Lab 6: Disjoint Set Forests

Manuel Ruvalcaba

CS2302 1:30-2:50

1. Introduction

The purpose of this program is to build a maze using a disjoint set forest so that there is there is exactly one simple path (that is, a path that does not visit any cell more than once) separating any two cells.

2. Proposed Solution Design and Implementation

In order to build the mazes, I was provided two methods named *draw_maze* and *wall_list*. The *draw_maze* method draws the maze that is created in the program. The *wall_list* method creates and returns a list of walls that are in the maze.

I also created various methods named *DisjointSetForest*, *find*, *find_c*, *union*, and *union_by_size.*

The method named *DisjointSetForest* accepts an integer as the size for the disjoint set forest, creating and returning a numpy array of zeros of the given size. In this case, the size would be the maze rows multiplied by the maze columns.

The *find* method accepts an integer and a disjoint set forest, finding and returning the root of the forest containing the integer.

The *find_c* method accepts an integer and a disjoint set forest, finding and returning the root of the forest containing the integer, while compressing the path between the integer and the root.

The *union* method accepts a disjoint set forest and two integers, then finds their roots. If the roots are the same, it does nothing. Otherwise, it unifies the trees.

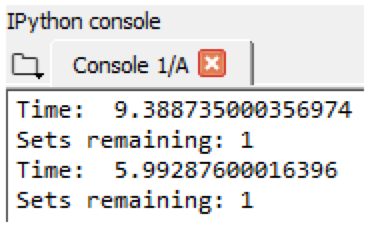
The *union_by_size* method accepts a disjoint set forest and two integers, then finds their roots. If the roots are the same, it does nothing. Otherwise, it makes the root of the smaller tree point to the root of the larger tree while compressing the path.

I then have a *while* loop that is executed as long as the number of sets in the disjoint set forest are greater than one. What the loop does is, it selects a random wall from the wall list. It then checks if the cells that are separated by the wall are in the same set. If they are, it does nothing. Otherwise, it unifies the sets, using either standard union or union by size, and removes the wall.

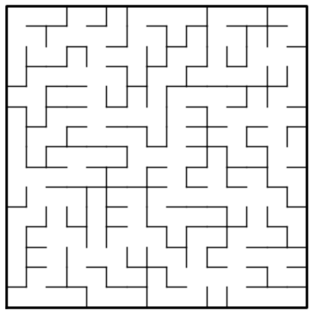
3. Experimental Results

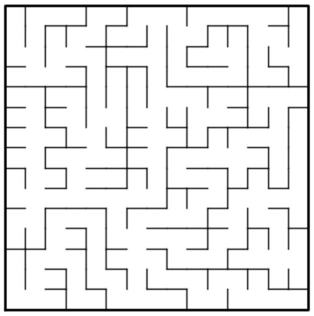
In order to test my code, I created mazes with three different sizes. I then ran the code, which printed the time it took to build the mazes, using the *timeit* function, as well as the number of sets remaining in the disjoint set forest.

My first test was with a maze with 15 rows and 15 columns. The runtime of the maze with standard union was 9.388735000356974 ms and the runtime for the maze with union by size and path compression was 5.99287600016396 ms.

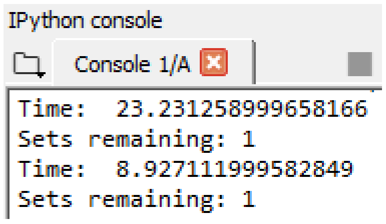


```
Time:  9.388735000356974
Sets remaining: 1
Time:  5.99287600016396
Sets remaining: 1
```

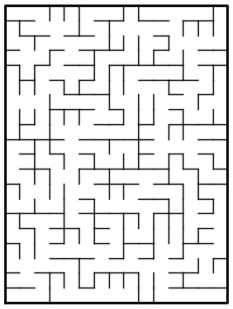


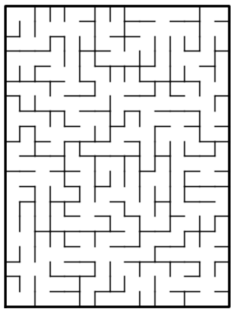


My second test was with a maze with 20 rows and 15 columns. The runtime of the maze with standard union was 23.231258999658166 ms and the runtime for the maze with union by size and path compression was 8.927111999582849 ms.

