**CS2302 - Data Structures**

**Spring 2019**

**Lab Report 7**

May 1, 2019

Sebastian Gomez

**Introduction:**

Using the code for making a maze by removing random walls, modify it to remove only the amount the user wants to remove. Depending on the choice, display a message regarding the situation if it has multiple paths, just one or not possible. Using 3 search algorithms, find a path from the bottom left, being the source, to the top right, being the destination.

**Implementation:**

I started from the code we modified for lab 6 in order to do the maze belonging to one set. For this lab, we were supposed to remove the walls that the user wanted to remove. Depending of the choice made by the user, a message was displayed. If the user chose a number less than the number of cells, a path was not possible. In order to remove only the walls that the user wanted, I modified my method maze by union to do what the user wanted. If the user typed the exact number of cells, it would display a message saying that there was only one path to the destination, but it the number typed was greater than, it would print that there were multiple paths. For this, I used an if statement to achieve the 3 possible outcomes. I modified my method maze by union by changing the while condition. Instead of being until one set was accomplished, I changed to while the number chosen by the user was not zero, then it would unite the sets and remove the walls. To do so, after uniting sets and removing walls, I decremented the value the user typed. When it reached zero, it would exit the while loop and stop removing walls.

I had a problem creating adjacency list from the maze because I did not know how to create the method. I know that depending on the connections of sets, I would connect the vertexes to do the adjacency list, but I did not know how to coded it. Unfortunately, because of this I could not implement the 3 search algorithms to know the desired path from source to destination, source being the bottom left and destination being the top right. Even though I could not do this, I still did the code for the three search algorithms. For the breadth first search, it needs to have to arrays that are if a vertex has been visited, being a Boolean array, and a previous array, to know the point before you arrived to a vertex. For this algorithm, a queue is needed to go through the vertexes. You start by enqueueing the element given in the parameter, and while the queue is not empty, you dequeue and then check if the vertex has been visited, knowing this from the array, and if not then it would mark the vertex as visited, it would store the predecessor and it would enqueue in order for the while condition be in order. When it reaches an empty queue, it would stop and then the shortest path would be available. For the depth first search it was the same idea and process, but instead of using a queue, it would use a stack. A stack works as the first goes in it would be the last one to go out, differing from the queue, which is first in first out. The third search algorithm was also depth first search but done recursively. To achieve this, it would use the same visited and previous array, but it would not the stack or the queue. Instead, it goes through every element and then stops when it has nothing else to traverse. Unfortunately, I could not implement these algorithms due to my failure of doing the adjacency list but the running times are the following:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Breadth First Search | Depth First Search Stack | Depth First Search Recursion |
| Running Times | O(V+E) | O(V+E) | O(V+E) |

**Conclusion:**

This lab showed how useful can be breadth first search to find the shortest path, and how the other search algorithms vary. Even though, I could not completely do the lab, I think that I need to reinforce this topic in order to improve.

**I, Sebastian Gomez, certify that this project is entirely my own work, I wrote, debugged, and tested the code being presented, performed 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.**

