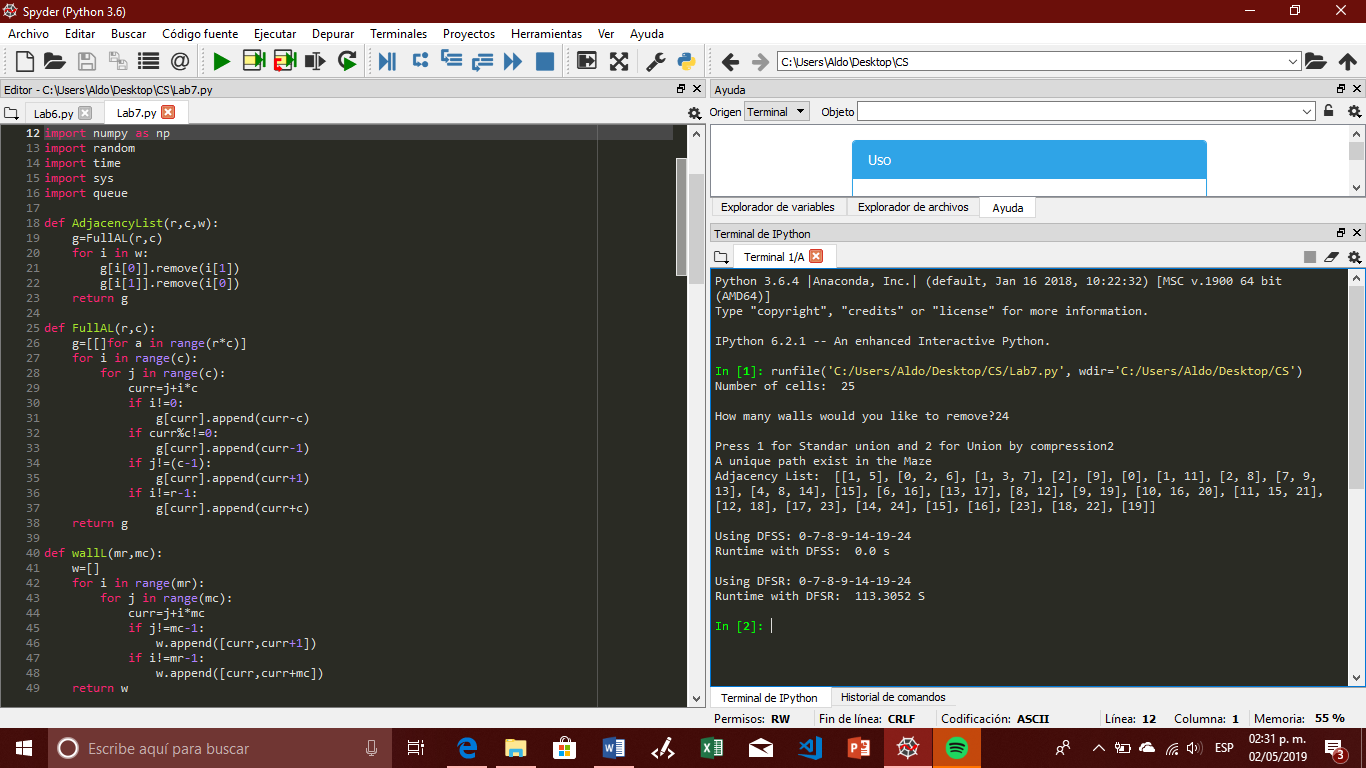
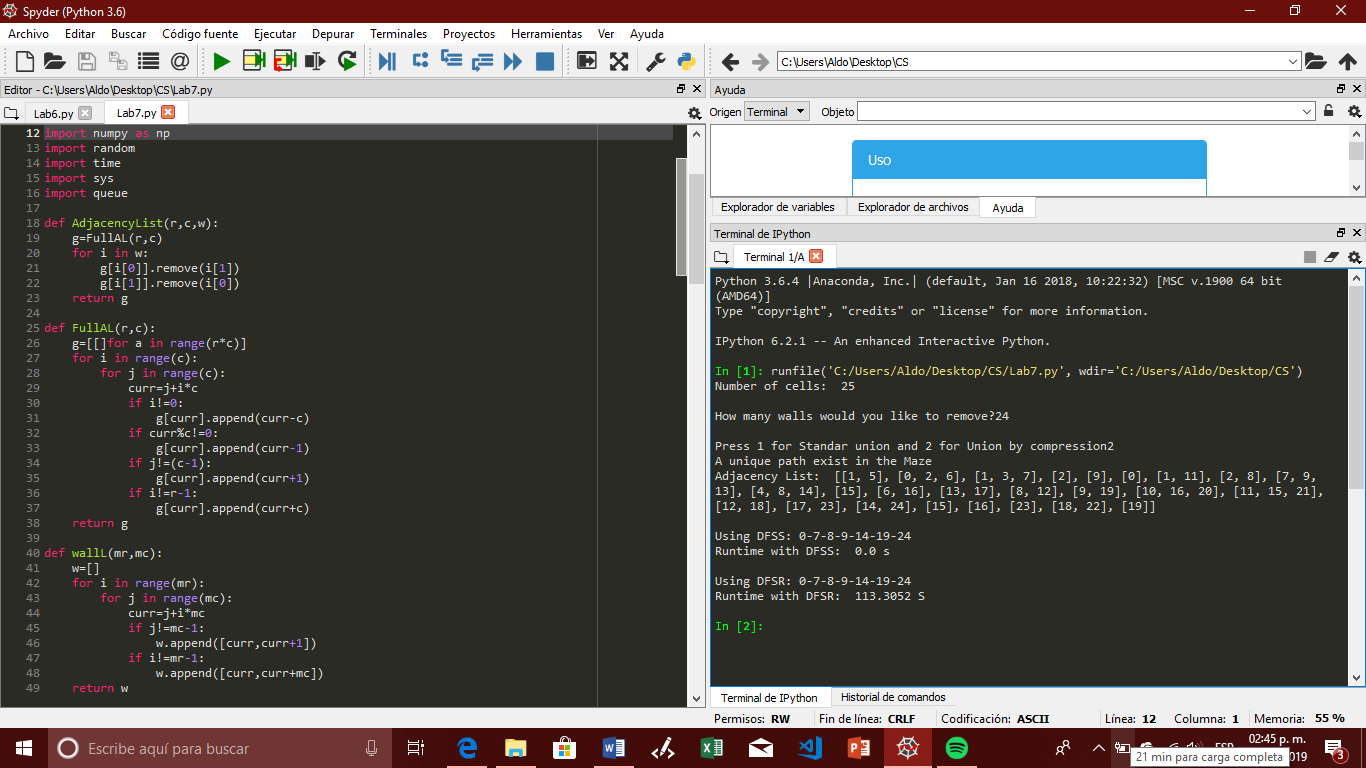
For this lab we were supposed to modify our code used for lab 6 to work with a new case, where the number of walls removed may be less than n-1, meaning that some cells may not be reachable. The user needs to select the number of walls that will be removed and what type of union will be used. Then, we need to create and display the adjacency list that represents the maze that is created after removing the walls. Finally, we need to implement 3 different search algorithms to solve the maze, assuming that the starting point is bottom left and the goal is top right, and to compare their running times.

