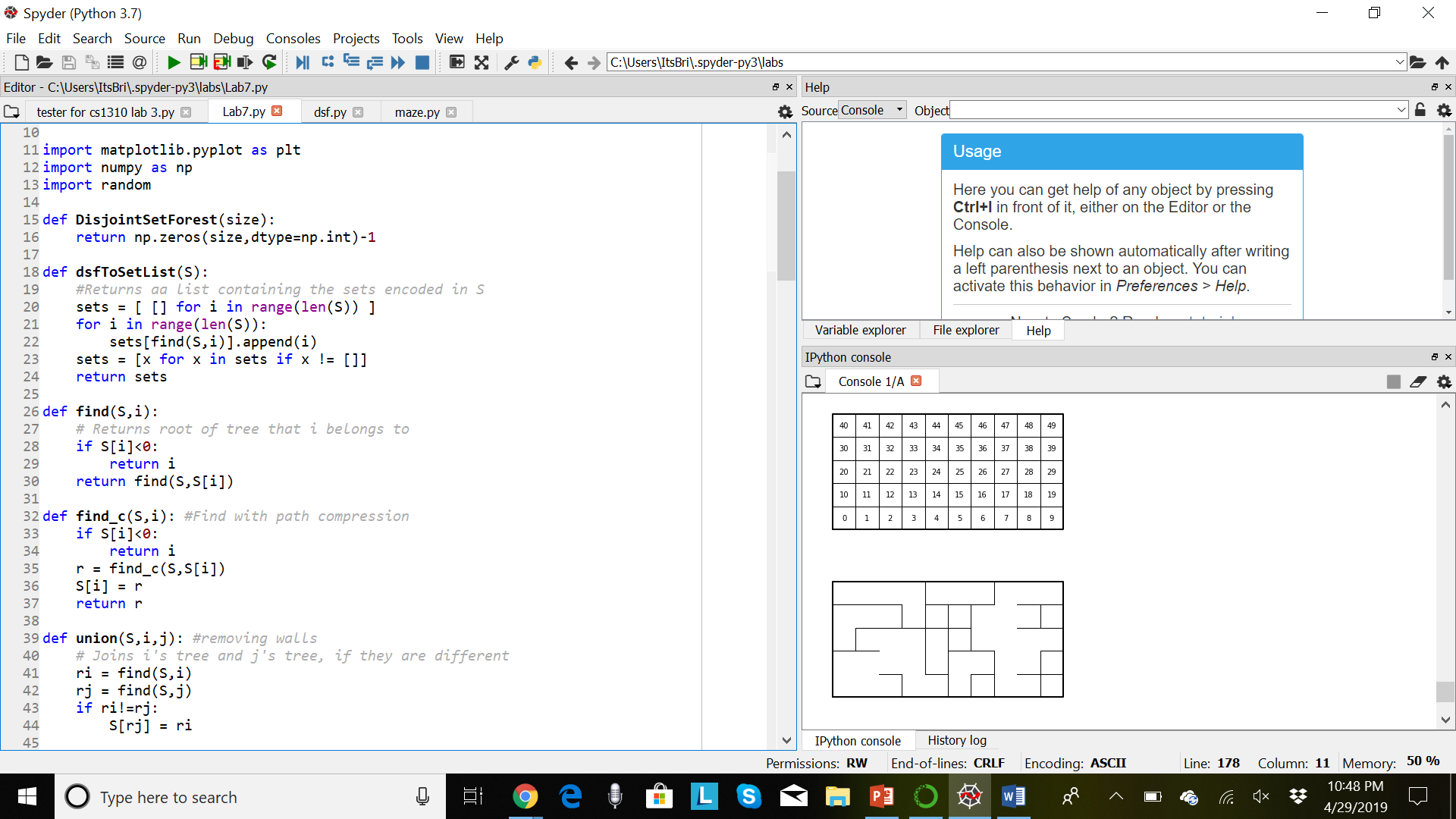
