Luis Renteria

ID 88740232

CS 2302

Dr. Fuentes

Lab Report 5

**Introduction**

Our assignment for lab 6 was to modify a program that creates a random maze so that there is a clear path that connects each cell to another. In order to keep track of which cells were already connected, we implemented a disjoint set forest.

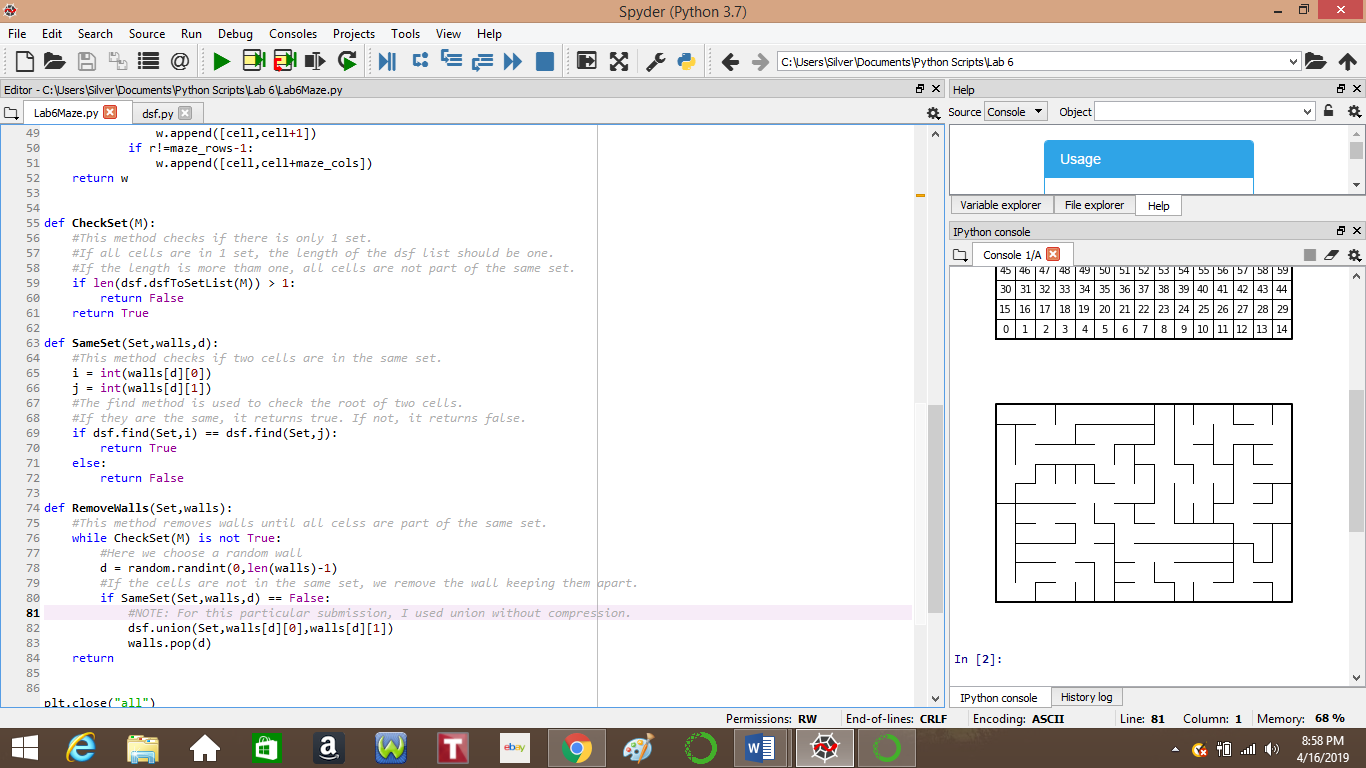
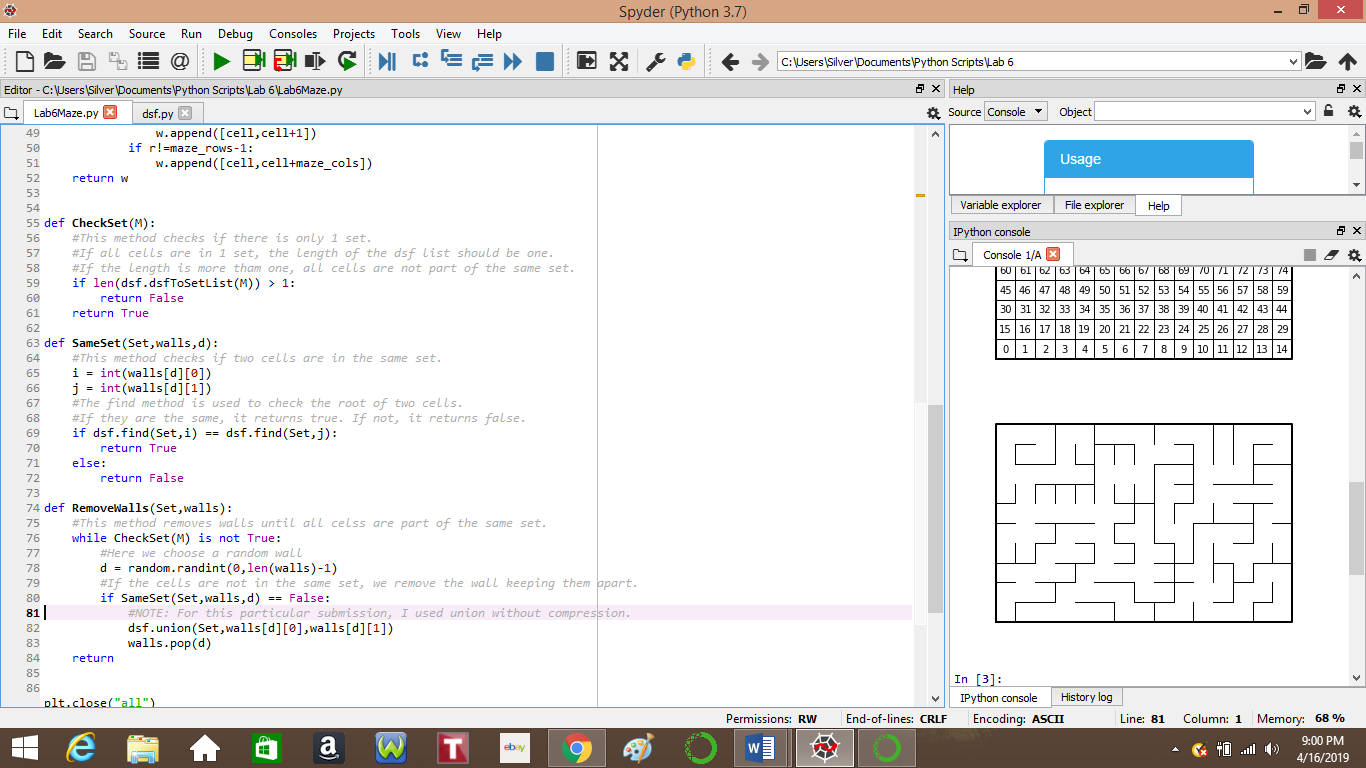
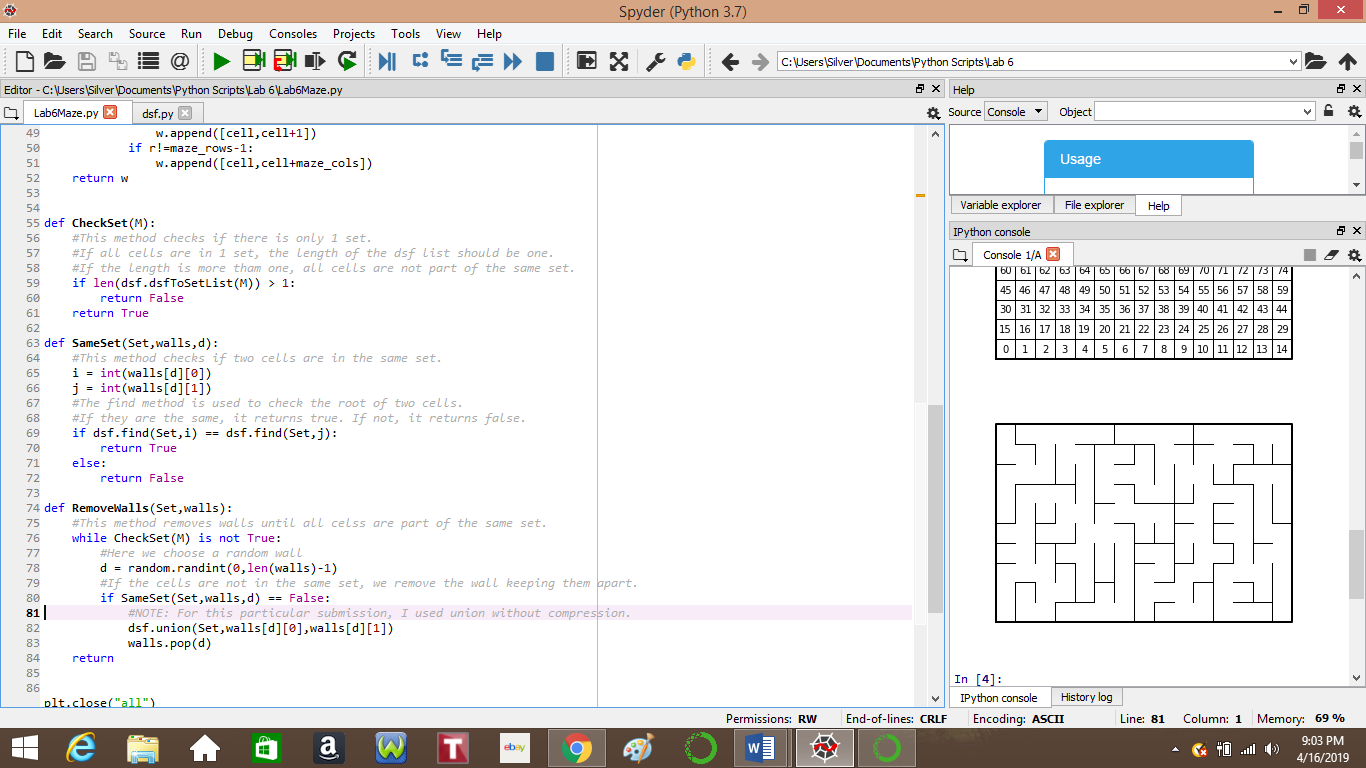
**Removing Walls**

