Luis Renteria

ID 88740232

CS 2302

Dr. Fuentes

Lab Report 7

**Introduction**

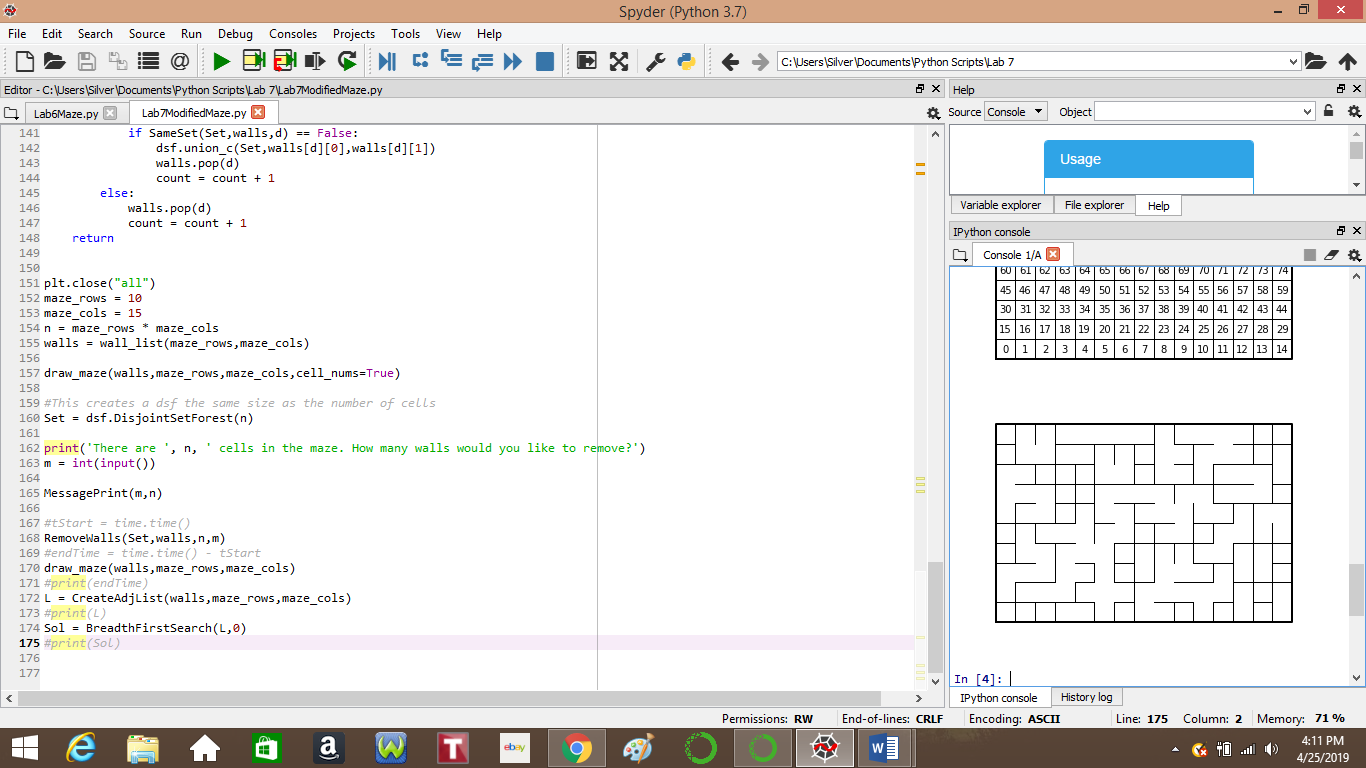
Our assignment for Lab 7 was to modify the program that we created in Lab 6, and use user input to determine the number of walls that needed to be removed. We were also required to use the maze that was created to create an adjacency list that could be used to create a graph representation of the maze.

