Author: *Nguyen, Abram*
Assignment: *Lab 6 Report*
Course: *CS 2302 - Data Structures*
Instructor: *Fuentes, Olac*
T.A.: *Nath, Anindita*

Introduction:

In this lab, I'm using disjoint set forests to create different sized mazes. The defining characteristic of the maze is that there is exactly one path that can be taken between the root and any other point. I use a disjoint set forest to help guide the making of my mazes. I will compare the running time in milliseconds of two methods of creating mazes. The first uses union by size with path compression. The second method uses just the original union and find methods we were given in class. Finally, I'll draw out my mazes using pyplot.

Proposed solution design and implementation:

```
Create full maze with all adjacent cells are separated by a wall
Assign each cell to a different set in a disjoint set forest S
While S has more than one set
  Select a random wall w =[c1,c2]
  If cells c1 and c2 belong to different sets, remove w and join c1's set and c2's set
  otherwise do nothing
Display maze
```

1) For my first method, called 'make_maze', I created a disjoint set forest of size equal to the area of a given maze. In this case, I thought it would be enough just to create a maze of equal length and width, so I gave my method an integer 'size' as well as a list named 'walls'. I modified the list 'walls', a list of every single wall between cells, alongside my disjoint set forest. I choose a random wall out of the list 'walls'. I use the find function (original, less efficient version) to find the root of both cells separated by the random wall I chose. This find function will iterate down to a node's/cell's root and, in the worst case, can take O(n) time. Once the root of both cells are found, I compare them. If they're not the same, meaning that they don't belong to the same set, I will unionize them. The union function (original, less efficient version) will use the original find function to find the root of two cells and have the root of one point to the root of the other. My method will keep doing this until the disjoint set forest contains only one set. To check the number of sets, I used an inefficient way of doing it, I think. I would count the number of roots in the set before I remove a wall. If the number of sets came out to 1, I would not continue looping.
2) For my second method, called 'make_mazeUBS', everything is exactly the same except for my union and find functions. Instead of find, I used the more efficient version called 'find_c'. Everytime this function runs, it would compress the set that it traverses over. It makes the nodes in the set point to that set's root. The union by size function is called 'union_by_size'. When two sets are unionized, the function will used 'find_c' to compress the paths of the nodes it's finding. Then, it will compare the size of each set and have

the root of the smaller set point to the root of the larger set. Using ‘find_c’ and ‘union_by_size’ made the method of forming mazes faster.

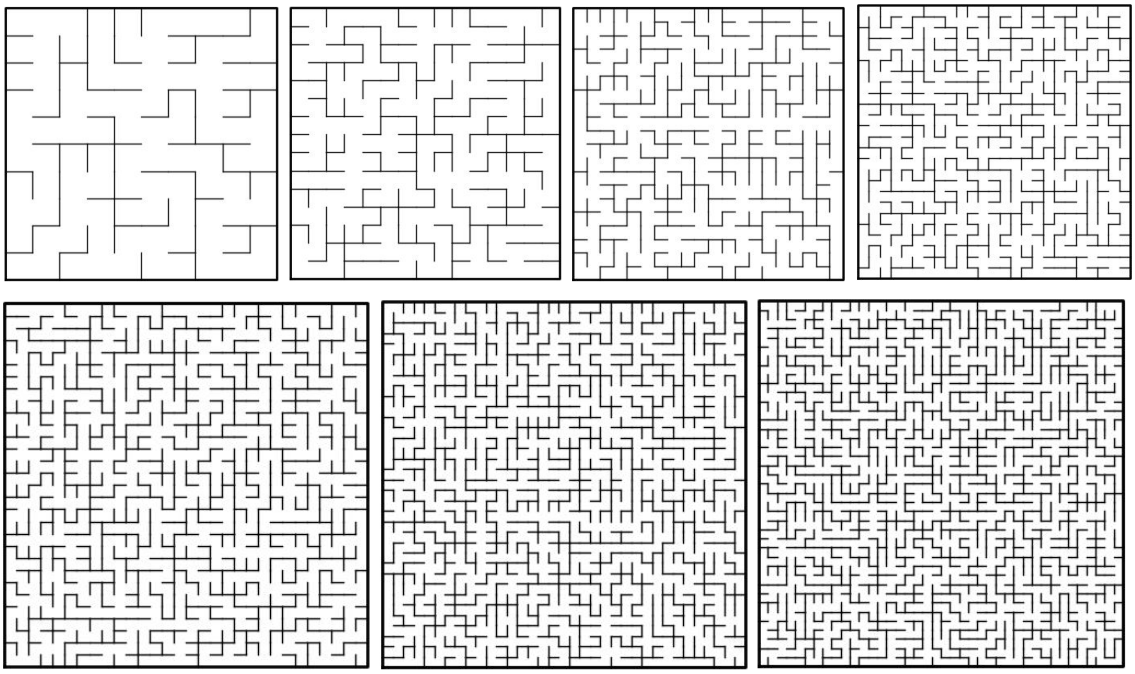
Experimental results:

