Yahir Rivas

Lab 6 Report

Instructor: Dr. Olac Fuentes

Teaching Assistant: Nath, Anindita

CS 2302

MW 1:30 - 2:50

**Lab 7 Report**

For this lab I was asked to modify the methods from lab 7 and implement several different methods. First, I had to ask the user the number of walls they wanted remove after giving them the number of cells in the maze. Depending on the answer from the user I would print whether if a path was possible with the number of walls removed. The next method was to build an adjacency list representation of the maze. Each cell being a vertex and an edge representing the path from one cell to another. The last part was implementing three methods to solve the puzzle. These methods being Breadth-first search and two different implementations of depth-first search. One recursive and the other using stacks instead of the queue used for the breadth-first search algorithm.

First, I tried implementing the adjacency list. I created a list with the size of the cells in the maze and appended every vertex to the list. Then if the user asked for more than 0 walls to be removed I checked if the vertices of the wall that I was trying to remove were not already part of the same set I popped the wall.

Then I implemented the other search methods starting with Breadth-First Search. For this method I took a graph (adjacency list) and the number of vertices. Then I created a list to store if a vertex was already visited and the previous vertices. Then I appended the vertex to the queue and if the Q was not empty then traverse the graph and if a vertex hasn’t been visited, then append it to the queue and assign a new value to visited vertex and previous vertices. Then I returned the list of previous vertices.

For Depth-First Search I used a stack instead of a queue. This method was very similar to the Breadth-First Search method. I created a visited list and a previous list. Then I appended the vertices to the stack and popped the first element. Then I traversed the whole list appending the previous index to the list and then returning the list of previous vertices.

For the recursive implementation of Depth-First Search I traversed the list with a for loop and if the index had not been visited I created a previous list with the element I’m currently on and go to the next element, creating a list of previous and then returning it.

Unfortunately, my program was unable to run since I had issues creating the adjacency list with the maze. However, I tried my best to implement the other methods even though I was unable to test them. In this lab I learned how to implement an adjacency list. I also learned how to use stacks and queues for the implementations of methods like Depth-First Search and Breadth-First Search.

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

"""

Created on Mon May 6 09:55:30 2019

@author: Yahir F. Rivas

"""

import matplotlib.pyplot as plt

import numpy as np

import random

import time

from collections import deque

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

def countSets(S):

c = 0

for i in S:

if i==-1:

c+=1

return c

def DisjointSetForest(size):

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

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):

# 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

return True

return False

################################################################### Compression Methods

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

return True

return False

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 unionSize(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: #if different root

if S[ri] > S[rj]: #if ri is bigger than rj then rj goes to ri

S[rj] += S[ri]

S[ri] = rj

return True

else:

S[ri] += S[rj] #if rj is bigger than ri then ri goes to rj

S[rj] = ri

return True

return False

def rWalls(maze, rw):

if maze < rw -1:

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

elif maze == rw -1:

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

elif maze > rw -1:

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

def adjacencyList(v, cells):

#creates list with size cells

G = [[] for i in range(cells)]

for i in v:

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

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

return G

def breadthFirst(G,v):

visited = []

visited = [False for i in range(len(G))]

prev = []

prev = [-1 for i in range(len(G))]

Q = deque([])

Q.append(v)

visited[v] = True

while Q:

#popleft removes first element in Q

u = Q.popleft()

for t in G[u]:

if not visited[t]:

visited[t] = True

prev[t] = u

Q.append(t)

#returns solution of this search

return prev

def depthFirst(G,source):

visited = [False for i in range(len(G))] #Create visited list

prev = [-1 for i in range(len(G))] #Previous element is set to -1

Stack = []

Stack.append(source)

visited[source] = True #mark as visited

while len(Stack) != 0: #while it is not empty

s = Stack.pop()

for t in G[s]:

if not visited[t]:

visited[t] = True #marks as visited

prev[t] = s #previous node is now the source

Stack.append(t)

return prev

def depthFirstR(G,source):

visited = False

prev = source

visited[source] = True

for t in G[source]: #checks every element in the graph

if not visited[t]: #if it hasn't been visited

visited[t] = True #mark as visited

prev[t] = source #source now becomes previous

depthFirstR(G,t)

return prev

#this method prints the path solution given the list

def printPath(prev,v):

if prev[v] !=-1:

printPath(prev,prev[v])

print(" -> ", end=' ')

print(v,end=' ')

maze\_rows = 10

maze\_cols = 15

cells = maze\_rows\*maze\_cols

print("The number of cells in the maze is: ", cells)

remWalls = int(input("How many walls do you want to remove?"))

if remWalls < cells - 1:

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

if remWalls == cells - 1:

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

if remWalls > cells - 1:

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

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

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

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

while countSets(newS) > 1:

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

if union(newS, walls[d][0], walls[d][1]): #if they are part of two different sets

walls.pop(d)

vertices = []

while remWalls != 0:

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

if union(newS,walls[d][0],walls[d][1]):

vertices.append(walls[d])

walls.pop(d)

remWalls -= 1

G = adjacencyList(vertices, cells)

visited = [False for i in range(len(G))]

prev = [-1 for i in range(len(G))]

print("Adjacency List: ")

print(G)

print()

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.