Introduction:

The problem that was to be solved was to make a maze out of disjoint set forests and apply different searches to it

Solution:

I first attempted at the problem by writing pseudocode and trying to make sense of it. I then implemented it into code and ran several times. Each time I ran it I got close to solving the problem. I finally followed it up with actually coming up with the solution.

Results:

The experiments included running the program several times, and finding out the outputs. The running times.



Conclusions:

I learned that disjoint set forests are very useful.

I Yury Ionov 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.



Source code:

import matplotlib.pyplot as plt

import numpy as np

import random

import time

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 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: # Do nothing if i and j belong to the same set

S[rj] = ri # Make j's root point to i's root

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):

ri = find\_c(S,i)

rj = find\_c(S,j)

if ri!=rj:

if S[ri]>S[rj]:

S[rj] += S[ri]

S[ri] = rj

else:

S[ri] += S[rj]

S[rj] = ri

def numOfSets(S):

total=0

for i in range(len(S)):

if S[i] <= -1:

total = total + 1

return total

def adjacencyListBuilder(originalWalls, modifiedWalls, rows, cols):

numOfCells = rows \* cols

if len(originalWalls) == len(modifiedWalls):

return [[] for i in range(numOfCells)]

adj\_list = [[] for i in range(numOfCells)]

wallsRemoved = []

for i in range(len(originalWalls)):

if originalWalls[i] not in modifiedWalls: #check to see which walls were removed

wallsRemoved.append(originalWalls[i]) #make a list of all the walls removed

"""

This is where I use the list of all of the walls removed to get the propper

indices for the adj\_list, then I append the other part of the wall to the

adj\_list. Example, wall [1, 0], 1 would be the an index for the adj\_list

then I would append 0 to adj\_list[1] and vice versa.

"""

for i in range(len(wallsRemoved)):

adj\_list[wallsRemoved[i][0]].append(wallsRemoved[i][1])#use the walls removed as indices for the adj\_list

adj\_list[wallsRemoved[i][1]].append(wallsRemoved[i][0])

return adj\_list

def breadthFirstSearch(G, startingVertex, end):

"""

I used a queue to store all of the neighbors of a node. I used a

normal list to represent a queue by poping (0). The popped items

get put in the visited list.

"""

visited = []

Q = [startingVertex]

while Q != []:

node = Q.pop(0) #acts like a queue

#adds node to the visited list

if node not in visited:

visited.append(node)

if node == end:

return visited

neighbors = G[node]

#adds neighbors to the queue

for i in neighbors:

Q.append(i)

return visited

def depthFirstSearch(G, startingVertex, end):

"""

I used a stack to store all of the neighbors of a node

when it a node was popped from the stack it'd go into the

visited list

"""

visited = []

stack = [startingVertex]

while stack != []:

current = stack.pop()

for i in G[current]: #i are the neighbors

if i not in visited:

stack.append(i)

visited.append(current)

if current == end: #stops once it reaches its destination

return visited

return visited

def depthFirstSearch\_Recursive(graph, vertex, path=[]):

path += [vertex]

for neighbor in graph[vertex]:

if neighbor not in path:

path = depthFirstSearch\_Recursive(graph, neighbor, path)

return path

def wallRemover(wallList, numRemoved, disjointSet):

if numRemoved > len(wallList):#doesn't do anything if the number of walls is greater than the length of the list

return wallList

removedWalls = []

while len(removedWalls) != numRemoved:#this is where I get the indices that I'm going to remove

randomIndex = random.randint(0, len(wallList) - 1)

if randomIndex not in removedWalls: #this is so there won't be any repeated inices

removedWalls.append(randomIndex)

a = sorted(removedWalls)

for i in range(len(removedWalls)):

wallList.pop(a[len(a) - 1]) #wallList and a had to be popped so there wouldn't be an out of bounds error

a.pop()

return wallList

def wallRemover2(wallList, disjointSet):

while numOfSets(S)>1:

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

if find(S,walls[d][0]) != find(S,walls[d][1]):

union(S,walls[d][0],walls[d][1])

walls.pop(d)

plt.close("all")

maze\_rows = 12

maze\_cols = 12

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

wallsCopy = wall\_list(maze\_rows, maze\_cols) #a copy to store orignail wall list

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

start\_time = time.time()

print("Number of cells:", maze\_rows\*maze\_cols)

numOfWallsToRemove = int(input("Enter the number of walls to remove \nWalls removed: "))

i=0

"""

while int(numOfWallsToRemove)>i:

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

if find(S,walls[d][0]) != find(S,walls[d][1]):

union(S,walls[d][0],walls[d][1])

walls.pop(d)

i=i+1

"""

newWalls = wallRemover(walls, numOfWallsToRemove, S)

if int(numOfWallsToRemove)<(maze\_rows\*maze\_cols)-1:

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

if int(numOfWallsToRemove)==(maze\_rows\*maze\_cols)-1:

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

if int(numOfWallsToRemove)>(maze\_rows\*maze\_cols)-1:

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

adj\_list = adjacencyListBuilder(wallsCopy, newWalls, maze\_rows, maze\_cols)

print(adj\_list)

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

print()

bfs = breadthFirstSearch(adj\_list, 0, len(adj\_list) - 1)

print('breadth first search: ', bfs)

print()

dfs = depthFirstSearch(adj\_list, 0, len(adj\_list) - 1)

print('depth first search: ', dfs)

print()

print()

dfs2=depthFirstSearch\_Recursive(adj\_list, 0, path=[])

print('depth first search recursively: ', dfs2)

print()

print(time.time()-start\_time)