**Course Name:** 2302 **Author:** Olugbenga Iyiola **ID:** 80638542 **Instructor:** Olac Fuentes **TA:** Nath Anindita/ Malileh Zargaran **LAB #7 Report**

**Introduction**

The purpose of this lab is to first use a disjoint set forest to build a maze which contains a collection of cells separated by walls in such a way that there is exactly one simple path (that is, a path that does not visit any cell more than once) separating any two cells.

We remove one wall at a time (making sure that the cells separated by that wall were not reachable from each other) until the disjoint set forest representing the maze has exactly one tree. If the maze has n cells, your program would remove exactly n − 1 walls to reach this situation

If you remove less than n − 1 walls, some cells will not be reachable from the start cell. If you remove more than n − 1 walls (notice that after removing n − 1 walls, all remaining walls separate cells that are reachable from each other, and thus belong to the same tree in the disjoint set forest), you would have multiple paths from the source to the destination.

We then write a method to build the adjacency list representation of your maze. Cells in the maze should be represented by vertices in the graph. If two cells u and v are contiguous and there is no wall separating them, then there must be an edge from u to v in the graph.

Lastly we solve the maze using Breadth-first search, Depth-first search using a stack and Depth-first search using recursion algorithms.

**Proposed Solution Design and Implementation**

To build the maze, the following procedures are followed;

* Let M be the number of rows and N be the number of columns of the square maze.
* When all walls are present, each of the M ∗ N cells in the maze belongs to a different set.
* Thus you have M ∗ N sets in your disjoint set forest.
* When a wall is removed, if the cells that were separated by that wall belong to different sets, these sets are united.
* This process is repeated until all cells belong to a single set.
* At that point the maze is displayed.

The following pseudocode illustrates the process to build the maze:

*Create full maze with all adjacent cells are separated by a wall*

*Assign each cell to a different set in a disjoint set forest S*

*if number of cells to be Removed < (Total Cells - 1):*

*while Number of Sets > (Total Cells-number of cells to be removed)*

*Select a random wall w =[c1,c2]*

*If cells c1 and c2 belong to different sets, remove w and join c1’s set and c2’s set*

*otherwise do nothing*

*else if number of cells to be removed == (Total Cells - 1):*

*while Number of Sets(S) > 1:*

*Select a random wall w =[c1,c2]*

*If cells c1 and c2 belong to different sets, remove w and join c1’s set and c2’s set*

*otherwise do nothing*

*else if number of cells to be removed > (Total Cells - 1):*

*counter = 0*

*while counter < (number of cells to be removed)*

*Select a random wall w =[c1,c2]*

*If cells c1 and c2 belong to different sets, remove w and join c1’s set and c2’s set*

*otherwise do nothing*

*Display maze*

**Breadth First Search**

*let Q be queue.*

*Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.*

*mark s as visited.*

*while ( Q is not empty)*

*//Removing that vertex from queue,whose neighbour will be visited now*

*v = Q.dequeue( )*

*//processing all the neighbours of v*

*for all neighbours w of v in Graph G*

*if w is not visited*

*Q.enqueue( w ) //Stores w in Q to further visit its neighbour*

*mark w as visited.*

**Depth First**

*source vertex*

*let S be stack*

*S.push( s ) //Inserting s in stack*

*mark s as visited.*

*while ( S is not empty):*

*//Pop a vertex from stack to visit next*

*v = S.top( )*

*S.pop( )*

*//Push all the neighbours of v in stack that are not visited*

*for all neighbours w of v in Graph G:*

*if w is not visited :*

*S.push( w )*

*mark w as visited*

**Depth First with Recursion**

*DFS-recursive(G, s):*

*mark s as visited*

*for all neighbours w of s in Graph G:*

*if w is not visited:*

*DFS-recursive(G, w)*

**Experimental Result**

System Specification: HP Windows 10, 1.60GHZ Intel® Celeron® , 4.GB RAM, 64-bit operating system

The results of the various test cases using different sizes from the file for each of the algorithms are shown below:

**Creating Maze Using Standard Union**

|  |  |
| --- | --- |
| **Number of Cells(Rows \* Cols)** | **Runtime in nanoseconds** |
| **100** | **19071330** |
| **225** | **27172070** |
| **400** | **529857596** |
| **600** | **719342699** |
| **1000** | **2881071394** |
|  |  |

O(n)

