Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: April 30, 2019

Instructor: Olac Fuentes

Assignment: Lab 7 Modifed Maze

TA: Anindita Nath & Maliheh Zaragan

**Introduction:**

In the last lab, we were introduced to disjoint set forests and the many uses it has for various instances it can be used for, such as making a maze from scratch. In this lab, I will further explore the ideas portrayed from lab 6, and I will alter it so that it doesn’t remove one wall at a time, but rather if the maze has a certain number of cells, that it removes exactly that number minus one to reach its destination. I should also indicate whenever that situation has less than one, more than one, or equal to the walls necessary to make the maze. I should also build an adjacency list for when the maze is built in its final form. Finally, I should be able to use three algorithms, breadth-first, depth-first using a stack, and depth-first using recursion to solve the maze.

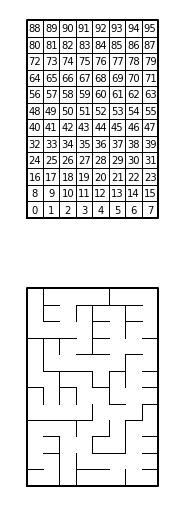
**Proposed solution design and implementation:**

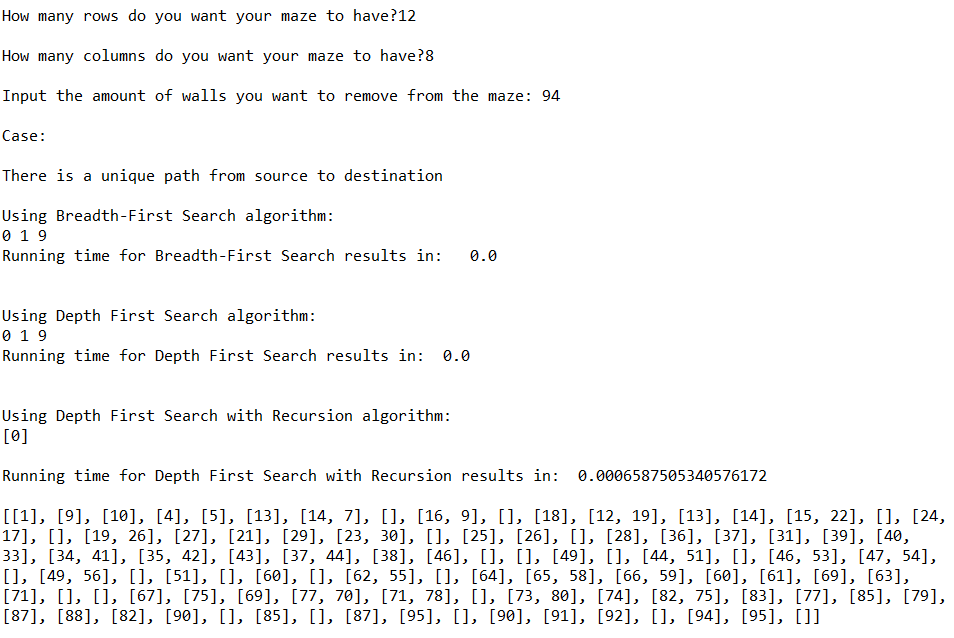
1. Path Message: This part of the lab should not take me long as it is the easiest part of the whole process. All I need to do is print specific messages when I have a certain value n, or number of cells, and a value m, the number of walls removed in the maze. Depending on the value of n and m, it will display a message to the user as asked in the lab.
2. Adjacency list: For this problem, all I need to do is built a method that builds the adjacency list on a new blank list, and then returns that list to be later printed to the user. The method to print the adjacency list will have to be a bit tricky to make, my best guess is that it probably needs to have some form of recursion in it so that it builds the list one adjacency a time.
3. Algorithms: Probably the hardest part of the lab, and quite frankly, the lengthiest, I need to find a way to implement, in various methods, a way for the maze to be built using breadth-first search, and the two forms of depth-first searches as well. For breadth-first search, I will need to find away to check each individual wall in a way that it mimics the methods I used in lab 6, but not in the way that it checks it one by one as that will take too much time, my best guess is that I need to use a list to keep track of all the lists that have been included on the maze. For the depth-first, I will need to know how deep the maze goes, and from there we can start to pick out the walls using either stacks and recursion.

