Maria F. Corona Ortega
ID: 80560734
E-Mail: mfcoronaortega@miners.utep.edu
CS 2302 | MW 1:30-2:50 am
Instructor: Dr. Olac Fuaentes
TA: Anindita Nath

CS 2302 Lab 7 Report

Introduction

The purpose of this code is to expand the code from Lab 6 into one where we can analyze paths and search algorithms within the graph in order to find solutions to a maze. We are to turn the created maze into an adjacency list then implement Breadth-first and Depth-First algorithms.

Proposed Solution

I will keep the original code to build the walls of the maze and will import only certain methods in DSF class. These methods are *DisjointSetForest, find/find_c, union/ union_by_size* very similarly to the Lab 6 starting point. Additionally from *graphs.py* we import the *random_graph* method for testing purposes. From here I will implement the prompted cases and the search algorithms.

- **Task 1 (Checking possible cases for *n* and *m*):** We prompt the user with the size of the maze gained from Lab 6 code to enter the number of walls to remove and using if statements we generate a maze with *m* walls removed and print possibilities for finding a path.
- **Task 2 (Build an adjacency list):** Using wall list form newly created maze, we convert this information into a 2D list with the index being the vertices and the contents being the vertices to which it is connected to.
- **Task 3 (Implementing search algorithms):** Using the newly diverted adjacency list we implement breadth-first, depth-first and depth-first recursively and return in essence a path that solves the maze.

Methods

UserPrompt: This method follows task number 1 in Lab instructions. We begin by informing the user of the size of the maze and asking how many walls they are wishing to remove. This answer is the parsed into variable *m* and checked with three if statements. First statements checks us *m* is less than the size of the maze minus one. The second checks if *m* is equal to the size of the maze minus one. The last checks if *m* is greater than size minus one. All these print the respective print statements and actually build the maze with *m* walls removed.

SolveBF: We use a boolean type array to keep track of the vertices that we have already visited and we use a queue to keep track of non visited vertices. Additionally I inserted a list named path to keep track of the breadth-first path. While the queue is not empty we equal variable s to queue at 0 and we append this vertex to our path. we then go into a loop that traverses the grapth. if such vertex in the prev array has not been visited we append to the queue and set that vertex to true in the prev array. Once the queue is empty we are left with the path array that kept track of the order of visited vertices which we return.

Conclusion

In conclusion, this lab taught us how to implement algorithms into graphs and connect this to disjoint sets. We also learned the variation within efficient and less efficient algorithms regarding graphs.

Appendix

```
#Course: CS 2302 Data Structures | Spring 2019
#Author: Maria Fernanda Corona Ortega
#Assignment: Lab 7
#Instructor: Olac Fuentes
#Purpose of Code: Starting point for program to build and draw a maze
# Modify program using disjoint set forest to ensure there is exactly
one
# simple path joiniung any two cells
#Last Modification: 4/1/2019

import matplotlib.pyplot as plt
import numpy as np
import random
from scipy import interpolate
import timeit
import dsf

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

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_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 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

plt.close("all")

n = 4
maze_rows = n
maze_cols =  n
walls = wall_list(maze_rows,maze_cols)
size = maze_rows*maze_cols
S = DisjointSetForest(size)
m = size * 2 +1

#print(walls)
#draw_maze(walls,maze_rows,maze_cols,cell_nums=True)

def buildMaze(walls, S, m):
    start = timeit.default_timer()
    for i in range(m):#While S has more than one set
        ran = random.randint(0,len(walls)-1)
        if find_c(S, walls[ran][0]) != find_c(S, walls[ran][1]):
#            print('root of', walls[ran][0],'is',find_c(S, walls[ran]
[0]))
#            print('root of', walls[ran][1],'is',find_c(S, walls[ran]
[1]))
#            print('removing wall ',walls[ran])
#            print()
#            print('union between sets', walls[ran][0], 'and ',
walls[ran][1])
            union_by_size(S,walls[ran][0],walls[ran][1])
#            print('root of', walls[ran][0],'and', walls[ran][1], 'is
NOW',find_c(S, walls[ran][0]))
#            print()
            walls.pop(ran)

    stop = timeit.default_timer()
```

```
#    print('Standard Union running time', stop-start, 'for size n =', n
,' in and nxn matrix')
#    print()
    return walls, S

def maze_toAl(NW, S):
    G = [ [] for i in range(len(S))]
    for i in range(len(G)):#While S has more than one set
#        ran = random.randint(0,len(walls)-1)
        print(NW[i][:])
#        if find_c(S, NW[i][0]) == i:
#            G.append(NW[i])
    print(S)
    print('remaining walls', NW)
    print(G)



#NW,NS = buildMaze(walls, S, m)
#print()
#print(NS)
#print()
#print(NW)
#print()
#draw_maze(NW,maze_rows,maze_cols)
#maze_toAl(NW,NS)

def random_graph(vertices, edges, duplicate=False):
    # Generates random graph with given number of vertices and edges
    # If duplicate is true, each edge is included twice in the list
    # that is, for edge (u,v), u is in G[v] and v is in G[u]
    G = [ [] for i in range(vertices) ]
    n=0
    while n<edges:
        s = random.randint(0, vertices-1)
        d = random.randint(0, vertices-1)
        if s<d and d not in G[s]:
            G[s].append(d)
            if duplicate:
                G[d].append(s)
            n+=1
    return G

G=random_graph(8,6,True)
print(G)


def SolveBF(G,s):
    prev = [False] * (len(G))
#    print(prev)
    queue = []
    queue.append(s)
#    print(queue)
    prev[s] = True
    path = [] # keeps track of Breadth frist path
    while queue:
        s = queue.pop(0)
        path.append(s)
```

```
        for i in G[s]:
#            print(prev[i])
            if prev[i] == False:
                queue.append(i)
                prev[i] = True
#                print(queue)
    return path

print(SolveBF(G, 0))


def SolveDF():
    return

def SolveDF_R():
    return



def UserPrompt():
   print('The current number of cells in this maze is', size)
   user = input('What is the numer of walls m you would like to
remove?')
   m = int(user)
   print()
   print('Choice = ', m)
   print()

   if m < size-1:
       buildMaze(walls, S, size,m*2)
       print('A path from source to destination is not guaranteed to
exist (when m < n - 1)')
   if m == size-1:
       buildMaze(walls, S, size,m*2)
       print('The is a unique path from source to destination (when m =
n - 1)')
   if m > size-1:
       buildMaze(walls, S, size,m*2)
       print('There is at least one path from source to destination
(when m > n - 1)')
```

Academic Honesty Certification

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.

Maria F. Corona Ortega

09/09/2017