Lab 7 report

UTEP CS2302

Javier Soto

# Introduction

The purpose of lab 7 is to modify out codes from lab 6 to be able to delete a desired number of walls in the maze. Next, I will need to crate an adjacency list representation of the maze where each cell of the maze is represented by a vertex in the graph. This adjacency list will later be used to solve the maze using either breadth first search, depth first search or depth first search with recursion.

# proposed solution

To remove only a given number of walls I implemented a new function named remove x walls that takes a maze size, walls list, number of walls to removed and as before a Boolean variable stating if the maze should be displayed on the conceal or not. If the size of the maze is to large to make visible comparison, then the wall is not drawn. A disjoint set forest is created equal to the size of the maze. For lab 6 a while loop was set to iterate while there was more than one set in the disjoint set forest to ensure there is path from every cell to every other cell. For this lab that while loop was disregarded and replace with a for loop to iterate the number of times equal to the number of walls to be removed. As in lab 6 through every iteration a random wall is chosen to be removed. The cells adjacent to the wall being removed are then added to the same set in the disjoint set forest and finally the wall is removed. The result is a maze that falls under one of three options. Either there is a guaranteed unique path one cell to every other cell. Option two there is no guaranteed unique path form one cell to any other or finally where there any be more than one path form any cell to any other. Once all desired wall is removed the list of remain walls is returned. This function has a time complexity of O(N^2).

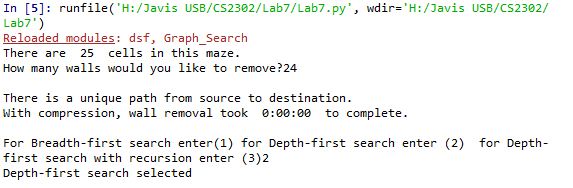
For the second part of the lab I settled with an algorithm that would first create an adjacency matrix based off the remaining walls. I would then create an adjacency list from this adjacency matrix. The creation of the adjacency matrix takes O(|E|) time and the transition form adjacency matrix to adjacency list takes O(|V^2|). To implement I first create an adjacency matrix of size number of cells by number of cells and initialize every index to true meaning there is a path form every cell to every other cell. Since the walls list represents only wall that remain I concluded that no matter which of the three outcomes results, it would be faster to iterate through only the remain walls vs setting the initial adjacency matrix to false, checking for a path and the switching a to true if a path exists. I crate a loop to iterate the walls list for every wall I set the adjacent cells to false for two indexes in the adjacency matrix. That is the both in ward and out ward are false since the wall remain.

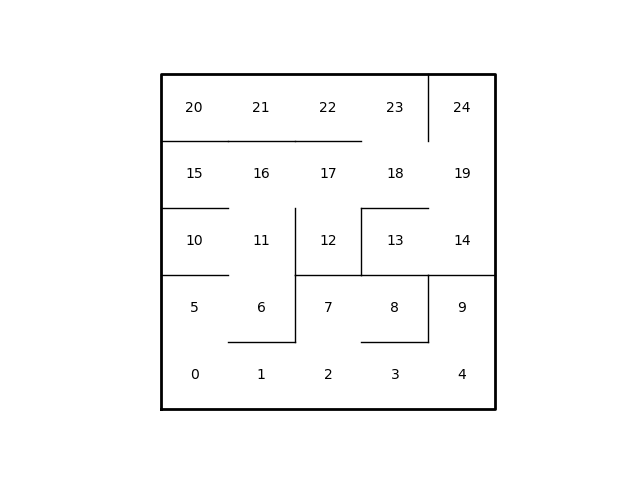
With the newly created matrix I call AM to AL function that creates an adjacency list from this matrix. To do so I crate an empty adjacency list of size equal to the number of vertices in the previous adjacency matrix which we have established is equal to the number of cells in the maze. I then iterate through the entirety of the adjacency matrix, if the current index is true then to the current index of the adjacency list I append the out bound edge. My algorithm can defiantly can be made faster for instance creating only one undirected edge from cell to cell instead of two directed, accessing edges would take half the time. With the time constraint however, my algorithm will suffice for lab 7. Also, I know there are different algorithms that will return a adjacency list from the maze that will be faster than mine. I could not think of one myself, but I do really wish to see that algorithm.

Now that I have the adjacency list I am ready to implement the graph search algorithms. For bread first search I first implement an array of type Boolean if size equal to the size of the graph. Next, I crate an array of the same size of type int that will store the previous vertices in the search. I crate a queue and add the starting vertices to that queue. I set the visited index of that vertices true. I create loop that will run all long as the queue is not empty. In the loop I next get the next item in the queue and add every out bound edge vertices of this vertex to the queue. I set the visited index of those vertices to true and I make the previous index of those vertices equal to the current vertex.