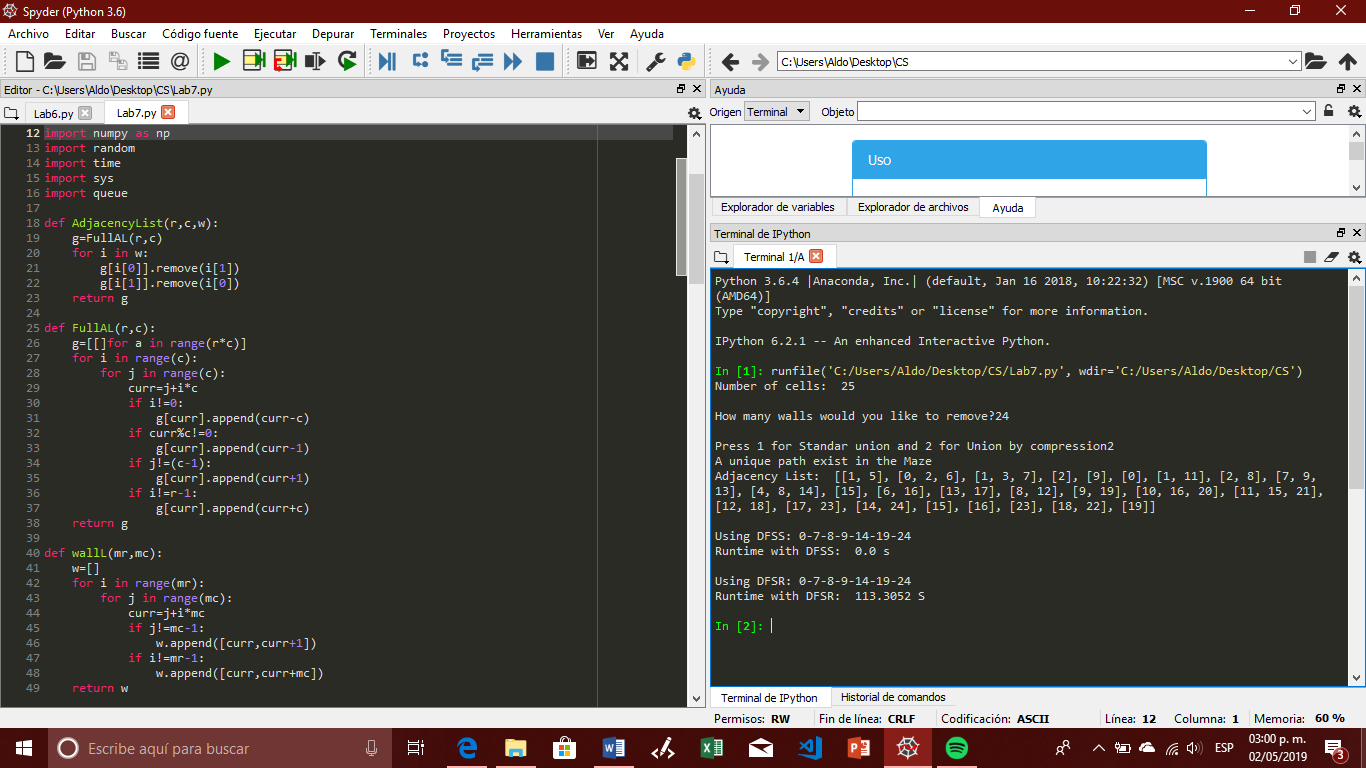
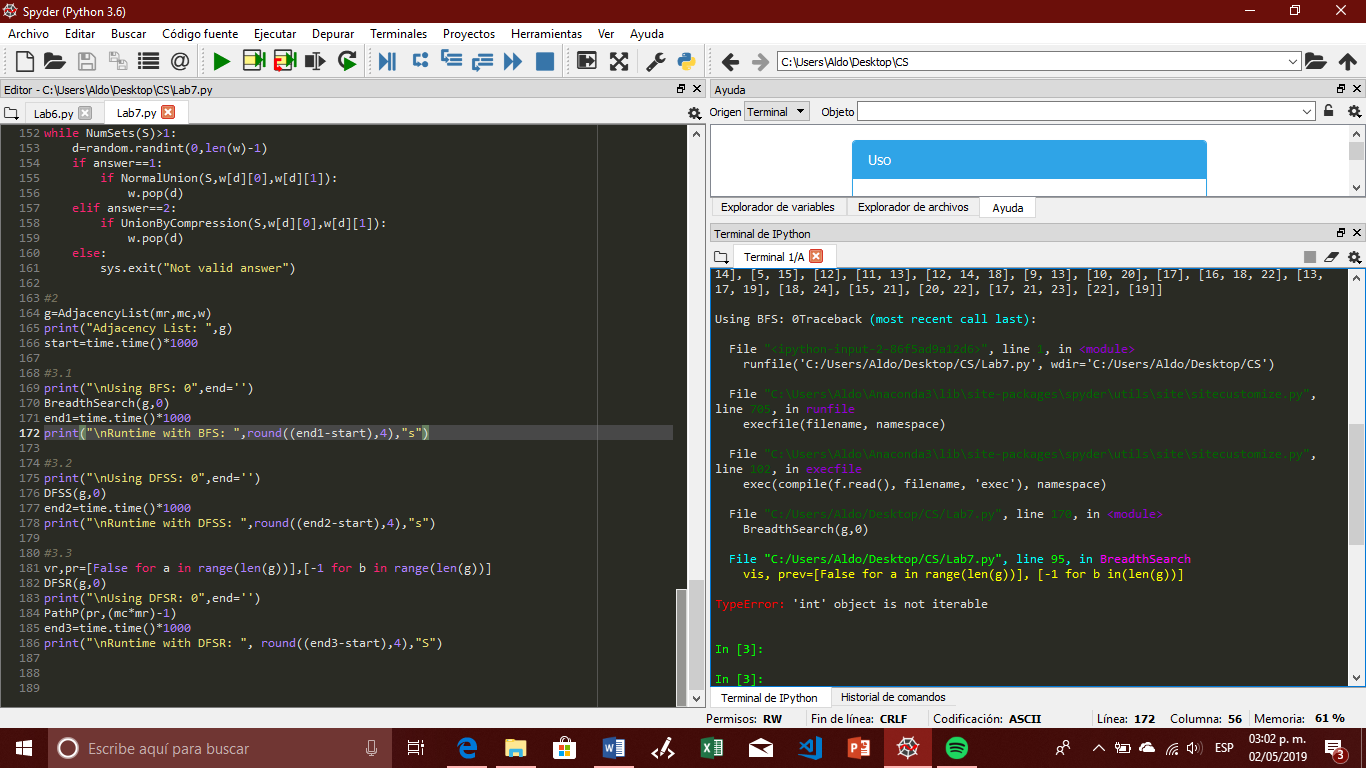
**Part 1:** The first thing done in this part, is to print a message that will tell the user if there is a unique path, if there is no path, or if there are multiple paths, according to the number of walls inputted. After this, depending on which type union has the user chosen, the process will be the same than in lab 6, but it will keep running until the number of walls inputted are removed, not until n-1 walls are removed.