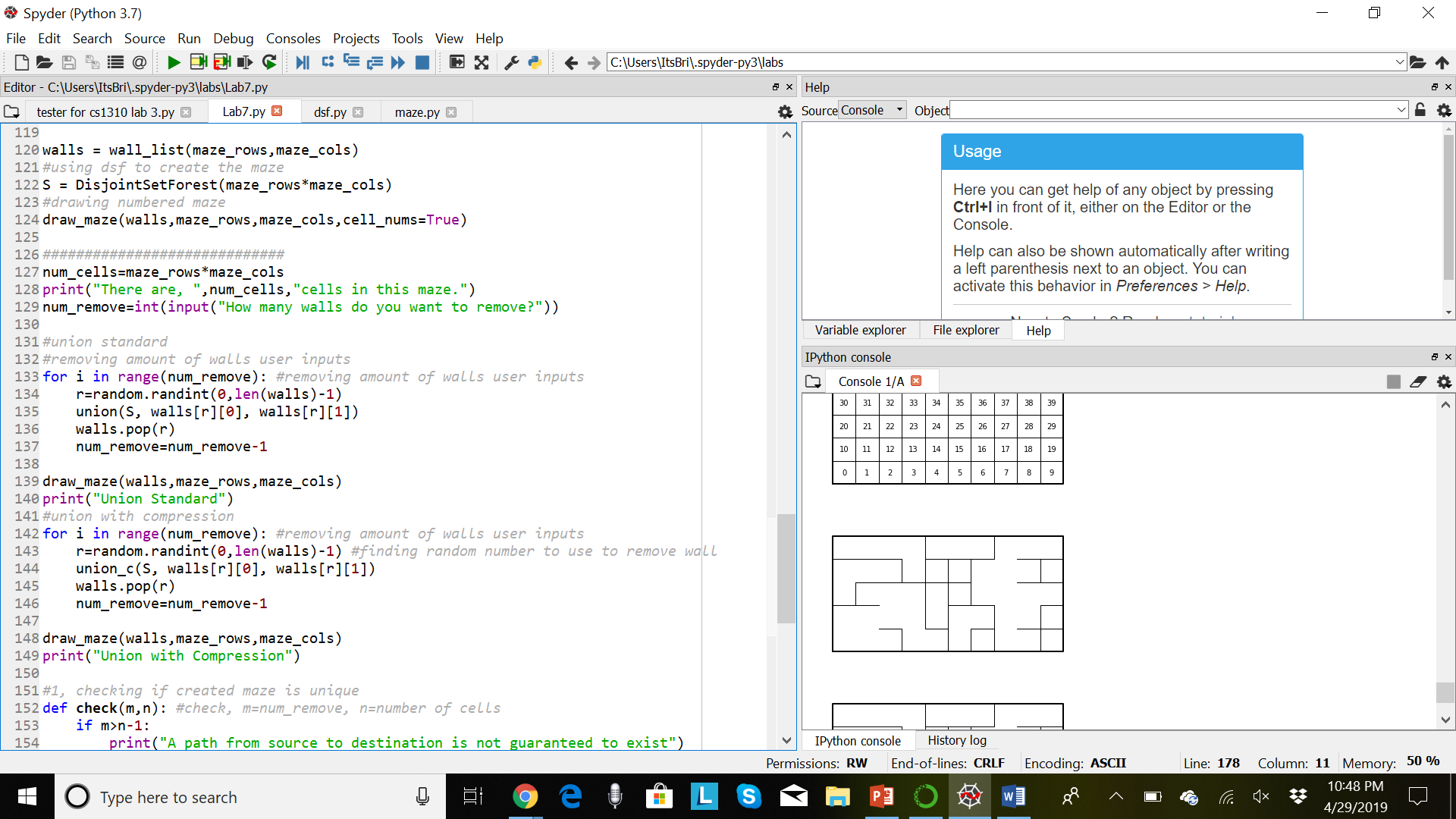
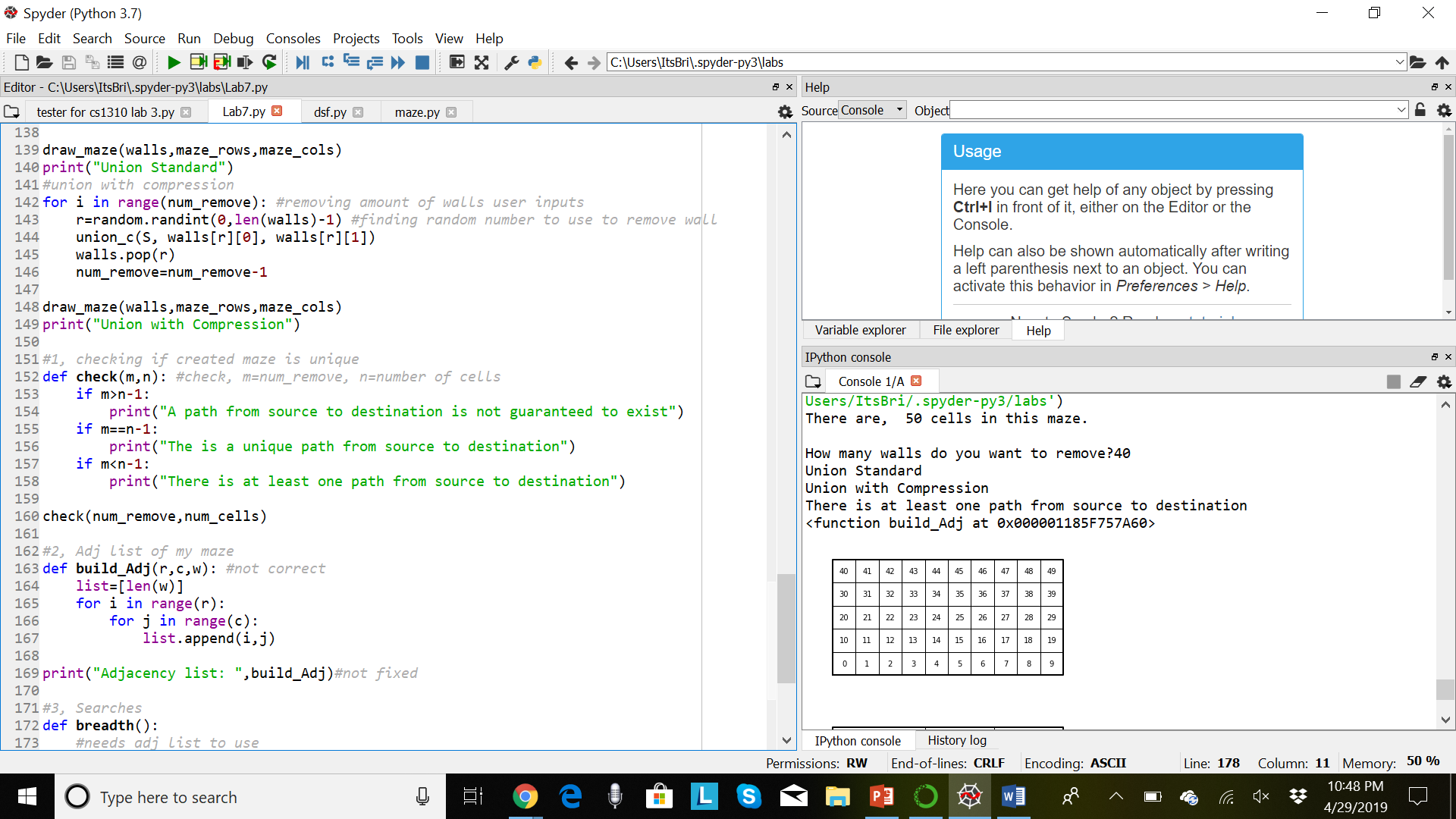