**Maze to Adjacency List**

|  |  |
| --- | --- |
| **Number of Cells(Input)** | **Runtime in nanoseconds** |
| **100** | **3254432462** |
| **225** | **3992503440** |
| **400** | **5438894045** |
| **600** | **7472084226** |
| **1000** | **15484318858** |
|  |  |

O(n)

**Breadth First Search**

|  |  |
| --- | --- |
| **Number of Cells(Input)** | **Runtime in nanoseconds** |
| **100** | **1077794** |
| **225** | **926109** |
| **400** | **1095715** |
| **600** | **1912382** |
| **1000** | **2868572** |
|  |  |

**O(|V| + |E|)**

**Depth First Search**

|  |  |
| --- | --- |
| **Number of Cells(Input)** | **Runtime in nanoseconds** |
| **100** | **858267** |
| **225** | **693782** |
| **400** | **947870** |
| **600** | **322571** |
| **1000** | **2139588** |
|  |  |

**O(|V| + |E|)**

**Depth First Search with Recursion**

|  |  |
| --- | --- |
| **Number of Cells(Input)** | **Runtime in nanoseconds** |
| **100** | **37762** |
| **225** | **127364** |
| **400** | **27521** |
| **600** | **8321** |
| **1000** | **62082** |
|  |  |

**O(|V| + |E|)**

**CONCLUSION**

In summary a graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. Adjacency list is a representation of a [directed graph](https://xlinux.nist.gov/dads/HTML/directedGraph.html) with n [vertices](https://xlinux.nist.gov/dads/HTML/vertex.html)using an [array](https://xlinux.nist.gov/dads/HTML/array.html)*of n*[lists](https://xlinux.nist.gov/dads/HTML/list.html) of vertices. List i contains vertex j if there is an [edge](https://xlinux.nist.gov/dads/HTML/edge.html) from vertex i to vertex j.

Breadth First Search(BFS) visit nodes **level by level** in Graph and uses Queue data structure to store un-explored nodes but Depth First Search(DFS) visit nodes of graph**depth wise and** uses Stack data structure to store Un-explored nodes. BFS is slower and require more memory while DFS is faster and require less memory.

**Appendix**

***Programmed by Olac Fuentes***

# Implementation of disjoint set forest

# Programmed by Olac Fuentes

# Last modified March 28, 2019

import matplotlib.pyplot as plt

import numpy as np

from scipy import interpolate

def DisjointSetForest(size):

return np.zeros(size,dtype=np.int)-1

def dsfToSetList(S):

#Returns aa list containing the sets encoded in S

sets = [ [] for i in range(len(S)) ]

for i in range(len(S)):

sets[find(S,i)].append(i)

sets = [x for x in sets if x != []]

return sets

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\_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):

# 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 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)

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

plt.close("all")

S = DisjointSetForest(8)

print(S)

draw\_dsf(S)

union(S,7,6)

print(S)

draw\_dsf(S)

union(S,0,2)

print(S)

draw\_dsf(S)

union(S,6,3)

print(S)

draw\_dsf(S)

union(S,5,2)

print(S)

draw\_dsf(S)

union(S,4,6)

print(S)

draw\_dsf(S)

print('Sets encoded by DSF:',dsfToSetList(S))

T = DisjointSetForest(8)

union(T, 7 , 0 )

union(T, 1 , 6 )

union(T, 3 , 0 )

union(T, 0 , 6 )

union(T, 3 , 4 )

union(T, 2 , 5 )

union(T, 6 , 0 )

union(T, 0 , 3 )

union(T, 4 , 2 )

union(T, 1 , 7 )

print(T)

draw\_dsf(T)

print('Sets encoded by DSF:',dsfToSetList(T))

U = DisjointSetForest(8)

for i in range(len(U)):

union(U, i , 0 )

print(U)

draw\_dsf(U)

Uc = DisjointSetForest(8)

for i in range(len(Uc)):

union\_c(Uc, i , 0 )

print(Uc)

draw\_dsf(Uc)

Us = DisjointSetForest(8)

for i in range(len(Us)):

union\_by\_size(Us, i , 0 )

print(Us)

draw\_dsf(Us)

**Wikipedia**

[**https://en.wikipedia.org/wiki/Sorting\_algorithm#Comparison\_of\_algorithms**](https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms)

**Academic Dishonesty**

This work was done by me without any act or practice of academic dishonesty

**SIGNATURE**

**OLUGBENGA IYIOLA(OT)**

**……………………………………………….**