**Emmanuel Alvarez**

**80567137**

**Introduction**

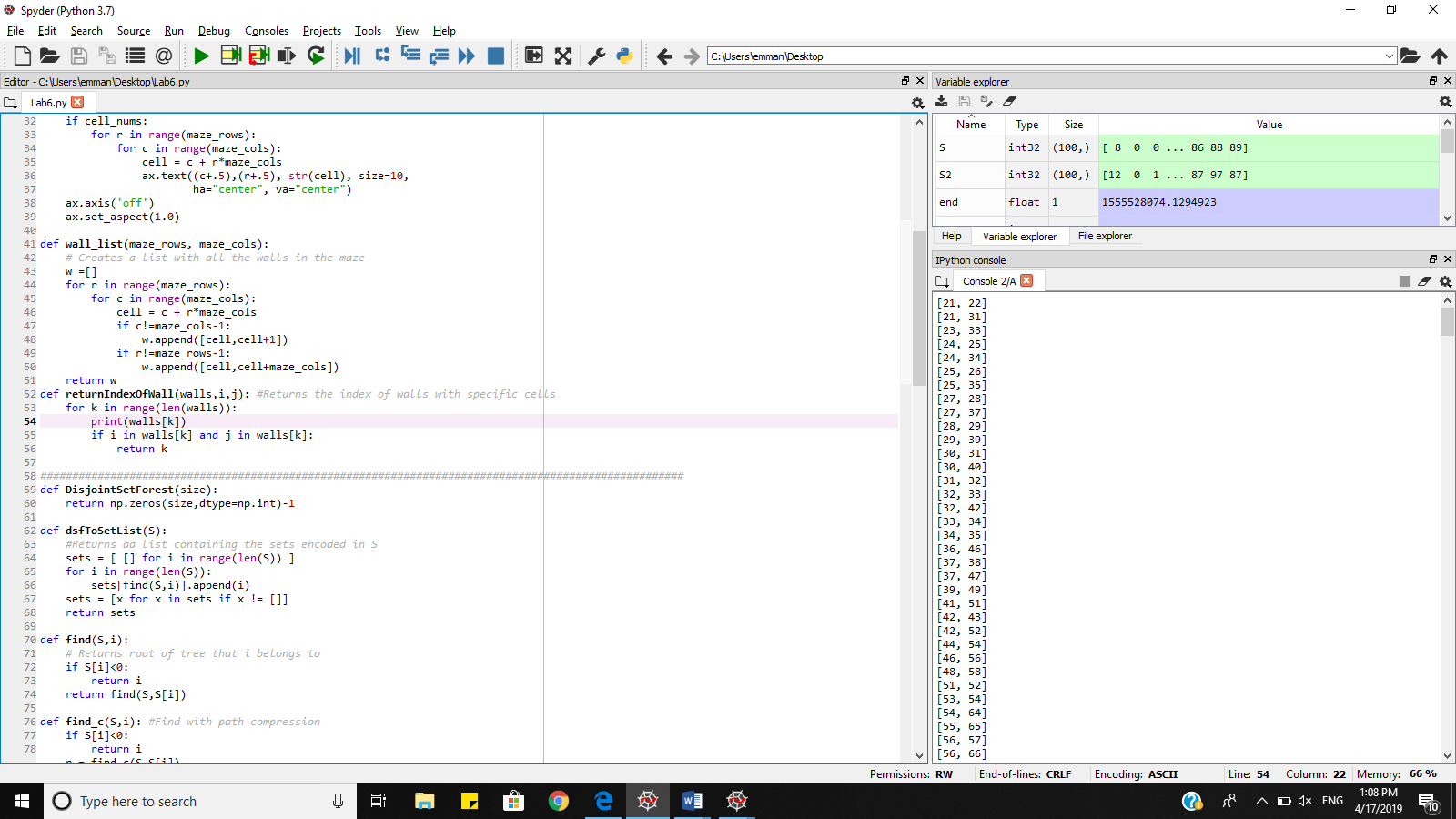
The purpose of this lab is to create a maze using a disjoint set forest structure. The goal is to separate the cells with walls and there must be a unique path between any two cells. In order to achieve it, I used two different methods, the standard union and the union by size with path compression.

**Proposed solution and design implementation**

For standard union method, I modified the method *union*() and I created another two methods called *returnIndexOfWall*, that returns the index of the array *walls* that we want to remove, and the method *OneSetInForest*, it returns True if the disjoint set forest has only one set. Thus, when I call the method *CreatePathByUnion,* it will be executed until the disjoint set forest has only one set, in each iteration I created a random number that will choose a random wall to be removed, then I called the method *union* to join the sets if they are different and in this method I remove the index of the array *walls.* When the while loop stops I just returned the array *walls* that will be used to draw the maze.

For the second method, called *CreatePathByUnionSize*, I basically did the same steps, but this time instead of calling the method *union,* I called the method *union\_by\_size,* where I added the line code to remove the index desired from the array *walls*.