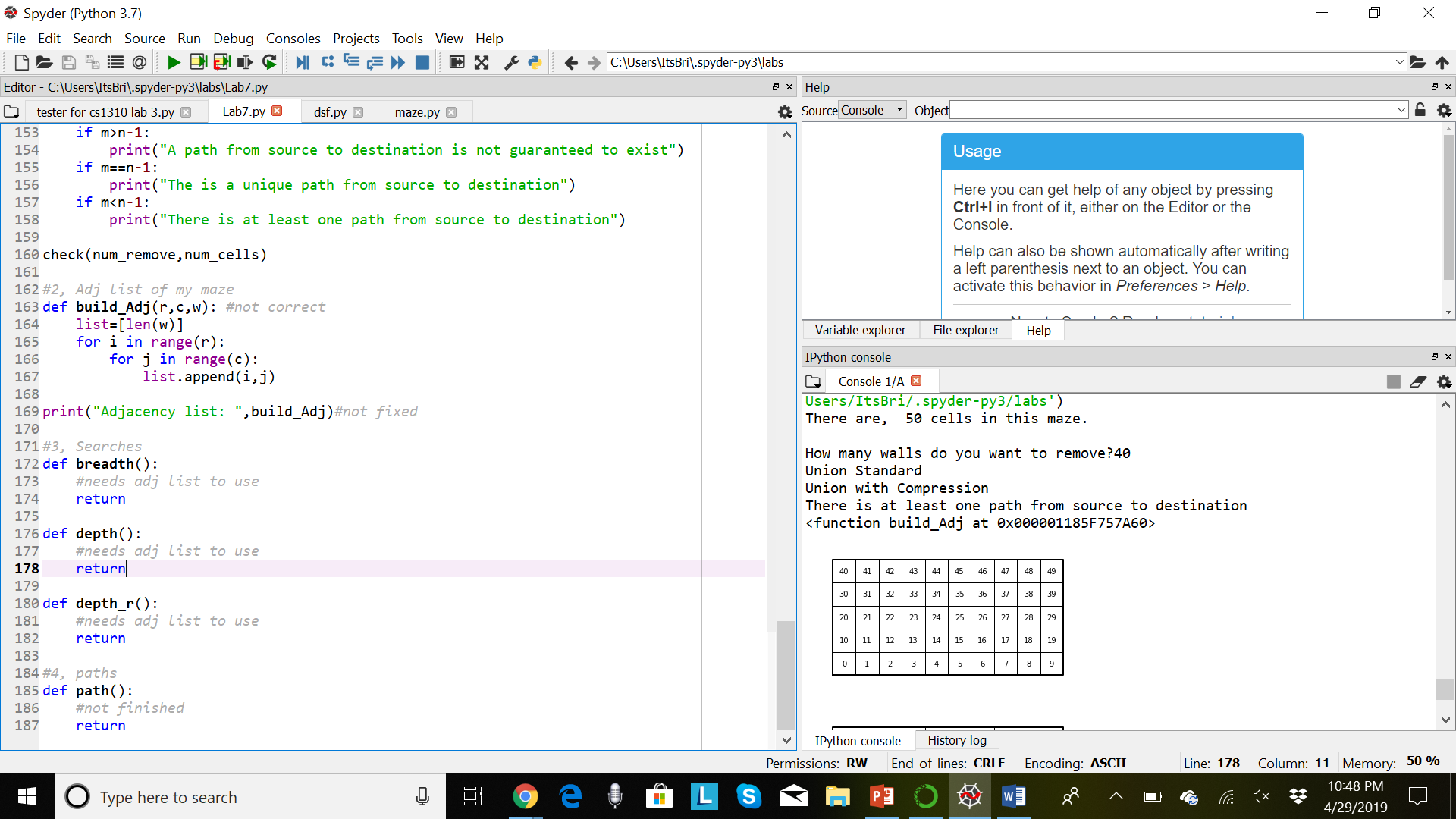
**Lab 7 report**