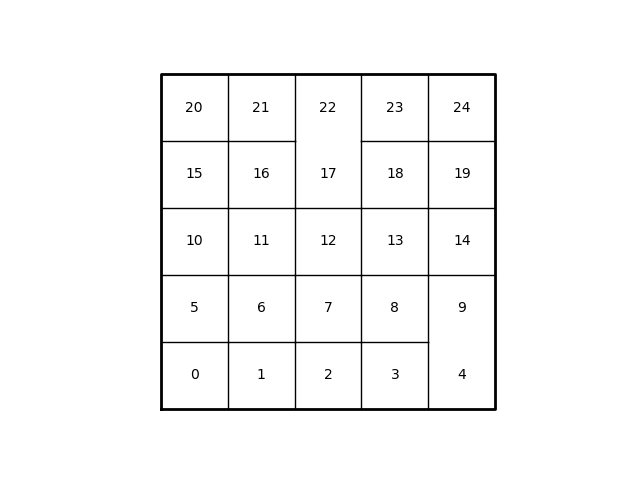
To implement depth first search I use the same algorithm however in stead of using a queue I replace with a stack. To implement a stack, I simple use a list and when popping from the list I always pop at the last index of the list to achieve the last in first out as a stack.

# experimental results

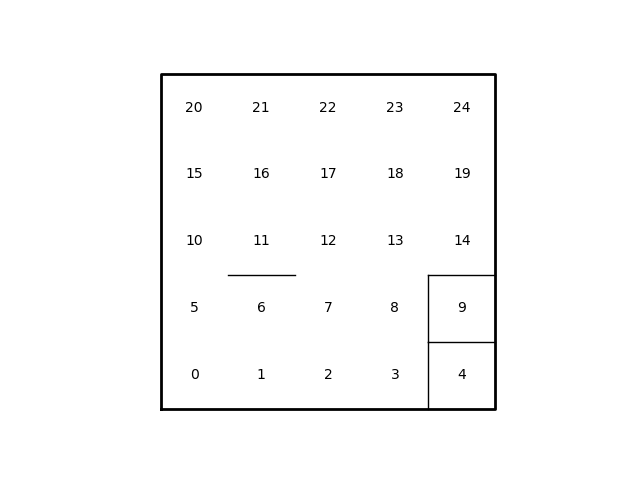




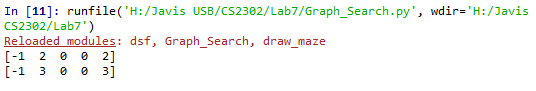
Maze when deleting n-1 walls



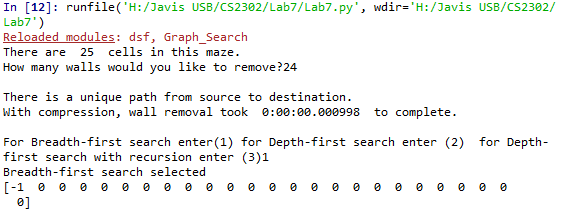
Maze when deleting less than n-1 walls



Maze when deleting more than n-1 walls



Results when testing graph search class. Top line is bread first search second is depth first search.



Failure when testing breadth first search on lab 7

# appendix

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

"""

Created on Wed Apr 17 11:19:59 2019

@author: yatha

"""

import dsf

import draw\_maze as DM

import Graph\_Search as GS

import numpy as np

def solve\_Maze(AL):

search\_type=int(input("For Breadth-first search enter(1) for Depth-first search enter (2) for Depth-first search with recursion enter (3)"))

if search\_type == 1:

print("Breadth-first search selected")

print(GS.breadth\_first\_search(AL,0))

elif search\_type == 2:

print('Depth-first search selected')

return

else:

print('Depth-first search with recursion selected')

return

'''

will take a list/dsf and return an adjacency matrix representation

'''

def list\_to\_AM(L,length,height):

num\_cells = length\*height

AM = np.full((num\_cells,num\_cells),True,dtype=bool)

for i in range(len(L)):

left = L[i][0]

right = L[i][1]

AM[left][right]=False

AM[right][left]=False

return AM

def Send\_It():

maze\_rows =5

maze\_cols = 5

maze\_size= maze\_rows \*maze\_cols

nMinOne = maze\_size-1

Print = False

if maze\_rows < 500 and maze\_cols <500:

Print = True

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

print('There are ',maze\_size, " cells in this maze.", end=" ")

m= int(input("How many walls would you like to remove?"))

print()

if nMinOne == m:

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

#method has been overloaded since original uses golbal variables

walls =DM.remove\_walls\_c(walls,Print,maze\_rows,maze\_cols)

AM =list\_to\_AM(walls,maze\_rows,maze\_cols)

AL = GS.AM\_to\_AL(AM)

#denug here

solve\_Maze(AL)

elif m< nMinOne:

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

#DM.remove\_x\_walls(walls, sets,Print,maze\_rows,maze\_cols,m)

walls =DM.remove\_x\_walls(walls,Print,maze\_rows,maze\_cols,m)

else:

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

walls =DM.remove\_x\_walls(walls,Print,maze\_rows,maze\_cols,m)

#for i in range(len(AM)):

#print(AM[i])

Send\_It()

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

"""

Created on Wed Apr 17 11:22:25 2019

@author: yatha

"""