**Experimental results**

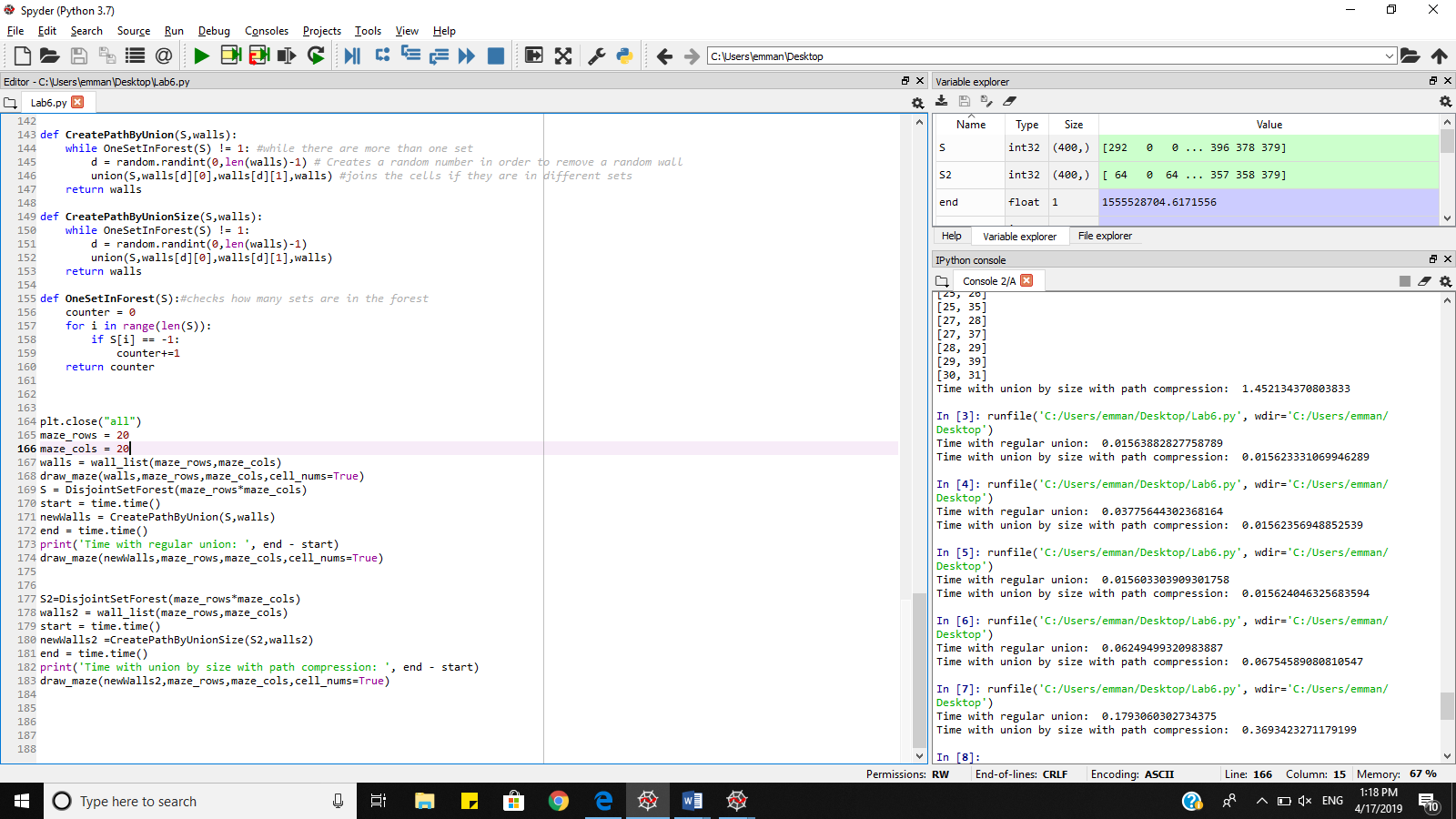
What I did was to print the array walls to get an idea of how it was structured and the results were as shown below. 

**Big O notation**

Both methods, standard union and union by size with compression has a O(log n \* n) because they will finish until there is no more than one set (log n ), and they are calling the method OneSetInForest that has a O(n).

**Conclusion**

I made many different comparisons with different maze sizes to know which method was faster and these where the results.



As we can see, sometimes standard method is faster than union by size with compression, but most of the time the second method is faster.

Appendix

"""

Author: Emmanuel Alvarez

Professor: Olac Fuentes

Last day of modification: 04-14-2019

The purpose of this program is to practice disjoint set forest by creating a maze with a unique

path among any two cells

"""

import matplotlib.pyplot as plt

import numpy as np

import random

from scipy import interpolate

import time

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 returnIndexOfWall(walls,i,j): #Returns the index of walls with specific cells

for k in range(len(walls)):

#print(walls[k])

if i in walls[k] and j in walls[k]:

return k

#####################################################################################################

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,walls):

# 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

index = returnIndexOfWall(walls,i,j) #gets the index of cells

walls.pop(index) #removes the wall of specific cells

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

index = returnIndexOfWall(walls,i,j) #gets the index of the cells

walls.pop(index)# removes the wall of those cells

else:

S[ri] += S[rj]

S[rj] = ri

index = returnIndexOfWall(walls,i,j)

walls.pop(index)

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 CreatePathByUnion(S,walls):

while OneSetInForest(S) != 1: #while there are more than one set

d = random.randint(0,len(walls)-1) # Creates a random number in order to remove a random wall

union(S,walls[d][0],walls[d][1],walls) #joins the cells if they are in different sets

return walls

def CreatePathByUnionSize(S,walls):

while OneSetInForest(S) != 1:

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

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

return walls

def OneSetInForest(S):#checks how many sets are in the forest

counter = 0

for i in range(len(S)):

if S[i] == -1:

counter+=1

return counter

plt.close("all")

maze\_rows = 20

maze\_cols = 20

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

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

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

start = time.time()

newWalls = CreatePathByUnion(S,walls)

end = time.time()

print('Time with regular union: ', end - start)

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

S2=DisjointSetForest(maze\_rows\*maze\_cols)

walls2 = wall\_list(maze\_rows,maze\_cols)

start = time.time()

newWalls2 =CreatePathByUnionSize(S2,walls2)

end = time.time()

print('Time with union by size with path compression: ', end - start)

draw\_maze(newWalls2,maze\_rows,maze\_cols,cell\_nums=True)

“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.”