LAB 7

CS2302-Graphs

Diego Quinones

Lab Report

Lab 7 built up on Lab 6 to ask us to solve more exercises. I personally had trouble with Lab 6 which initially caused me trouble so I had to fully understand how Lab 6 properly worked. Now that I got more involved with maze building and using disjoint set forests with them and I didn't have much trouble with Lab 7.

We were prompted with the task of modifying our previous maze method so we can ask the user for how many walls he wanted to remove, and based on his answer we would display 3 different outputs, these outputs where decided using a comparison between m (being the user input) and n (the number of total walls).

Our second task was to create a method that built an adjacency list based on the vertices that are created through the maze (not connected walls).

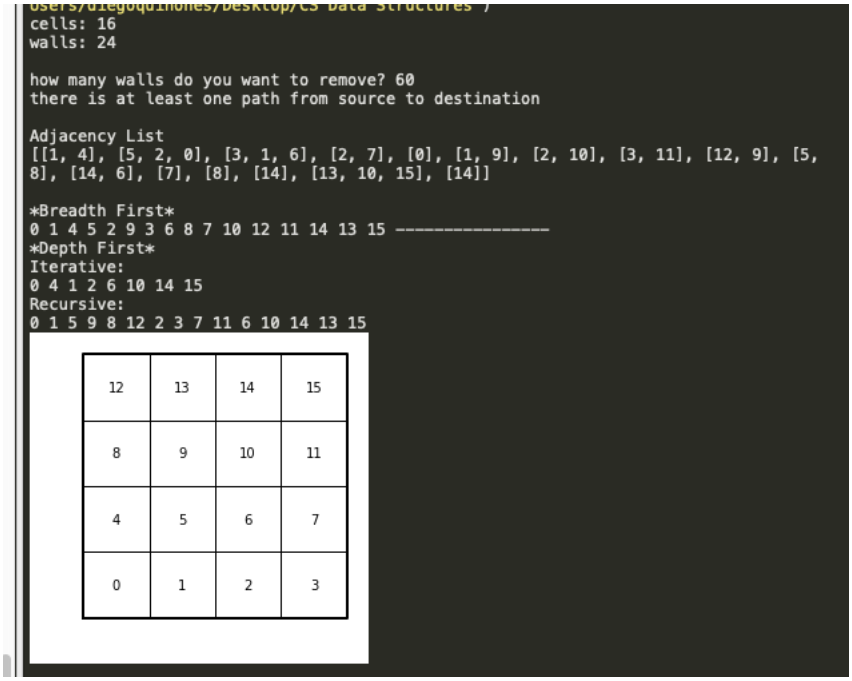
After that we had to use the recently seen on lecture search algorithms, Breadth*first and Depth-first (for this one we used it on its recursive form and its iterative form)

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.”

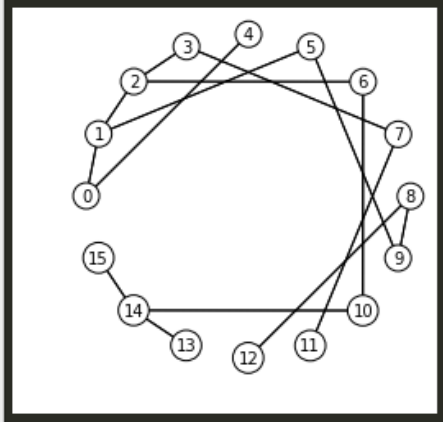
-Diego Quinones

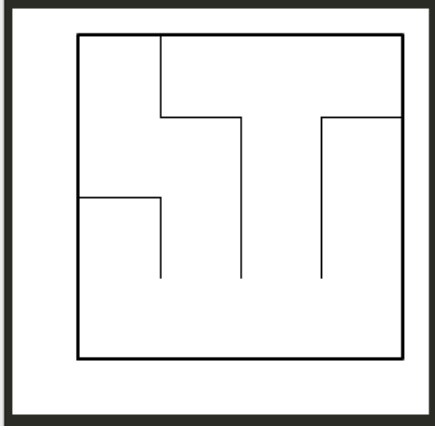
Screenshots

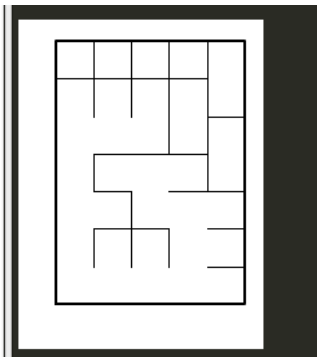


```
In [39]: runfile('/Users/diegoquinones/Desktop/CS Data Structures/x.py', wdir='/
Users/diegoquinones/Desktop/CS Data Structures')
Cells: 35
Walls: 58

How many walls do you want to remove? 60
there is at least one path from source to destination

*Breath First*
0 1 5 2 6 10 7 3 11 15 8 4 20 13 9 25 21 12 14 26 22 17 27 16 18
---------------
*Depth First*
Iterative:
0 5 10 15 20 21 22 27 26 25 11 1 6 2 3 4 8 9 13 14 12 17 18 16 7
Recursive:
0 1 2 7 3 8 13 12 17 16 18 14 9 4 6 5 10 11 15 20 25 21 26 22 27
```

