**Lab 6**

In this lab the code is made to create a random labyrinth by making use of the disjointed set forests’ property that an element can be in only one set at a time. To implement this, the union() and union\_by\_size() methods were modified by adding a return Boolean condition if the union was true or not, the method one\_Forest() checks if the elements in the forest are in more than one set and returns either true or false depending on the forest.

**Experimental Results**

|  |  |  |
| --- | --- | --- |
| Inputs | Time | Output |
| 150 cells | Regular Union: 0.01596 seconds |  |
| Union by Size: 0.02099 seconds |  |
| 300 cells | Regular Union: 0.06481 seconds |  |
| Union by Size: 0.06782 seconds |  |
| 600 cells | Regular Union: 0.27328 seconds |  |
| Union by Size: 0.28923 seconds |  |

**Time Complexity**

Union() = O()

Union\_by\_size() = O()

One\_Forest() = O()

**Conclusion**

Although Union\_by\_size will be more convenient for prolonged or continual access to information, in this case it only adds an unnecessary step to the building function when compared to regular union, thus making regular union a more appropriate option for building labyrinth.

**Source Code**

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

"""

Created on Fri Apr 5 10:09:45 2019

CS 2302 - Data Structures

Instructor:Olac Fuentes

Lab 6, Disjoint Set Forests

Creates a random labyrinth by making use of the Forest's characteristics

TA:Anindita Nath

@author: Hugo Chavez

"""

import time

# Starting point for program to build and draw a maze

# Modify program using disjoint set forest to ensure there is exactly one

# simple path joiniung any two cells

# Programmed by Olac Fuentes

# Last modified March 28, 2019

import matplotlib.pyplot as plt

import numpy as np

import random

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 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 union(S,i,j):

# Joins i's tree and j's tree, if they are different

# returns a True value if the roots 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

return True

return False

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\_c(S,i,j):

# Joins i's tree and j's tree, if they are different

# Uses path compression

# returns a True value if the roots are different

ri = find\_c(S,i)

rj = find\_c(S,j)

if ri!=rj:

S[rj] = ri

return True

return False

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

# returns a True value if the roots are different

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

return True

return False

def one\_Forest(S):

#checks if the Forest has more than one root

count = 0

for i in range(len(S)):

if S[i] < 0:

count +=1

if count == 1:

return True

return False

#--------------------------------------------------------

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

plt.close("all")

maze\_rows = 10

maze\_cols = 15

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

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

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

start = time.time()

# Builds maze with standard union

while not one\_Forest(forest): #!= 1:

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

if union(forest, walls[d][0],walls[d][1]):

print('removing wall ',walls[d])

walls.pop(d)

'''

# Builds maze with union by size compression

while not one\_Forest(forest): #!= 1:

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

if union\_by\_size(forest, walls[d][0],walls[d][1]):

print('removing wall ',walls[d])

walls.pop(d)

'''

end = time.time()

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

print('time:', round((end-start),5))

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 provided inappropriate assistance to any student in the class.

\_\_\_\_\_\_\_\_\_\_\_\_Hugo Chavez\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_