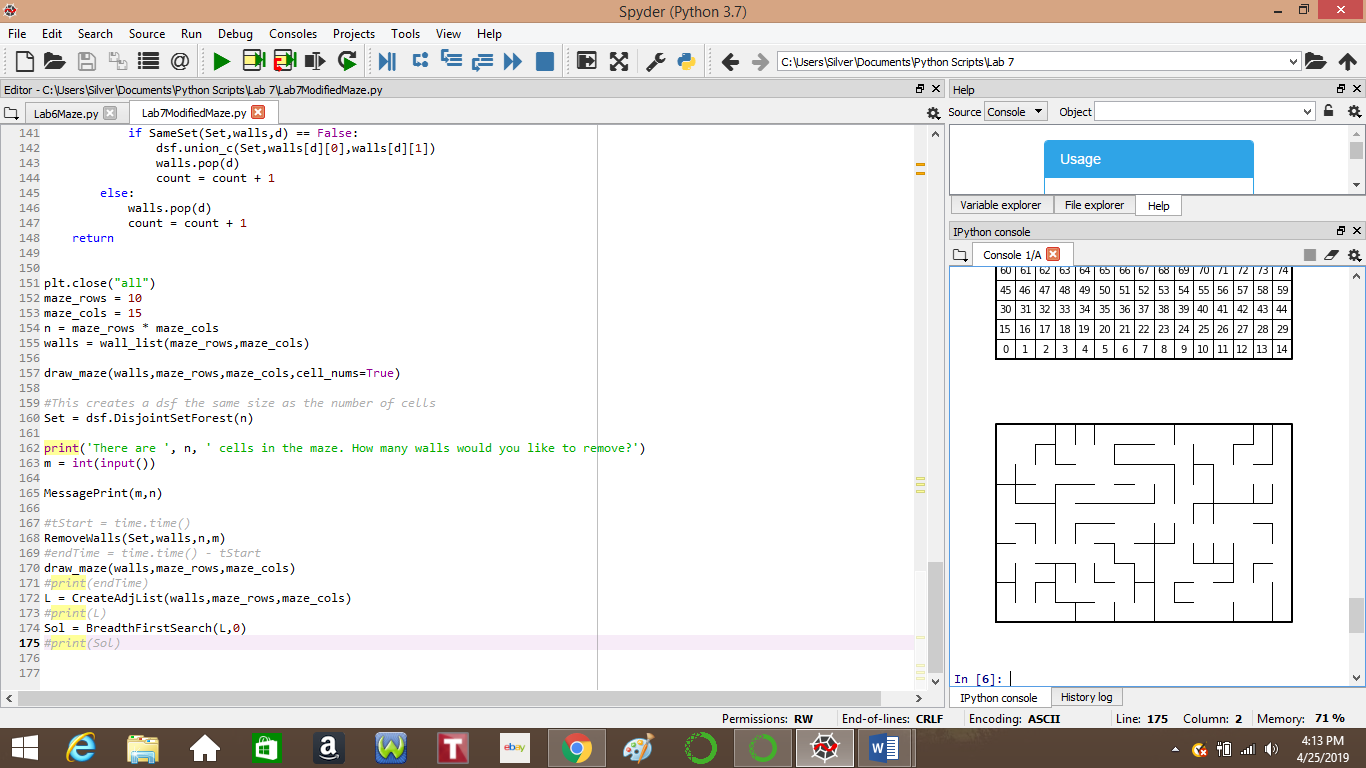
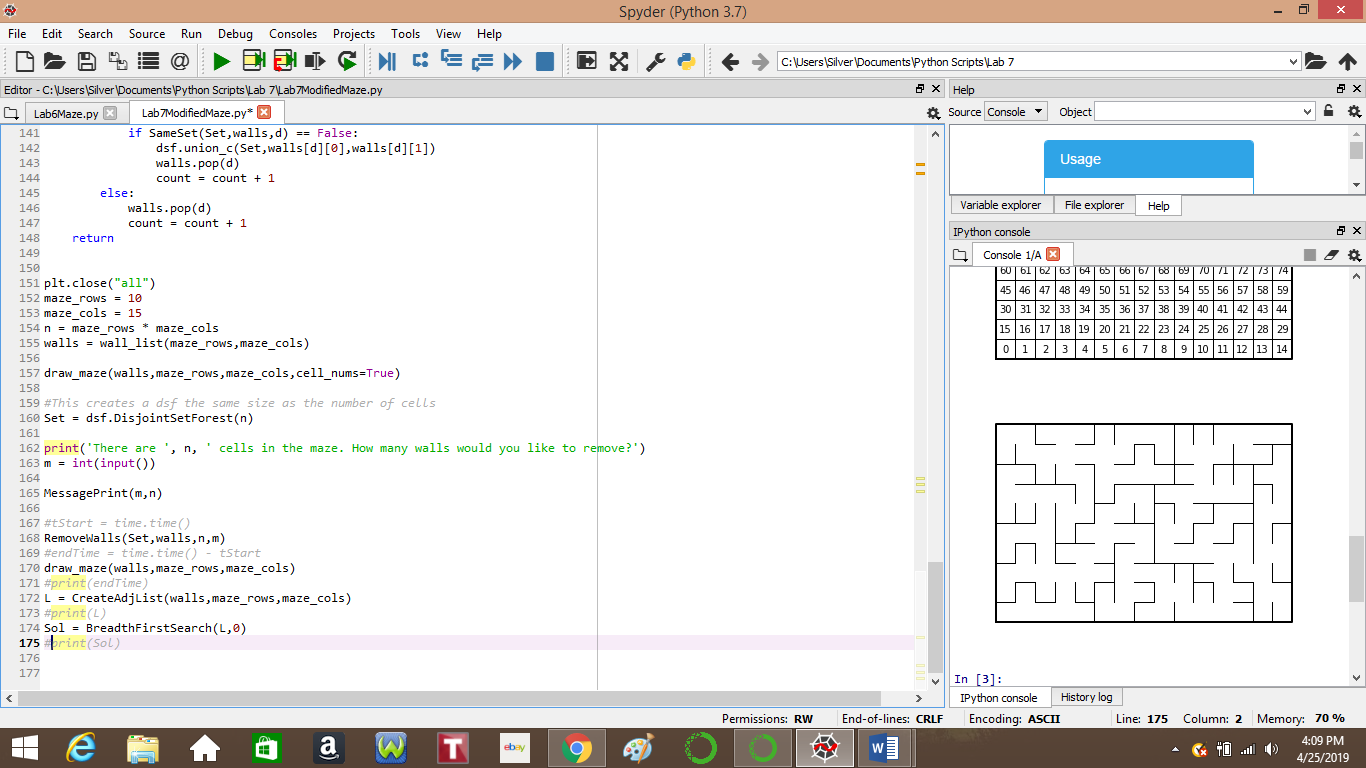
**Removing Walls**