For both of my methods, I tested 7 differently sized mazes. For each maze, I took the average time in milliseconds that it took to create the maze. The results are what I predicted, for the most part. Making mazes using union by size with path compression is faster than making mazes using normal union and find functions. As the sizes of the input increases, the difference in the running times increase as well. The difference between the running times of the 10*10 maze is about half a millisecond. The difference between the running times of the 40*40 maze is about 400 milliseconds. Below, I’ve included the results as well as seven different maze drawings.

```
Making mazes using normal union and find functions...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Average time making 10 * 10 maze → 6.5870285 ms
Average time making 15 * 15 maze → 28.84316444 ms
Average time making 20 * 20 maze → 123.24425152 ms
Average time making 25 * 25 maze → 296.26560211 ms
Average time making 30 * 30 maze → 555.66079276 ms
Average time making 35 * 35 maze → 1120.55982862 ms
Average time making 40 * 40 maze → 2321.52945655 ms

Making mazes using union by size w/ path compression...
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Average time making 10 * 10 maze → 6.0069561 ms
Average time making 15 * 15 maze → 27.40744182 ms
Average time making 20 * 20 maze → 88.83387702 ms
Average time making 25 * 25 maze → 286.57865524 ms
Average time making 30 * 30 maze → 638.54285649 ms
Average time making 35 * 35 maze → 1053.82227898 ms
Average time making 40 * 40 maze → 1892.04764366 ms
```

	Times(ms)	
Maze Size	**Normal**	**Union by size w/ find compression**
10*10	6.5870285	6.0069561
15*15	28.84316444	27.40744182
20*20	123.2442515	88.83387702
25*25	296.2656021	286.5786552
30*30	555.6607928	638.5428565
35*35	1120.559829	1053.822279
40*40	2321.529457	1892.047644



Conclusion:

This lab helped me learn a lot about disjoint set forests and how they can be used. This lab is one of the most interesting assignments in this course so far. I think that mazes are a great way of demonstrating the use of disjoint set forests. I think there's a faster and more efficient way of counting the number of sets in the disjoint set forest, but I wasn't sure exactly how to go about it. That's the only part of my work that I'm unhappy with.

Appendix:

```
"""

        Author:             Nguyen, Abram
        Assignment:         Lab 6
        Course:             CS 2302 - Data Structures
        Instructor:         Fuentes, Olac
        T.A.:               Nath, Anindita
        Last modified:      April 12, 2019

        Purpose of program: The purpose of this program is to demonstrate the application
                               of a disjoint set forest as a data structure. I've used
                               a disjoint set forest to create a maze. The defining
                               characteristic of the maze is that there is exactly one
                               path between two points.
        """
        import matplotlib.pyplot as plt
```

```
import numpy as np
import random
import time

def DisjointSetForest(size):
    return np.zeros(size,dtype=np.int)-1

def find(S,i):
    # Returns root of tree that i belongs to
    if S[i]<0:
        return i
    return find(S,S[i])

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: # Do nothing if i and j belong to the same set
        S[rj] = ri  # Make j's root point to i's root

def find_c(S,i):
    if S[i]<0:
        return i
    r = find_c(S,S[i])
    S[i] = r
    return S[i]

def union_by_size(S, i, j):
    ri = find_c(S, i)
    rj = find_c(S, j)
    if ri != rj:
        if S[ri] > S[rj]:
            S[rj] += S[ri]
            S[ri] = rj
        else:
            S[ri] += S[rj]
            S[rj] = ri

def wall_list(maze_rows, maze_cols):
    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

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)

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

# normal union and find functions
def make_maze(size, walls):
    #Assign each cell to a different set in a disjoint set forest S
    S = DisjointSetForest(size*size)
    sets = 2
    #While S has more than 1 set:
    while sets > 1:
```

```
        #check number of sets
        sets = 0
        for i in range(len(S)):
            if S[i] < 0:
                sets+=1
        #Select a random wall w =[c1,c2]
        w = random.randint(0, len(walls)-1)
        #If cells c1 and c2 belong to different sets,
        if find(S, walls[w][1]) != find(S, walls[w][0]):
            #remove w and join c1's set and c2's set
            union(S, walls[w][0], walls[w][1])
            walls.pop(w)
    return walls



# using union by size and find path compression functions
def make_mazeUBS(size, walls):
    #Assign each cell to a different set in a disjoint set forest S
    S = DisjointSetForest(size*size)
    sets = 2
    #While S has more than 1 set:
    while sets > 1:
        #check number of sets
        sets = 0
        for i in range(len(S)):
            if S[i] < 0:
                sets+=1
        #Select a random wall w =[c1,c2]
        w = random.randint(0, len(walls)-1)
        #If cells c1 and c2 belong to different sets,
        if find_c(S, walls[w][1]) != find_c(S, walls[w][0]):
            #remove w and join c1's set and c2's set
            union_by_size(S, walls[w][0], walls[w][1])
            walls.pop(w)
    return walls

"""
###############################################################################
###############################################################################
###############################################################################
"""
```

```
plt.close("all")
maze_size = [5, 10, 20, 30, 40] # more than 40 might crash my program

#normal union and find function
print("Making mazes using normal union and find functions...")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
for s in maze_size: #for each sized maze,
   sum = 0
   for r in range(5): #collect an average maze creation time
       walls = wall_list(s, s)
       start = time.time() # start time
       maze = make_maze(s, walls) # timing make_maze() function (normal)
       end = time.time() # end time
       sum += end-start
   print("Average time making",s,"*",s,"maze →", round((sum*1000)/5, 8), "ms")



# union by size w/ path compression function
print("\nMaking mazes using union by size w/ path compression...")
print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~")
for s in maze_size: #for each sized maze,
   sum = 0
   for r in range(5): #collect an average maze creation time
       wallsUBS = wall_list(s, s)
       start = time.time() # start time
       mazeUBS = make_mazeUBS(s, wallsUBS) # timing maze_mazeUBS() function (union by size w/compression)
       end = time.time() # end time
       sum += end-start
   print("Average time making",s,"*",s,"maze →", round((sum*1000)/5, 8), "ms")
   # 5 maze drawings using union by size w/ path compression
   draw_maze(mazeUBS, s, s, cell_nums=False)
################################################################################
# End of program
```

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.

- Abram Nguyen