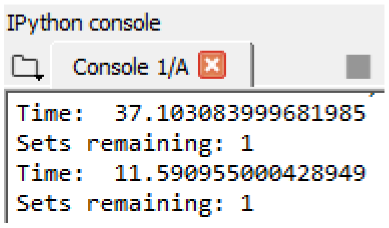


```
Time:  23.231258999658166
Sets remaining: 1
Time:  8.927111999582849
Sets remaining: 1
```

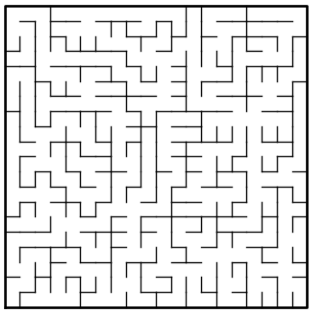


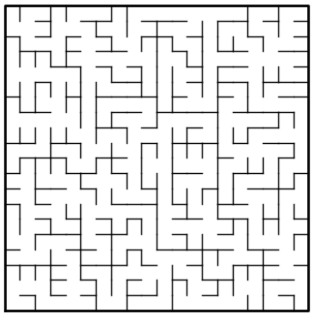


My final test was with a maze with 20 rows and 20 columns. The runtime of the maze with standard union was 37.103083999681985 ms and the runtime for the maze with union by size and path compression was 11.590955000428949 ms.



```
Time:  37.103083999681985
Sets remaining: 1
Time:  11.590955000428949
Sets remaining: 1
```





Here is a table with the values.

	Rows = 15 Columns = 15	Rows = 20 Columns = 15	Rows = 20 Columns = 20
Time for maze with standard union	9.388735	23.231259	37.103084
Time for maze with union by size	5.992876	8.927112	11.590955

4. Conclusion

With this project, I have learned how to build a maze that has exactly one simple path (that is, a path that does not visit any cell more than once) separating any two cells, using a disjoint set forest. I guess I can create a website that makes mazes of a size requested by the user now.

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.

[illegible]

5. Appendix

```
'''
Course: CS2302 MW 1:30-2:50
Author: Manuel A. Ruvalcaba
Assignment: Lab #6 Disjoint Set Forests
Instructor: Dr. Olac Fuentes
TA: Anindita Nath, Maliheh Zargaran
Date of Last Modification: April 10, 2019
Purpose of the Program: The purpose of this program is to build a maze using a
                        disjoint set forest so that there is there is exactly
                        one simple path (that is, a path that does not visit
                        any cell more than once) separating any two cells. This
                        program also tests the running times when using standard
                        union and union by size
'''

import matplotlib.pyplot as plt
import numpy as np
import random
import timeit
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

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

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 union_by_size(S,i,j):
    # Joins i's tree and j's tree, if they are different
    ri = find_c(S,i)
    rj = find_c(S,j)
    if ri!=rj: # Do nothing if i and j belong to the same set
        if S[rj] < S[ri]:
            S[rj]+=S[ri]
            S[ri] = rj # Make i's root point to j's root
        else:
            S[ri]+=S[rj]
            S[rj] = ri # Make j's root point to i's root

#This is where you test the code

plt.close("all")
maze_rows = 20
maze_cols = 20
```

```
S = DisjointSetForest(maze_rows*maze_cols)
sets = len(S)

walls = wall_list(maze_rows,maze_cols)

draw_maze(walls,maze_rows,maze_cols,cell_nums=True)
start = timeit.default_timer()
while sets > 1: #removes walls until there is only one set using standard union
    d = random.randint(0,len(walls)-1)
    a = find(S,walls[d][0])
    b = find(S,walls[d][1])
    if a != b: #checks if the two cells are not part of the same set
        sets -= 1
        union(S,walls[d][0],walls[d][1])
#        print('removing wall ',walls[d])
        walls.pop(d)
stop = timeit.default_timer()
print('Time: ', (stop - start)*1000)
print('Sets remaining:',sets)
draw_maze(walls,maze_rows,maze_cols)

maze_rows = 20
maze_cols = 20

S = DisjointSetForest(maze_rows*maze_cols)
sets = len(S)

walls = wall_list(maze_rows,maze_cols)

start = timeit.default_timer()
while sets > 1: #removes walls until there is only one set using union by size and
path compression
```

```
    d = random.randint(0,len(walls)-1)
    a = find_c(S,walls[d][0])
    b = find_c(S,walls[d][1])
    if a != b: #checks if the two cells are not part of the same set
        sets -= 1
        union_by_size(S,walls[d][0],walls[d][1])
#        print('removing wall ',walls[d])
        walls.pop(d)
stop = timeit.default_timer()
print('Time: ', (stop - start)*1000)
print('Sets remaining:',sets)
draw_maze(walls,maze_rows,maze_cols)
```