30	31	32	33	34
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4

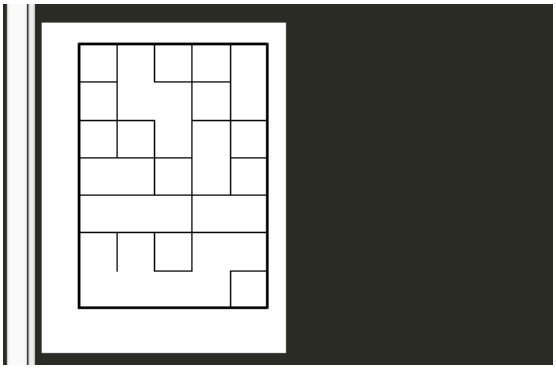






```
cells: 35
walls: 58

how many walls do you want to remove? 30
a path from source to destination is not guaranteed to exist

*Breath First*
0 1 5 2 6 3 8 9
_______________
*Depth First*
Iterative:
0 5 1 6 2 3 8 9
Recursive:
0 1 2 3 8 9 6 5
```

30	31	32	33	34
25	26	27	28	29
20	21	22	23	24
15	16	17	18	19
10	11	12	13	14
5	6	7	8	9
0	1	2	3	4



Source 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
# Programmed by Diego Quinones


import matplotlib.pyplot as plt
import numpy as np
import random
import math

#code from previous lab

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 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(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

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

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 NumSets(S):
    sets = 0
    count = np.zeros(len(S),dtype = int)
    for i in range(len(S)):
        if S[i]<0:
            sets+=1
        count[find(S,i)] +=1
    return sets

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 walls(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 adjacent_finder(walls,n):
    adj = []
    for i in walls:
        if i[0]==n:
            adj.append(i[1])
        if i[1]==n:
            adj.append(i[0])
    return adj

def adjacents(walls,maze_rows,maze_cols ):
    #store adjacent values
    L = [ ]
    for i in range(maze_rows*maze_cols):
        L.append(adjacent_finder(walls,i))
    return L

def wall_finder(walls,c,r):

    for i in range(len(walls)):
        if walls[i]==[c,r] or  walls[i] == [r,c]:
```

```
                return i
    return None

def maze(S,walls,adjacents):
    print('cells:',len(S))
    print('walls:',len(walls))
    #asking the user how many walls to remove and based on their answer we give the the
option
    r = int(input("how many walls do you want to remove? "))
    type(r)
    if r <(len(walls)-1):
        print('a path from source to destination is not guaranteed to exist')
    if r ==(len(walls)-1):
        print("there is a unique path from source to destination ")
    if r>(len(walls)-1):
        print('there is at least one path from source to destination ')
    a = random.randint(0,len(S)-1)
    #array with already visited cells
    v =[]
    for i in range(r):
            r = random.randint(0,len(adjacents[a])-1)
            if find(S,a)!=find(S,adjacents[a][r]):
                #checks if there is a union, if not it creates it
                union(S,a,adjacents[a][r])
                v.append(walls.pop(wall_finder(walls,a,adjacents[a][r])))
                a = r
            else:
                #if not, move to another random cell
                a =random.randint(0,len(S)-1)
    return v

def iterative(adj):
    v = np.zeros(len(adj),dtype = int)
    #list holding visited cells
    s = 0
    stack = []
    stack.append(s)
    v[0]=1
    while stack:
        cell =  stack.pop( )
        print(cell,end=' ')
        if cell == len(adj)-1:
                break
        for i in adj[cell]:
                if v[i] == 0:
                    #values in adj are being stored on the stack
                    stack.append(i)
                    v[i]=1

def recursive(initial,v,adj,maze):
        v[initial] = 1
        maze.append(initial)
        for i in adj[initial]:
            if v[i]==0:
                #recursivecall
                recursive(i,v,adj,maze)

def breadth(adj):
        #start with visited list empty
        v = np.zeros(len(adj),dtype = int)
        q=[]
        c=0
        q.append(c)
        v[c] = 1
        while q:
            c = q.pop(0)
            print (c, end = " ")
            if c == len(adj)-1:
                break
```

```
            for i in adj[c]:
                if v[i] == 0:
                    #append values in adj[c]
                    q.append(i)
                    v[i]=1

def dfs_print(maze,adj):
    for i in maze:
        print(i,end=' ')
        if i == len(adj)-1:
            break

def draw_graph(G):
    fig, ax = plt.subplots()
    n = len(G)
    r = 30
    coords =[]
    for i in range(n):
        theta = 2*math.pi*i/n+.001 # Add small constant to avoid drawing horizontal lines,
which matplotlib doesn't do very well
        coords.append([-r*np.cos(theta),r*np.sin(theta)])
    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')
    for i in range(n):
        ax.text(coords[i][0],coords[i][1],str(i), size=10,ha="center", va="center",
         bbox=dict(facecolor='w',boxstyle="circle"))
    ax.set_aspect(1.0)
    ax.axis('off')



plt.close("all")
maze_rows = 4
maze_cols = 4
walls = walls(maze_rows,maze_cols)
S = DisjointSetForest(maze_rows*maze_cols)
draw_maze(walls,maze_rows,maze_cols,cell_nums=True)
adj = adjacents(walls,maze_rows,maze_cols)
m = maze(S,walls,adj)
adjacentlist=(adjacents(m,maze_rows,maze_cols))
print()
print('Adjacency List')
print(adjacentlist)

print()
print('*Breadth First*')
breadth(adjacentlist)
draw_graph(adjacentlist)
print('----------------')
print('*Depth First*')
print('Iterative:')
iterative(adjacentlist)
visited = np.zeros(len(adjacentlist),dtype = int)
dfs =[]
print()
print('Recursive:')
recursive(0,visited,adjacentlist,dfs)
dfs_print(dfs,adjacentlist)
draw_maze(walls,maze_rows,maze_cols)
```