Armando Ortiz

May 1, 2019

CS 2302 Data Structures

Lab Report 7

UTEP

Instructor: Olac Fuentes

**Introduction**

The purpose of this program is to use different functions of graphs to modify the mazes that were created in Lab 6. The first goal was to allow the user to decide how many walls they wanted removed from the maze, and to notify them if there was a path to the end. The second problem was to create a graph of the vertices and edges inside of the maze. The final problem was to use breadth-first search, depth-first search using stacks, and depth-first search using recursion to create a path from the beginning of the maze to the end.

**Proposed Solution and Design Implementation**

The process to allowing the user to decide the number of walls removed was done by modifying the maze building function that uses union by size from lab 6. That function would make sure all the cells were connected, and that there was one main path, and it would stop until there was 1 set left. This function stops at m, the number of cells the user wants removed. This function would basically run the same as the other unless the user picked a number larger than number of walls -1. If the user picked this the loop enters an if-statement that uses regular union and pops walls until the number is reached. The popped walls are also placed in a list representing open paths. A try-except was used to warn the user in case they picked a number larger than the set of walls.

Drawing a graph was done by creating an adjacency list using the path list made in the maze building function. This function was then sent to the graph drawing function. The function provided to draw a graph was edited to change the shape drawn by the graph, which is the shape of the maze. The coordinates used to draw the maze are a modified version of the maze drawing function, combined with the bubbled numbers of the graph drawing function.

The breadth-first search and depth-first search using recursion functions were created using the pseudocode provided by professor. The queue for the breadth-first search was imported from the queue module on python. Depth-first search using a stack was done iteratively. This was accomplished by using a list as a stack. The first item was entered into a while loop, and this is performed until the stack is empty. The current item is popped item from the stack, and then if that item wasn’t already visited the neighbors of that item were checked to see if those were visited, and if they weren’t they were added to the stack, and changed in the prev list.

In a separate function a list is created by moving through the prev list recursively, this new list creates the path used to move through the maze. The function to draw the path through the maze was done by using a modified version of the maze drawing function. Once the maze is created a separate for loop goes through the path, connecting the items in the path according to their distance from each other. The origin points for the x and y-axis are modified so they are in the center of the maze. This is done until the list reaches the second to last item in the list. The connection from the final item in the path list is then connected to the final edge in the al at the location of the final path item.

**Results**

**Output**

The following maze examples were drawn using 10 by 5 size mazes. The biggest issue with graph is that it becomes difficult to see as the mazes grow in size.

Graph

A close up of a keyboard

Description automatically generated

m = n-1

A close up of a clock

Description automatically generated

m < n-1

A picture containing shoji

Description automatically generated

m > n-1

A clock on each of it s sides

Description automatically generatedA close up of a clock

Description automatically generatedA close up of a clock

Description automatically generated

**Runtimes (Build and Draw paths)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Maze Size** | **Breadth First Search** | **Depth First Search (Stack)** | **Depth First Search (Recursion)** |
| **10 x 5** | ﻿0.0776 | ﻿0.0727 | ﻿0.0803 |
| **20 x 10** | ﻿0.2380 | ﻿0.2261 | ﻿0.2286 |
| **30 x 15** | ﻿0.5172 | ﻿0.5087 | ﻿0.6027 |
| **40 x 20** | ﻿0.9446 | ﻿1.1140 | ﻿0.9816 |
| **50 x 25** | ﻿1.8399 | ﻿1.9362 | ﻿2.0186 |

**Conclusion**

The search algorithms seem take about the same amount of time. The part taking up the bulk of the runtime is drawing the path and maze for each of these paths. This causes the runtime to double in size as the maze grows. The interesting thing about these different methods of searching is that in mazes with more than one path they can be used together to find the different possible paths, and to not even draw if there is no path to be found. This creates an interesting method for creating and solving mazes.

**Appendix**

﻿"""

Author: Armando Ortiz

Course: CS 2302 Data Structures

Assignment: Lab 7

Instructor: Olac Fuentes

T.A.: Anindita Nath, Maliheh Zargaran

Last Modification: May 1, 2019

Purpose: The purpose of this lab is to modify the maze building program with new features

- Allow the user to pick the number of walls to remove, while warning them if there might be less paths

- create a graph of the adjacency list of open cells in the maze

- Perform Breadth-First Search,Depth-First Search with stacks, and Depth-First Search using recursion

- Display the path of these search functions in the maze

"""

import matplotlib.pyplot as plt

import numpy as np

import random

import time

import queue

# Initialize Disjoint Set Forest with size of maze

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

#Builds maze using union by size

def mazeUBS(walls,D,m):

i = 0

path = []#Array to build adjacent list of paths through maze

while i <= m: #While there is more than one set

w = random.randint(0,len(walls)-1)# picks a random wall in the set

c1 = find(D,walls[w][0])# Finds root of set

c2 = find(D,walls[w][1])# Finds root of set

if i >= len(D)-1:

union(D,c1,c2)

path.append(walls.pop(w))

i+=1

elif c1!= c2:# If sets are different

union\_by\_size(D,c1,c2)# Uses union by compression

path.append(walls.pop(w))#pops the wall while adding it to path array

i +=1

return path

def adjlist(L,n):

AL = [[] for i in range(n)]

for d in range(n):

for j in range(len(L)):

if L[j][0] <= d:

if d == L[j][0]:

AL[d].append(L[j][1])

elif d == L[j][1]:

AL[d].append(L[j][0])

return AL

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

# Draws the maze

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)