Our first task was to remove a certain number of walls, as specified by the user. I first printed out a message stating the number of cells that were within the maze. The user then inputs an integer and a message is printed out in return. If a user chooses a number too little, a path is not guaranteed to exist, too many and there may be more than one path, the right amount, and there is a unique path.

The RemoveWalls method functions much like the original one, with a few changes. A counter keeps track of the number of walls removed, and a while loop is used to continue removing walls until the number of desired walls is removed. A dsf is still used to keep track of cells in the same set, and once there is a unique path, the program randomly removes walls if the counter is still not yet finished.

Below are some examples of mazes created, depending on the number of walls removed. Left has less removed walls than necessary, middle has the right number of removed walls, and right is more walls removed than necessary.





**Adjacency List**

Once the maze is complete, we come to the task of constructing an adjacency list. In order to this, I use the list of walls that are present after all removals. In the CreateAdjList method I use a for loop to traverse all cells but the very last one. In a series of if statements, I check if two cells are adjacent but checking if there is a wall of the particular cell and its neighbor within the walls list. If there is not a wall, then they are adjacent, and the neighbor is appended into a list of lists at the index of the cell. I continue this process until all cells are checked and the list is complete.

**Search Methods**

Using the pseudocode that Dr. Fuentes, I attempted to create the Breadth First and Depth First search methods, but was unsuccessful. BreadthFirstSearch creates a list that is a path within the maze, but fails to reach the goal, which is the top left corner. Depth First Search seemingly does not create a list of a path so I am unsure if it works.

**Conclusion**

With this lab, I was able to learn more about graphs and the uses of a disjoint set forest, as well as how a graph is created from an adjacency list. Even though I was not successful in creating the solving algorithms, I will continue to try and understand them.

**Academic Statement**

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.

Signed, Luis Renteria.

**Appendix**

Below is the code that I used from Lab 6, as well as the code given to us by Dr. Fuentes.

def SameSet(Set,walls,d):

#This method checks if two cells are in the same set.

i = int(walls[d][0])

j = int(walls[d][1])

#The find method is used to check the root of two cells.

#If they are the same, it returns true. If not, it returns false.

if dsf.find(Set,i) == dsf.find(Set,j):

return True

else:

return False

def RemoveWalls(Set,walls):

#This method removes walls until all celss are part of the same set.

while CheckSet(M) is not True:

#Here we choose a random wall

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

#If the cells are not in the same set, we remove the wall keeping them apart.

if SameSet(Set,walls,d) == False:

#NOTE: For this particular submission, I used union without compression.

dsf.union(Set,walls[d][0],walls[d][1])

walls.pop(d)

return

Here marks the start of the code given to us by Dr. Fuentes.

# Starting point for program to build and draw a maze

# Modify program using disjoint set forest to ensure there is exactly one

# simple path joiniung any two cells

# Programmed by Olac Fuentes

# Last modified March 28, 2019

import matplotlib.pyplot as plt

import numpy as np

import random

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 = 10

maze\_cols = 15

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

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

for i in range(len(walls)//2): #Remove 1/2 of the walls

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

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

walls.pop(d)

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