Laila Romero

CS 2302 Data Structures

April 16, 2019

Lab 6

* Introduction

For this lab we were trying to create a randomized maze through mathplotlib and a disjoint set forest. We needed to make sure that all the cells were in the same set/pointed to the same root every time it was randomized. We also needed to compare the run times for randomizing the mazes with union by compression and regular union.

• Proposed Solution Design and Implementation

To get make sure the cells of the maze were always in the same set, I made a disjoint set forest measuring the size of the maze columns \* maze rows. Then I made a method to count the number of sets in the disjoint set forest called NumSets which took the disjoint set forest as a parameter. It then kept a counter called num which increased by 1 every time the index of a cell was less than -1 (indicating it belonged to a different set) as it traversed through the forest. At the end it returned the number of sets in the forest.

Then I made a method called MazeNorm which took the number of rows, columns, the disjoint set forest, and the wall list are parameters. It then checked for the number of sets in the disjoint set forest, while it is greater than one set it creates a randomized integer called d. It then checks if the roots of the cells are equal to each other using the find method in the dsf code. If the roots are not, then it uses the union method from the dsf code to give the cells the same root. It then randomly pops a wall between the two cells.

I implemented this same method excepted using union by size and the find compression method provided in the dsf code. I made a method called MazeComp which took the number of rows, columns, the disjoint set forest, and the wall list are parameters. It then checked for the number of sets in the disjoint set forest, while it is greater than one set it creates a randomized integer called d. It then checks if the roots of the cells are equal to each other using the find\_c method in the dsf code. If the roots are not, then it uses the union\_by\_size method from the dsf code to give the cells the same root. It then randomly pops a wall between the two cells.

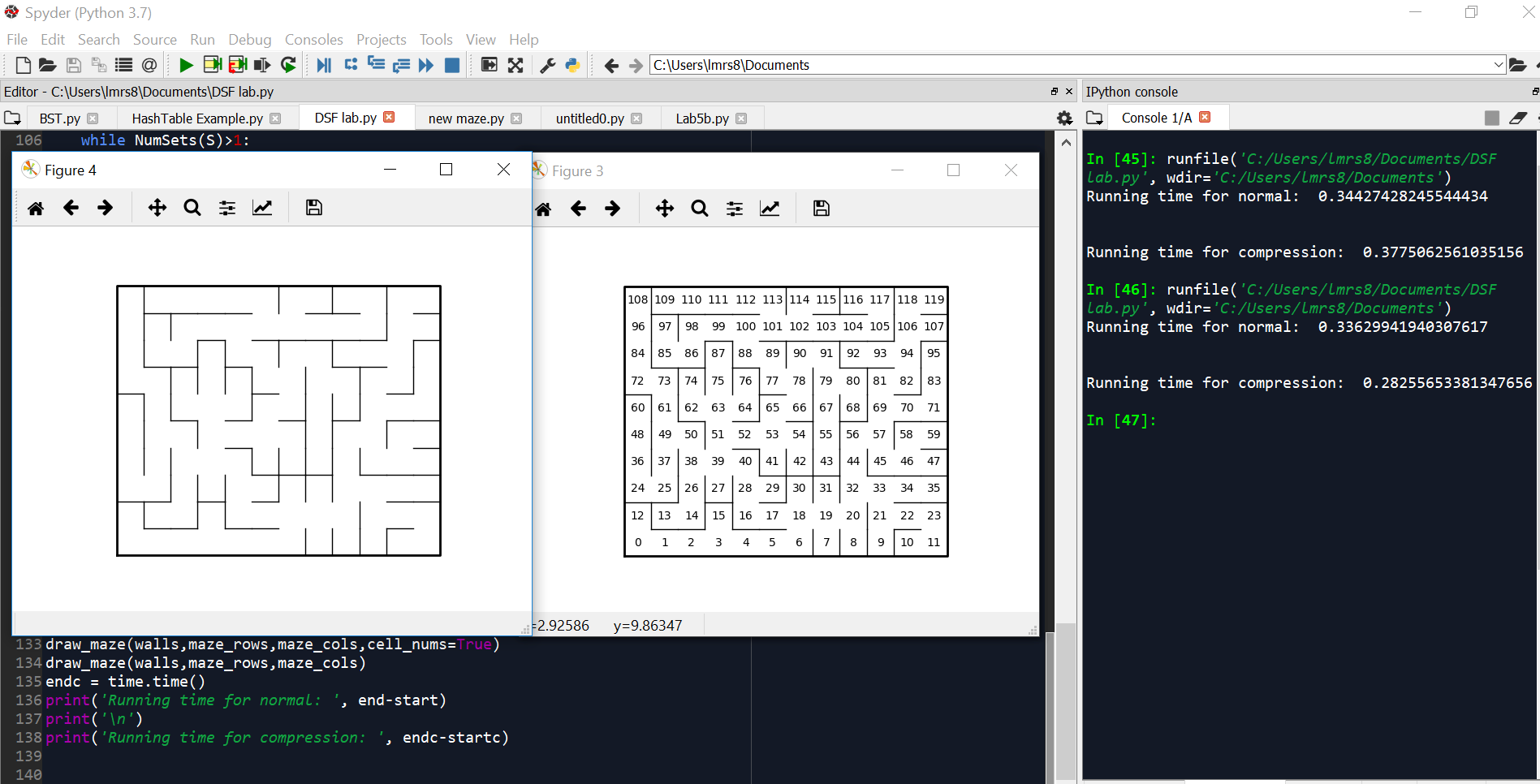
• Experimental results

In order to test my program, I tried different sizes of mazes to make sure the method worked under all circumstances. I also compared the running times to make the same randomized maze with the MazeNorm and MazeComp and found out the MazeComp method was generally faster to produce the maze. The average times for each method to produce a 10 x 12 maze were:

Running time for normal: 0.33629941940307617

Running time for compression: 0.28255653381347656

|  |  |
| --- | --- |
| **Method** | **Big O** |
| MazeNorm | O(n) |
| MazeComp | O(n) |
| NumSets | O(n) |



• Conclusion

In conclusion, I learned how to create a maze with both union by size and union. I learned how to implement disjoint set forests and to make all the cells point to one root even when randomized. I also learned that the compression method ran faster than the regular method to create a maze.

• Appendix – Source codes

import matplotlib.pyplot as plt

import numpy as np

import random

import time

from random import shuffle, randrange

def DisjointSetForest(size):

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

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 walk(x, y):

w[sy][sx] = 1

d = [(x - 1, y), (x, y + 1), (x + 1, y), (x, y - 1)]

shuffle(d)

walk(x0, y0)

walk(randrange(sx), randrange(sy))

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 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\_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 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

ri = find(S,i)

rj = find(S,j)

if ri!=rj:

S[rj] = ri

def NumSets(S):

num =0

for i in range(len(S)):

if S[i] <0:

num+=1

return num

def MazeNorm(rows,cols,S,walls):

while NumSets(S)>1:

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

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

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

walls.pop(d)

def MazeComp(rows,cols,S,walls):

while NumSets(S)>1:

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

if find\_c(S,walls[d][0])!= find\_c(S,walls[d][1]):

union\_by\_size(S,walls[d][0],walls[d][1])

walls.pop(d)

plt.close("all")

maze\_rows = 10

maze\_cols = 12

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

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

start = time.time()

MazeNorm(maze\_rows, maze\_cols,S,walls)

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

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

end = time.time()

startc = time.time()

MazeComp(maze\_rows, maze\_cols, S, walls)

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

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

endc = time.time()

print('Running time for normal: ', end-start)

print('\n')

print('Running time for compression: ', endc-startc)

“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.”