#Draws graph of connected vertices

def draw\_graph(G,maze\_rows,maze\_cols):

fig, ax = plt.subplots()

for i in range(len(G)):

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

if G[i][j] - i == 1:#horizontal lines

x0 = i%maze\_cols

x1 = x0+1

y0 = i//maze\_cols

y1 = y0

elif G[i][j] - i == maze\_cols:#vertical lines

x0 = i%maze\_cols

x1 = x0

y0 = i//maze\_cols

y1 = y0+1

ax.plot([x0,x1],[y0,y1],linewidth=1,color='k')

for r in range(maze\_rows):

for c in range(maze\_cols):

cell = c + r\*maze\_cols

ax.text((c),(r),str(cell), size=8, ha="center", va="center",

bbox=dict(facecolor='w',boxstyle="circle"))

ax.set\_aspect(1.0)

ax.axis('off')

#Draws path through maze

def draw\_path(walls,al,path,maze\_rows,maze\_cols):

fig, ax = plt.subplots()

for w in walls:#Draws walls of maze

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')

for i in range(len(path)-1):#Draws Path through maze

if path[i+1] - path[i] == 1:#horizontal lines

x0 = (path[i]%maze\_cols)+.5#Centers path on the x-axis

x1 = x0+1

y0 = path[i]//maze\_cols+.5

y1 = y0

elif path[i] - path[i+1] == 1:#horizontal lines

x0 = (path[i]%maze\_cols)+.5

x1 = x0-1

y0 = path[i]//maze\_cols +.5

y1 = y0

elif path[i+1] - path[i] == maze\_cols:#vertical lines

x0 = (path[i]%maze\_cols)+.5

x1 = x0

y0 = path[i]//maze\_cols +.5

y1 = y0+1

elif path[i] - path[i+1] == maze\_cols:#vertical lines

x0 = (path[i]%maze\_cols)+.5

x1 = x0

y0 = path[i]//maze\_cols +.5

y1 = y0-1

ax.plot([x0,x1],[y0,y1],linewidth=1,color='r')

fs = maze\_rows\*maze\_cols-1

if fs in al[path[-1]]:#if final item in path is points to 19

if fs - path[-1] == 1:

x0 = (path[-1]%maze\_cols)+.5

x1 = x0+1

y0 = path[-1]//maze\_cols+.5

y1 = y0

else:

x0 = (path[-1]%maze\_cols)+.5

x1 = x0

y0 = path[-1]//maze\_cols +.5

y1 = y0+1

ax.plot([x0,x1],[y0,y1],linewidth=1,color='r')

ax.set\_aspect(1.0)

ax.axis('off')

#Builds path through maze

def Path(prev,v,path):

if prev[v] != -1:

Path(prev,prev[v],path)

path.append(v)

#Search Algorithms

# Breadth-first Search

def breadthFirstSearch(G,v):

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

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

Q = queue.Queue(len(G))

Q.put(v)

visited[v] = True

while Q.empty() == False:

u = Q.get()

for t in G[u]:

if(visited[t]==False):

visited[t] = True

prev[t] = u

Q.put(t)

return prev

#Depth-first search using Stack

def depthFirstSearch(G,source):

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

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

stack = []

stack.append(source)

while len(stack) > 0:

c = stack.pop()

if visited[c] == False:

visited[c] = True

for i in G[c]:

if visited[i] == False:

stack.append(i)

prev[i] = c

return prev

#Depth-first search using recursion

def dfsrecursive(G,source,visited,prev):

visited[source] = True

for t in G[source]:

if (visited[t] == False):

prev[t] = source

dfsrecursive(G,t,visited,prev)

plt.close("all")

maze\_rows = 50

maze\_cols = 25

path = []

n = maze\_rows\*maze\_cols

print("Number of Cells:",(n))

# Maze using regular union

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

try:

m = int(input("How many walls do you want to remove? "))

except:

print("Invalid entry")

else:

if m < n-1:

print("A path is not guaranteed to exist.")

elif m == n-1:

print("There is a unique path to the destination")

else:

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

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

try:

path = mazeUBS(walls,D,m)

except:

print("Too many walls removed")

else:

al = adjlist(path,n)

draw\_graph(al,maze\_rows,maze\_cols)

draw\_maze(walls,maze\_rows,maze\_cols)

# Breadth-First Search

start = time.time()

r = breadthFirstSearch(al,0)

pathBFS = []

Path(r,r[-1],pathBFS)

draw\_path(walls,al,pathBFS,maze\_rows,maze\_cols)

end = time.time()

rtBFS = end - start

# Depth-First Search using Stack

start = time.time()

dfs = depthFirstSearch(al,0)

pathDFS = []

Path(dfs,dfs[-1],pathDFS)

draw\_path(walls,al,pathDFS,maze\_rows,maze\_cols)

end = time.time()

rtDFS = end-start

# Depth-First Search using Recursion

start = time.time()

visit = np.zeros(len(al),dtype = bool)

prev = np.zeros(len(al),dtype = int)-1

dfsrecursive(al,0,visit,prev)

pathDFSR =[]

Path(prev,prev[-1],pathDFSR)

draw\_path(walls,al,pathDFSR,maze\_rows,maze\_cols)

end = time.time()

rtDFSR = end - start

print("Breadth First Search runtime:",rtBFS)

print("Depth First Search using Stack runtime:",rtDFS)

print("Depth First Search using Recursion runtime:",rtDFSR)

A screenshot of a cell phone

Description automatically generated