFS\_stack(g, v):

visited, prev = [False for b in range(len(g))], [-1 for a in range(len(g))] #visited var and previous state

s = []

s.append(v) #stack

visited[v] = True

while s!=[]: #as long as stack has something

u = s.pop() #pop the newest value

for t in g[u]:

if not visited[t]: #non checked

visited[t] = True

prev[t] = u

s.append(t) #push

printPath(prev, len(g)-1) #print the path

#breadth first search

def BFS(g, v):

visited, prev = [False for b in range(len(g))], [-1 for a in range(len(g))] #visited var and previous state

q = queue.Queue(10) #declare Queue

q.put(v) #enqueue

visited[v] = True

while not q.empty(): #has something

u = q.get() #dequeue

for t in g[u]:

if not visited[t]: #non checked

visited[t] = True

prev[t] = u

q.put(t) #enqueu

printPath(prev, len(g)-1) #print the path

#check if maze is unique or not

def checkValidRemove(r, c):

if r==c-1:

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

elif r<c-1:

print("A path from source to destination is not guarateed to exist.")

else:

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

#Find with path compression

def find\_c(S,i):

if S[i]<0:

return i

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

S[i] = r

return r

# Returns root of tree that i belongs to

def find(S,i):

if S[i]<0:

return i

return find(S,S[i])

# Joins i's tree and j's tree with compressed manner, if they are different

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

ri = find\_c(S,i)

rj = find\_c(S,j)

if ri!=rj: # Do nothing if i and j belong to the same set

S[rj] = ri

return True #return true since i and j joined

return False

# Joins i's tree and j's tree, if they are different

def union(S,i,j):

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 # Make j's root point to i's root

return True #return true since i and j joined

return False

#create a array(dsf) with give size

def DisjointSetForest(size):

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

#return the number of sets in S

def count\_sets(S):

c = 0

for i in S:

if i==-1:

c+=1

return c

#draw maze

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)

#create a list with

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

#create full Adjacency List

def createFullAL(row, col):

g = [[] for a in range(row\*col)] #row x col

for r in range(row):

for c in range(col):

cell = c + r\*col

if r!=0: #down

g[cell].append(cell-col)

if cell%col!=0: #left

g[cell].append(cell-1)

if c!=(col-1): #right

g[cell].append(cell+1)

if r!=row-1:

g[cell].append(cell+col)

return g

#create Adjacencylist

def createAL(r, c, w):

g = createFullAL(r, c)

for i in w:

g[i[0]].remove(i[1]) #remove

g[i[1]].remove(i[0]) #remove

return g

plt.close("all")

maze\_rows = 100

maze\_cols = 100

print("There are ", (maze\_rows\*maze\_cols), "cells.")

rev = int(input("How many walls whould you like to remove?\t"))

op = int(input("Type 1 to use standard union, type 2 to use union with path compressions.\t"))

walls = wall\_list(maze\_rows,maze\_cols) #create a maze separeated by walls

S = DisjointSetForest(maze\_rows\*maze\_cols) #create dsf to create maze

#part 1

checkValidRemove(rev, (maze\_cols\*maze\_rows))

while count\_sets(S)>1 : #while there are more than 1 set

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

if op==1: #standard union

if union(S, walls[d][0], walls[d][1]): #if two numbers were in different sets

walls.pop(d) #remove

elif op==2: #union with path compression

if union\_c(S, walls[d][0], walls[d][1]): #if two numbers were in different sets

walls.pop(d) #remove

else: #exit program

sys.exit("You chose the wrong input number!")

#part2

g = createAL(maze\_rows, maze\_cols, walls) #create Adjacency list with given maze

print("Adjacency List: ", g)

s0 = time.time()\*1000

#part3-a

print("\nUsing BFS: 0", end='')

BFS(g, 0) #breadth first search

s1 = time.time()\*1000

print("\nRuntime of BFS: ", round((s1-s0), 4), "s.")

#part3-b

print("\nUsing DFS stack: 0", end='')

DFS\_stack(g, 0) #depth first search - stack

s2 = time.time()\*1000

print("\nRuntime of DFS - stack ver.: ", round((s2-s1), 4), "s.")

#part3-c

visited\_rec, prev\_rec = [False for b in range(len(g))], [-1 for a in range(len(g))]

DFS\_rec(g, 0) #depth first search - recursion

print("\nUsing DFS recursiong: 0", end='')

printPath(prev\_rec, (maze\_cols\*maze\_rows)-1) #printing the path

s3 = time.time()\*1000

print("\nRuntime of DFS - recursion ver.: ", round((s3-s2), 4), "s.")

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

plt.show()

**Conclusion:**

Throughout this lab, I learned how to manage well algorithms to get the shortest path form source to destination, which are depth-first search and breadth first search. I found similarities between them especially depth-first search using stack and breadth first search. The only difference was that one uses stack and the other is queue. I found that these algorithm could be used to find the solution to the maze very easily then I expected. Also I realized how disjoint set forest are related to each other. Therefore in this lab, it was a great practice and good preparation for the exam.

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.