**Appendix:**

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

**"""**

**@author: Sebastian Gomez**

**Course: Data Structure 2302**

**Assignment: Lab 7**

**Instructor: Olac Fuentes**

**T.A: Anindita Nath and Maliheh Zargaran**

**Purpose: Find the path from source to destination using different algorithmic searches.**

**"""**

**import matplotlib.pyplot as plt**

**import numpy as np**

**import random**

**from scipy import interpolate**

**import time**

**import math**

**class Stack:**

**def \_\_init\_\_(self):**

**self.items = []**

**def isEmpty(self):**

**return self.items == []**

**def push(self, item):**

**self.items.append(item)**

**def pop(self):**

**return self.items.pop()**

**def peek(self):**

**return self.items[len(self.items)-1]**

**def size(self):**

**return len(self.items)**

**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**

**def NumSets(S):**

**count =0**

**for i in range(len(S)):**

**if S[i]<0:**

**count += 1**

**return count**

**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 draw\_dsf(S):**

**scale = 30**

**fig, ax = plt.subplots()**

**for i in range(len(S)):**

**if S[i]<0: # i is a root**

**ax.plot([i\*scale,i\*scale],[0,scale],linewidth=1,color='k')**

**ax.plot([i\*scale-1,i\*scale,i\*scale+1],[scale-2,scale,scale-2],linewidth=1,color='k')**

**else:**

**x = np.linspace(i\*scale,S[i]\*scale)**

**x0 = np.linspace(i\*scale,S[i]\*scale,num=5)**

**diff = np.abs(S[i]-i)**

**if diff == 1: #i and S[i] are neighbors; draw straight line**

**y0 = [0,0,0,0,0]**

**else: #i and S[i] are not neighbors; draw arc**

**y0 = [0,-6\*diff,-8\*diff,-6\*diff,0]**

**f = interpolate.interp1d(x0, y0, kind='cubic')**

**y = f(x)**

**ax.plot(x,y,linewidth=1,color='k')**

**ax.plot([x0[2]+2\*np.sign(i-S[i]),x0[2],x0[2]+2\*np.sign(i-S[i])],[y0[2]-1,y0[2],y0[2]+1],linewidth=1,color='k')**

**ax.text(i\*scale,0, str(i), size=20,ha="center", va="center",**

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

**ax.axis('off')**

**ax.set\_aspect(1.0)**

**def mazeByUnion(S,walls,a):**

**while a!=0:**

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

**c1 = find(S,walls[d][0])**

**c2 = find(S,walls[d][1])**

**if c1 != c2:**

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

**walls.pop(d)**

**a-=1**

**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')**

**def Breadth\_first\_search(G,v):**

**visited = [False]\*(len(G))**

**prev = [-math.inf] \*len(G)**

**queue = []**

**queue.append(v)**

**visited[v] = True**

**while queue:**

**u = queue.pop()**

**for t in G[u]:**

**if not visited[t]:**

**visited[t] = True**

**prev[t] = u**

**queue.append(t)**

**return prev**

**def depthFirst\_stack(G,source):**

**visited = [False] \*(len(G))**

**prev = [-math.inf]\*len(G)**

**visited[source] = True**

**S = Stack()**

**S.push(source)**

**while S:**

**u = S.pop()**

**for t in G[u]:**

**if not visited[t]:**

**visited[t] = True**

**prev[t] = u**

**S.push(t)**

**return prev**

**def depthFirst\_rec(G,source):**

**visited = [False]\*(len(G))**

**prev = [-math.inf] \*len(G)**

**visited[source] = True**

**for t in G[source]:**

**if not visited[t]:**

**prev[t] = source**

**depthFirst\_rec(G,t)**

**def adjList\_fromMaze(S):**

**G = [ [] for i in range(len(S)) ]**

**for j in range(len(S)):**

**if S[j] != -1:**

**G[j].append(S[j])**

**return G**

**plt.close("all")**

**maze\_rows = 11**

**maze\_cols = 15**

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

**draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True)#displaying before creating the maze**

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

**draw\_dsf(S)#displaying before union**

**m = maze\_rows\*maze\_cols**

**print('There are ', m-1 , 'cells')**

**a = int(input('How many walls would you like to remove? '))**

**if a < m-1:**

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

**elif a == m-1:**

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

**else:**

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

**mazeByUnion(S,walls,a)**

**print(S)**

**G = adjList\_fromMaze(S)**

**print(G)**

**start = time.time()**

**p = depthFirst\_rec(G,3)**

**end = time.time()**

**print(end-start)**

**draw\_graph(G)**

**draw\_dsf(S)#to check if there is only one set**

**draw\_maze(walls,maze\_rows,maze\_cols)#displaying the maze**