**Part 2:** This method creates the adjacency that represents the maze, putting in the same set the elements that are not separated by a wall.



**Part 3:** Here we need to apply the different search methods to solve the maze assuming the start point is bottom left and the goal is top right. It is important to say that I wrote the function to compute the search with Breadth-first search, but for some reason it does not work, and shows an error saying that it is not possible to iterate that element.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ans=1 | 1 | 5 | 10 | 15 | 24 |
| DFSS | 0 | 0.1 | 0 | 0.9954 | 0.9634 |
| DFSR | 0 | 0.3 | 0.964 | 0.9954 | 0.9634 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| Ans=2 | 1 | 5 | 10 | 15 | 24 |
| DFSS | 0 | 0 | 0 | 0 | 0 |
| DFSR | 0 | 0.9722 | 0.964 | 0.9954 | 0.9614 |

After completing this lab, I feel that I have a better notion of how a maze works, how to solve them and how to iterate the. In general, I feel that I have a better sense on how to work with mazes, and how to implement different functions or characteristics with them.

import numpy as np

import random

import time

import sys

import queue

def AdjacencyList(r,c,w):

g=FullAL(r,c)

for i in w:

g[i[0]].remove(i[1])

g[i[1]].remove(i[0])

return g

def FullAL(r,c):

g=[[]for a in range(r\*c)]

for i in range(c):

for j in range(c):

curr=j+i\*c

if i!=0:

g[curr].append(curr-c)

if curr%c!=0:

g[curr].append(curr-1)

if j!=(c-1):

g[curr].append(curr+1)

if i!=r-1:

g[curr].append(curr+c)

return g

def wallL(mr,mc):

w=[]

for i in range(mr):

for j in range(mc):

curr=j+i\*mc

if j!=mc-1:

w.append([curr,curr+1])

if i!=mr-1:

w.append([curr,curr+mc])

return w

def NumSets(S):

c=0

for i in S:

if i==-1:

c+=1

return c

def NormalUnion(S,i,j):

I=Find(S,i)

J=Find(S,j)

if I!=J:

S[J]=I

return True

return False

def UnionByCompression(S,i,j):

I=FindForCompression(S,i)

J=FindForCompression(S,j)

if I!=J:

S[J]=I

return True

return False

def Find(S,i):

if S[i]<0:

return i

return Find(S,S[i])

def FindForCompression(S,i):

if S[i]<0:

return i

ans=FindForCompression(S,S[i])

S[i]=ans

return ans

def is\_unique(r,c):

if r==c-1:

print("A unique path exist in the Maze")

elif r<c-1:

print("Path may not be existent")

else:

print("Exist at least one path")

def BreadthSearch(g,v):

vis, prev=[False for a in range(len(g))], [-1 for b in(len(g))]

q=queue.Queue(1)

vis[v]=True

while not q.empty():

u=q.get()

for i in g[u]:

if not vis[i]:

vis[i]=True

prev[i]=u

q.put(i)

PathP(prev,len(g)-1)

def DFSS(g,v):

vis, prev=[False for a in range(len(g))], [-1 for b in range(len(g))]

s=[]

s.append(v)

vis[v]=True

while s!=[]:

q=s.pop()

for i in g[q]:

if not vis[i]:

vis[i]=True

prev[i]=q

s.append(i)

PathP(prev, len(g)-1)

def DFSR(g,s):

global vr

global pr

vr[s]=True

for i in g[s]:

if not vr[i]:

vr[i]=True

pr[i]=s

DFSR(g,i)

def PathP(prev,v):

if prev[v]!=1:

PathP(prev,prev[v])

print("-",end='')

print(v,end='')

def DSF(s):

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

#Main

mr=5

mc=5

print("Number of cells: ",mr\*mc)

NumWalls=int(input("How many walls would you like to remove?"))

answer=int(input("Press 1 for Standar union and 2 for Union by compression"))

w=wallL(mr,mc)

S=DSF(mr\*mc)

#1

is\_unique(NumWalls,(mc\*mr))

while NumSets(S)>1:

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

if answer==1:

if NormalUnion(S,w[d][0],w[d][1]):

w.pop(d)

elif answer==2:

if UnionByCompression(S,w[d][0],w[d][1]):

w.pop(d)

else:

sys.exit("Not valid answer")

#2

g=AdjacencyList(mr,mc,w)

print("Adjacency List: ",g)

start=time.time()\*1000

#3.1

"""print("\nUsing BFS: 0",end='')

BreadthSearch(g,0)

end1=time.time()\*1000

print("\nRuntime with BFS: ",round((end1-start),4),"s")"""

#3.2

print("\nUsing DFSS: 0",end='')

DFSS(g,0)

end2=time.time()\*1000

print("\nRuntime with DFSS: ",round((end2-start),4),"s")

#3.3

vr,pr=[False for a in range(len(g))],[-1 for b in range(len(g))]

DFSR(g,0)

print("\nUsing DFSR: 0",end='')

PathP(pr,(mc\*mr)-1)

end3=time.time()\*1000

print("\nRuntime with DFSR: ", round((end3-start),4),"S")

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