For this lab, understanding how to create a maze was much easier to comprehend. When starting my code, I first added the functions and drawing functions Dr. Fuentes gave to us. Using this, I re-started my lab 6 for creating the maze. This time, the maze we needed to create was to be implemented by the user’s input. Although from the past lab I tried to understand, I didn’t complete it. This lab, on the other hand, I was able to create the maze from the user’s input of removing the number of walls they wanted. First off, I wrote down on paper how creating the maze would work. From then on, I used trial and error, testing which line of code I switched values on wrongly, or even what would print of the maze. It varied from either moving the code up or down, after this line or the next. After a good amount of time figuring how this maze would be created, I finally ended up with my current code as how it stands, and it works! Onto the second part, I started on task 1 for the lab. Task 1 called for running through the made maze and confirming if the maze is unique, not, or having at least one path from source to destination. For this, I didn’t need to write this part down. It was already written down in the task itself. All I needed to do was to change the wording into code. That function took a lot less time that I had already spent on the first set-up I created for user input of the maze. Next, I started on task 2, building an adjacency list from the maze my code created. I knew this list needed two for loops to track which part of the list has what vertex, so I started with that. (for I in range(r), for j in range(c)) The line before that, I created the list and what size it would be, taking the length of the w list of walls. Next, I figured the code needed to append the vertex at the cross of r=rows and c=columns. When trying to print this out, I got a “<>” message instead of what I needed to be printed out. I switched around my lines, trying a different approach to the code, but couldn’t finish in time. External family events, my job, and other classes have been at my tail this semester, and I still continue to do my best to try and finish or give what I can. The only part I have of task 3 and 4 are setting up the functions but not finishing them to print anything.