**Experimental results**:

1. Path Message: As mentioned before, this part was not too difficult as all I needed to do was to print three different messages depending on what the values of n and m where. In my lab however, the names of these values are different, n is numCells, and m is walls\_removed in my lab. When m less is than n-1, it will print 'A Path from source to destination is not guaranteed to exist'. When m = n-1, it will print 'There is a unique path from source to destination'. When m is greater than n-1, it will print 'There is at least one path from the source to destination'. Adjacency list: For my solution, I essentially used a method that removes the walls, but I wrote the function on it before getting to the algorithms as it would be easier to call on the list to the algorithms, and it also served as an easier way to read where my logic for this lab was located. After
2. Algorithms: For breadth-first search, I used a new list called K, and I made sure that when the wall was already visited, then the wall would be added to that list, and would continue to visit the rest of the walls contained inside of the maze. For the depth-first method using stacks, I will essentially made a new stack that would append for every wall already visited, and ignore the neighbor if necessary, the one for depth-first using recursion works the same way, just that it uses recursion to make the list, and it inserts them into a normal list rather than a stack.

The results running a 12x8 maze with 94 walls removed is shown below:





**Running times Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Attempt | Breadth-first Search | Depth-first Search (Stack) | Depth-first Search (Recursion) |
| 1 | 0.000994 sec | 0.001086 sec | 0.000085 sec |
| 2 | 0.001085 sec | 0.000968 sec | 0.409376 sec |
| 3 | 0.000968 sec | 0.000999 sec | 0.000089 sec |

As seen above, the method with the overall best time was the depth-first search using a stack, as it has the overall best time compared to breadth first search, and the depth first search using recursion. This is not entirely

**Conclusions**:

With this lab, I was able to learn to code better using the Python language, including using algorithms to create and modify disjoint set forest to create a maze with multiple solutions. I was also able to learn to solve different problems by using breadth-first and depth-first searches, throughout my lab, including to determine the placement of each wall found throughout the mazes.

**Appendix :**

**lab7.py**

"""

Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: April 29, 2019

Instructor: Olac Fuentes

Assingment: Lab 7 Modified Maze

TA: Anindita Nath & Maliheh Zaragan

Purpose: to implement both standard union and compression techniques

as well as new techniques to modify the maze program to create a new

solution for that specififcally maze.

"""

#Imports various tools to help us calculate the hash tables to be used in this lab

import time

import matplotlib.pyplot as plt

import numpy as np

import random

#Makes a new class that will create the functions necessary to make a graoh

class Graph:

def \_\_init\_\_(self,vertices):

self.vertices = vertices

self.graph = []

for v in range(vertices):

self.graph.append([])

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

#Method that creates a Disjoint Set Forest

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\_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

#Method that finds the number of cells contained in the maze

def cells\_number(S):

count = 0

#If S is none, then it returns 0

if S is None:

return 0

#Else, it will use a for loop to count the number of cells

for i in S:

count+=1

#Returns the number of cells labeled count

return count

#Method that where n displays the number of cells and m is the amount of walls the user wants to remove

def cells\_display(n,m):

#While m is less than n minus 1

if m < n-1:

print('A Path from source to destination is not guranteed to exist')

#While m is equal to n minus 1

elif m == n-1:

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

#While m is greater than n minus 1

elif m > n-1:

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

#method to use depth First Search using a stack to answer question 3

def dfsStack(adjList, initialNode):

#Creates a new list that will add the visited nodes

visitedNode = []

#Creates the stack from the initial Node

stack = [initialNode]

#While Stack is True

while stack:

#Sest Inititial Node to pop

initialNode = stack.pop()

#Adds the visited node to the new list

visitedNode.append(initialNode)

#Prints the current node

print(initialNode,end=" ")

#For i in the adjacency list started from intitial node

for i in adjList.graph[initialNode]:

#Append to to the stack

stack.append(i)

#Returns the list visitedNode

return visitedNode

#Method to use depth First Search with recursion to answer question 3

def dfsRecursion(adjList,startNode, visited = None):