# Implementation of simple graph operations

# Programmed by Olac Fuentes

# Last modified April 15, 2019

import random

import dsf

import numpy as np

import matplotlib.pyplot as plt

import math

from queue import \*

'''

will take an adjacency matrix and return the adjacency list representation

of the graph

'''

def AM\_to\_AL(AM):

AL = Create\_AL(len(AM))

for i in range(len(AM)):

for j in range(len(AM[i])):

if AM[i][j] == True:

AL[i].append(j)

#AL[j].append(i)

return AL

def breadth\_first\_search(G,v):

#will take a graph represented as Adjaceny list and return an array

#containng the previous vertex of that index

visited = np.full(len(G), False, dtype=bool)

prev = np.full(len(G), -1, dtype=int)

#Q holds items discoverd be still needing to be visited

Q = Queue(maxsize=0)

Q.put(v)

visited[v] = True

while not Q.empty():

u = Q.get(0)

#will print the traversal order

#print(u)

for t in G[u]:

if not visited[t]:

visited[t] = True

prev[t] = u

Q.put(t)

return prev

def depth\_first\_search(G,source,visited,prev):

visited[source] = True

for t in G[source]:

if not visited[t]:

prev[t] = source

depth\_first\_search(G,t,visited,prev)

print(prev)

def depth\_first\_iter(G,v):

#will take a graph represented as Adjaceny list and return an array

#containng the previous vertex of that index

visited = np.full(len(G), False, dtype=bool)

prev = np.full(len(G), -1, dtype=int)

#S holds items discoverd be still needing to be visited

S = []

S.append(v)

visited[v] = True

while len(S)>0:

u = S.pop(-1)

#will print the traversal order

#print(u)

for t in G[u]:

if not visited[t]:

visited[t] = True

prev[t] = u

S.append(t)

return prev

def Create\_AM(rows,cols):

#will create 2d list of false type boolean

#given the size of rows\*columns

x =[]

for i in range(rows):

x.append(False)

AM =[]

for i in range(cols):

AM.append(x)

return AM

def True\_AM(rows,cols):

x =[]

for i in range(rows):

x.append(True)

AM =[]

for i in range(cols):

AM.append(x)

return AM

'''

#creates an empty Adjacency list

'''

def Create\_AL(vertices):

temp = []

AL =[]

for i in range(vertices):

AL.append(temp)

return AL

def dijkstra(G,source):

Known = np.full(len(G), False, dtype=bool)

prev = np.full(len(G), -1, dtype=int)

dist = np.full(len(G), sys.maxint, dtype=int)

dist[source] =0

knownVertices = 0

while KnownVertices < len(G):

return

return

def adj\_list\_to\_adj\_mat(G):

g = np.zeros((len(G),len(G)),dtype=bool)

for source in range(len(G)):

for dest in G[source]:

g[source,dest] = True

g[dest,source] = True #Comment out if graph is directed

return g

def adj\_list\_to\_edge\_list(G):

g = []

for source in range(len(G)):

for dest in G[source]:

if dest>=source: #ignore duplicate edges

g.append([source,dest])

return g

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

def random\_graph2(vertices):

G =[]

for i in range(vertices):

G.append([])

for s in range(vertices):

d = random.randint(1, vertices-1)

d = (d+s)%vertices

G[s].append(d)

return G

def connected\_components(G,diplay\_dsf=False):

S= dsf.DisjointSetForest(len(G))

for source in range(len(G)):

for dest in G[source]:

dsf.union\_by\_size(S,source,dest)

if diplay\_dsf:

dsf.draw\_dsf(S)

return dsf.NumSets(S), S

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')

if \_\_name\_\_ == "\_\_main\_\_":

plt.close("all")

random.seed(a=86)

#G=random\_graph(8,6,True)

#draw\_graph(G)

#print('Adjacency list representation:')

#print(G)

#AM = adj\_list\_to\_adj\_mat(G)

#print('Adjacency matrix representation:')

#print(AM)

#print('Edge list representation:')

#EL = adj\_list\_to\_edge\_list(G)

#print(EL)

#n,S = connected\_components(G,True)

#print('Connected components=',n)

#dsf.draw\_dsf(S)

#print('Sets:',dsf.dsfToSetList(S))

AM = [[False, False, True, True, False],

[False, False, True, True, False],

[True, True, False, False, True],

[True, True, False, False, True],

[False, False, True, True, False]]

AL= [[2,3],

[3,2],

[0,1,4],

[0,1,4],

[2,3]]

EL = [[0,2],[0,3], [1,3], [1,2], [2,4], [3,4]]

visited = np.full(len(AL), False, dtype=bool)

prev = np.full(len(AL), -1, dtype=int)

#draw\_graph(AL)

test = breadth\_first\_search(AL,0)

print(test)

print(depth\_first\_iter(AL,0))

# depth\_first\_search(AL,0,visited,prev)

Academic dishonesty

I, Javier Soto, certify that this script and lab report are of my own unless otherwise documented above.

