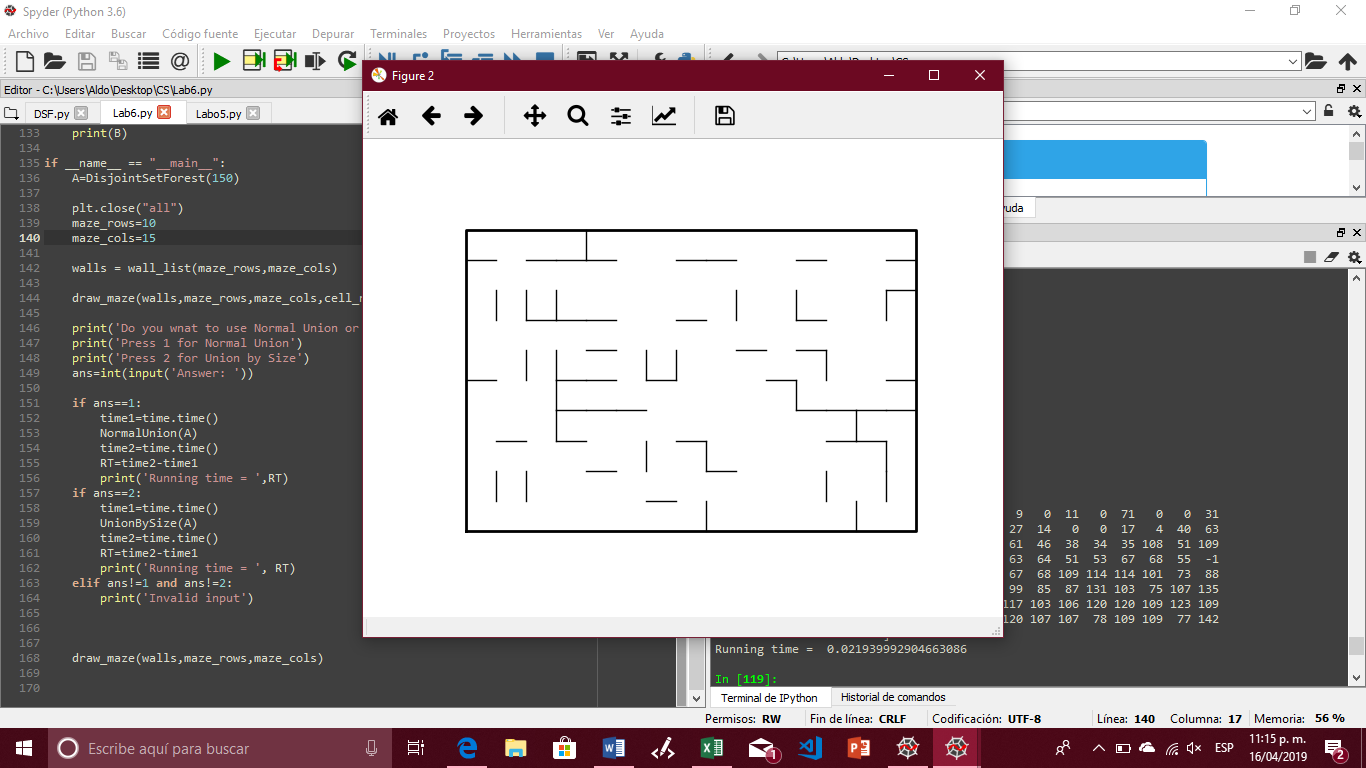
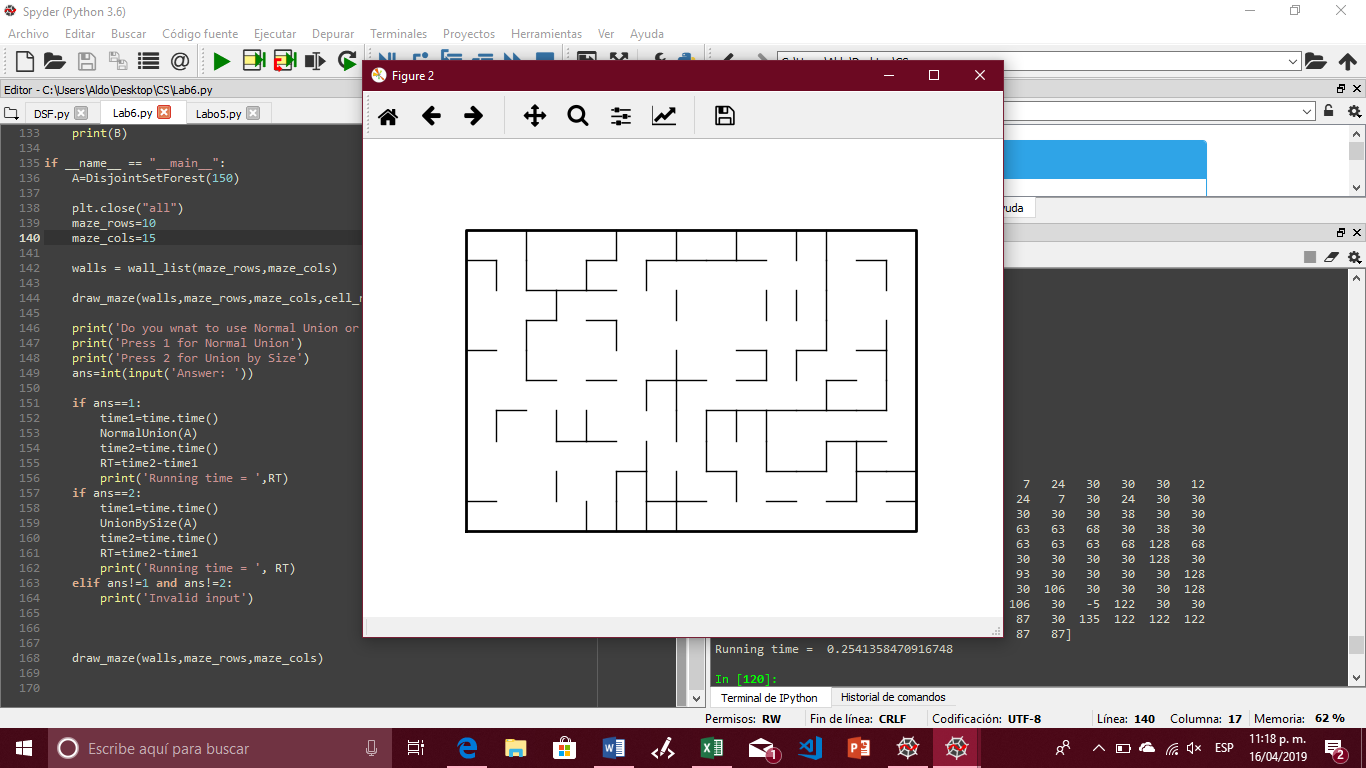
For this lab we were supposed to create a maze with a specific size based on a disjoint set forest. After that, we needed to randomly remove walls from the maze, but just of the elements separated for the wall were in different sets, it would just continue otherwise, repeating the process until the number of sets in the maze equals to one. This process must be executed with two different types of union, normal union and union by sizes, the user is going to decide which one to implement, and after being executed it will print the time that it takes to complete the process.