No running times could be created since I didn’t finish.

Honesty Statement:

Academic dishonesty includes but is not limited to cheating, plagiarism and collusion. Cheating may involve

copying from or providing information to another student, possessing unauthorized materials during a test, or

falsifying data (for example program outputs) in laboratory reports. Plagiarism occurs when someone

represents the work or ideas of another person as his/her own. Collusion involves collaborating with another

person to commit an academically dishonest act. Professors are required to - and will - report academic

dishonesty and any other violation of the Standards of Conduct to the Dean of Students.

I hereby state hereby this code is mine and mine alone.

**Appendix:**

import matplotlib.pyplot as plt

import numpy as np

import random

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): #removing 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

def union\_c(S,i,j): #removing walls

# 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): #removing walls

# 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):

count =0

for i in range(len(S)):

if S[i]<0:

count += 1

return count

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

#displaying the maze

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

plt.close("all")

maze\_rows = 5

maze\_cols = 10

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

#using dsf to create the maze

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

#drawing numbered maze

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

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

num\_cells=maze\_rows\*maze\_cols

print("There are, ",num\_cells,"cells in this maze.")

num\_remove=int(input("How many walls do you want to remove?"))

#union standard

#removing amount of walls user inputs

for i in range(num\_remove): #removing amount of walls user inputs

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

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

walls.pop(r)

num\_remove=num\_remove-1

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

print("Union Standard")

#union with compression

for i in range(num\_remove): #removing amount of walls user inputs

r=random.randint(0,len(walls)-1) #finding random number to use to remove wall

union\_c(S, walls[r][0], walls[r][1])

walls.pop(r)

num\_remove=num\_remove-1

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

print("Union with Compression")

#1, checking if created maze is unique

def check(m,n): #check, m=num\_remove, n=number of cells

if m>n-1:

print("A path from source to destination is not guaranteed to exist")

if m==n-1:

print("The is a unique path from source to destination")

if m<n-1:

print("There is at least one path from source to destination")

check(num\_remove,num\_cells)

#2, Adj list of my maze

def build\_Adj(r,c,w): #not correct

list=[len(w)]

for i in range(r):

for j in range(c):

list.append(i,j)

print("Adjacency list: ",build\_Adj)#not fixed

#3, Searches

def breadth():

#needs adj list to use

return

def depth():

#needs adj list to use

return

def depth\_r():

#needs adj list to use

return

#4, paths

def path():

#not finished

return