#If the visited none is None, it will create a new list called visited

if visited is None:

visitedNode = []

#Appends startNode to list visited

visitedNode.append(startNode)

#For i in the adjacency list

for i in adjList.graph[startNode]:

#If i has not been visisted, recaals itslef recursively until visisted

if i not in visitedNode:

dfsRecursion(adjList,i,visited)

#Returns the list visited

return visitedNode

#Method that creates the adjacency List and applies it to this lab better unlike the graph

def addEdge(G,v1,v2):

G.graph[v1].append(v2)

#Method to use Breadth First Search used for Question 3

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

#Checks if it has visited that cell

visit\_check = [False]\*(len(adjList.graph))

#Creates a new list called K

K=[]

#Adds the item v to list K

K.append(v)

#Changes the visted check to True

visit\_check[v] = True

#While K is true

while K:

#Pops item 0 from K to v

v = K.pop(0)

#Prints the vector v

print(v,end=" ")

#FOr i in adjacency list

for i in adjList.graph[v]:

#If the visit check is false, append to list K, and check visit check to true

if visit\_check[i]==False:

K.append(i)

visit\_check[i]=True

#When the graphs show up, it will dsiplay the close points set to all

plt.close("all")

#Asks the user to input the number of rows he/she wants to have in their maze, the number will then be assigned to variable rows

r = input('How many rows do you want your maze to have?')

rows = int(r)

#Asks the user to input the number of columns he/she wants to have in their maze, the number will then be assigned to variable columns

c = input('How many columns do you want your maze to have?')

columns = int(c)

#Creates the adjacency list necessary for the program by using a new class called 'Graph', this is basically n

ad\_list=Graph(rows\*columns)

#Asks the user to input the number of walls he/she wants to remove, the number will then be assigned to variable, this is basically m

x=input('Input the amount of walls you want to remove from the maze: ')

walls\_removed = int(x)

walls\_removed += 1

#Gets the list of walls in the maze by using the method wall\_list from the previous lab

walls\_number = wall\_list(rows,columns)

#Uses the draw maze method and makes the complete maze without any deletion

draw\_maze(walls\_number,rows,columns,cell\_nums=True)

# makes the new DSF by combining the rows and columns

S = DisjointSetForest(columns\*rows)

# cells amount of the dsf

numCells = cells\_number(S)

#Used to answer question question and then displays the message for the user

print()

print('Case:')

print()

cells\_display(numCells,walls\_removed)

#This essentially serves as the remove walls method so it can be used in the main section

while walls\_removed > 0:

#W is a wall that gets randomly selected

w = random.choice(walls\_number)

#Gets the position 'p' where we chose the wall to delete

p = walls\_number.index(w)

#If it finds that wall 0 does not equal to 1, then it deletes the wall, unites after deletion, and sends it to the adjacency list used for question 2

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

walls\_number.pop(p)

union(S,w[0],w[1])

addEdge(ad\_list,w[0],w[1])

walls\_removed -= 1

print()

#Uses various path algorithms and prints out their running time

startTime1=time.time()

print('Using Breadth-First Search algorithm:')

breadth\_first\_search(ad\_list,0)

endTime1=time.time()

finalTime1 = endTime1-startTime1

print()

print('Running time for Breadth-First Search results in: ',finalTime1)

print()

print()

startTime2=time.time()

print('Using Depth First Search algorithm:')

dfsStack(ad\_list,0)

endTime2=time.time()

finalTime2 = endTime2-startTime2

print()

print('Running time for Depth First Search results in: ',finalTime2)

print()

#Uses

print()

startTime3=time.time()

print('Using Depth First Search with Recursion algorithm:')

print(dfsRecursion(ad\_list,0))

endTime3=time.time()

finalTime3 = endTime3-startTime3

print()

print('Running time for Depth First Search with Recursion results in: ',finalTime3)

print()

#Use this to print the graph with the adjacency List, this is essentially used to answer question 2 in the form of a list

draw\_maze(walls\_number,rows,columns)

#Prints the final graph to be displayed to the user

print(ad\_list.graph)

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 provide inappropriate assistance to any student in the class.