In this case, most of the functions needed for this lab were already provided, so I just needed to create two functions that are almost the same but calling the two different functions. In these functions I have a loop that will keep running while the number of sets in the maze is bigger than one, then a wall will be selected randomly and it will go to the condition to make sure if the elements separated for that wall belong to the same set, if they do the wall stays, the wall will be eliminated otherwise. At the end the maze will be printed with the final result.

**Normal Union: Union by Size:**

Maze size=150 Maze size=150

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **Maze Size** | | | | | | |
| **Union Type** | | **10** | **50** | **100** | **150** | **500** | **1000** | **10000** |
| Normal Union | | 0.007 | 0.00504 | 0.01276 | 0.03106 | 0.25708 | 0.84834 | 50.6404 |
| Union by Sizes | | 0.00084 | 0.0206 | 0.01382 | 0.02072 | 0.22018 | 0.68908 | 47.8815 |

After completing this lab a have a better sense of how a disjoint set forest works, and how it can be represented with a maze.

import matplotlib.pyplot as plt

import numpy as np

from scipy import interpolate

import random

import time

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 NumSets(S):

if S is None:

return 0

sum=0

for i in S:

if i==-1:

sum+=1

return sum

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

#Methot to remove walls using normal union

def NormalUnion(A):

while NumSets(A)!=1:

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

if union(A,walls[d][0],walls[d][1])!=False:

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

walls.pop(d)

print(A)

#Method to remove walls using Union by Size

def UnionBySize(B):

while NumSets(B)!=1:

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

if union\_by\_size(B,walls[d][0],walls[d][1])!=False:

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

walls.pop(d)

print(B)

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

A=DisjointSetForest(10000)

plt.close("all")

maze\_rows=100

maze\_cols=100

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

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

print('Do you wnat to use Normal Union or Union by Size?')

print('Press 1 for Normal Union')

print('Press 2 for Union by Size')

ans=int(input('Answer: '))

if ans==1:

time1=time.time()

NormalUnion(A)

time2=time.time()

RT=time2-time1

print('Running time = ',RT)

if ans==2:

time1=time.time()

UnionBySize(A)

time2=time.time()

RT=time2-time1

print('Running time = ', RT)

elif ans!=1 and ans!=2:

print('Invalid input')

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

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.

Aldo A. Venzor