The program to create the maze was already written for us, but the code randomly removed walls, resulting in a maze with unreachable cells and no definitive start or end. In order to keep track of each maze cell, I initialized a Disjoint Set Forest (dsf) that was the same size as the number of cells, so that each set in the dsf represented each cell in the maze. I then sent the dsf, along with the list of all walls in the maze, which was already given to us.

In the RemoveWalls method, I created a while loop with the method CheckSet as the condition. The CheckSet method determines if all elements are in one set. this is done by simply cheking the length of the set. If the length is 1, all elements are in the same set, which means there is a path from any one cell to another. As long as the CheckSet method is false, the while loop will continue. Within the loop, a random wall is chosen. The wall, wall list, and dsf are sent to the method SameSet.

Within the SameSet method, the cells that create the wall are compared by using dsf.find. If the roots are the same, then the two cells belong to the same set, and the method returns True. If the roots are not the same, then the two cells are not part of the same set, the method returns False. If the SameSet returns false, the two corresponding cells are joined using dsf.union, and the wall between the two cells is popped.

This process continues until all cells are within the same set, at which point, the finished maze is printed. Below are some of the random mazes the program outputs. Although there is no official start and end, you can reach any location from any other location in the maze.



**Running Times**

For my initial program, I used the union and find functions that do not use path compression. Below is a table of running times as well as the average running time for a 150-cell maze built with or without path compression. Each time is in seconds, rounded to the nearest four decimal places.

|  |  |  |
| --- | --- | --- |
| **Trial #** | **No Path Compression** | **With Path Compression** |
| 1. | 0.3123s | 0.2188s |
| 2. | 0.7491s | 0.1875s |
| 3. | 0.4756s | 0.2031s |
| 4. | 0.5870s | 0.2031s |
| 5. | 0.4883s | 0.2032s |
| Average | 0.5225s | 0.2031s |

As seen by the table, without path compression, the running time of the program is inconsistent, ranging from 0.3 to 0.7 seconds, while the program run with compression has a range of 0.1 to 0.2. In trials 3, 4, and 5 of the program with path compression, the running time is almost identical. With these results, it can be seen that no path compression results in longer running time while path compression is both faster, and nearly constant time.

**Conclusion**

With this lab, I was able to see an implementation of a Disjoint Set Forest. Although most of the code was written for us, the parts of the code that we had to program ourselves required us to understand how the maze is created, and how a dsf works. This lab improved my ability to read code that is given to us, as well as understand how to use a dsf and the functions within it.

**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 was given to us to complete this lab. Although we did use the dsf program, I did not add it here, as I did not edit it and it is available on the class webpage.

# 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)