Maria F. Corona Ortega
ID: 80560734
E-Mail: mfcoronaortega@miners.utep.edu
CS 2302 | MW 1:30-2:50 am
Instructor: Dr. Olac Fuaentes
TA: Anindita Nath

CS 2302 Lab 6 Report

Introduction

The purpose of this assignment is to build a maze using disjoint sets. For every cell in the maze of size m x n we are to create a set. From here using the code provided to build a maze, we are to build the maze walls into a list and select random walls. When these walls are removed we are to unite sets using both standard union and union by compression. In the end we are to have a single set of all walls being connected when removing a wall from the list. This assignment is for us to explore disjoint sets and improve our problem solving skills.

Proposed Solution

I will keep the original code to build the walls of the maze and will import only certain methods in DSF class. These methods are *DisjointSetForest, find/find_c, union/ union_c.* From here I will follow the pseudocode provided on the lab.

- **Task 1 (Build Maze using DFS standard union):** While S is more than one set. Then a random variable is generated which helps us select a wall. In an if statement we check if the sets attached to the wall share a root meaning they belong to the same set. If this is true we perform standard union and pop said wall.
- **Task 1 (Build Maze using DFS union by compression):** Almost identical to the previous task, while S is more than one set. Then a random variable is generated which helps us select a wall. In an if statement we check if the sets attached to the wall share a root meaning they belong to the same set. If this is true we perform union by compression and pop the wall.

Methods

buildMaze: This method receives three parameters the list of walls, the empty disjoint set, and size the size of the maze. A timer is initialized to record running time. We then use the size of the disjoint set and maze to traverse the random walls. I then used *find* to check the root of the sets associated with the wall that was randomly selected. If the roots differ, we perform *union* and pop that said

wall making it disappear from final maze draw. We stop the timer and print the results then draw the maze.

buildMaze_c: This method receives three parameters the list of walls, the empty disjoint set, and size the size of the maze. A timer is initialized to record running time. We then use the size of the disjoint set and maze to traverse the random walls. I then used *find_c* to check the root of the sets associated with the wall that was randomly selected. If the roots differ, we perform *union_c* and pop that said wall making it disappear from final maze draw. We stop the timer and print the results then draw the maze.

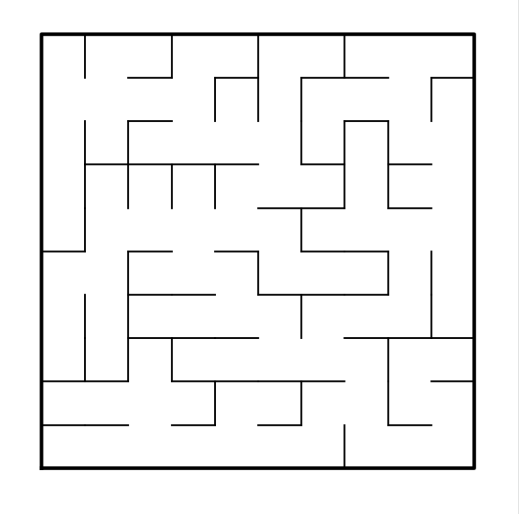
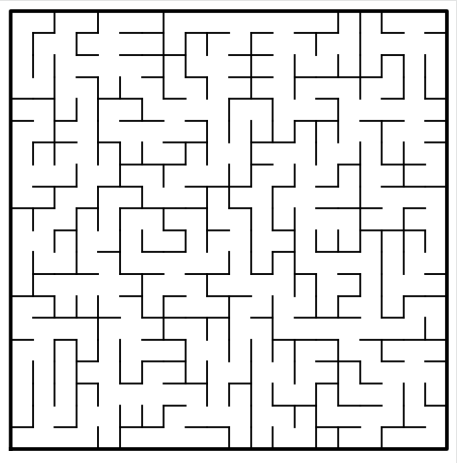
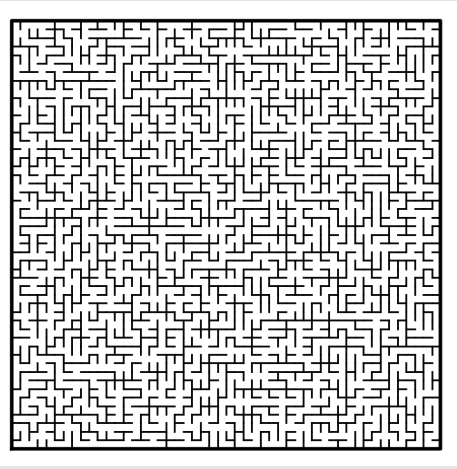
Experimental Results

Building Maze with Disjoint sets using standard union

n for an *n x n* maze	Running Rime
10	0.001033502005157060
20	0.005372216997784560
50	0.04156860899820460
100	0.27225661400007100

Building Maze with Disjoint sets using union with path compression

n for an *n x n* maze	Running Rime
10	0.0011204500042367700
20	0.003653628009487870
50	0.05486352200387050
100	0.7544950000010430

	size n x n
	n = 10
	n = 20
	n = 50

Conclusion

In conclusion, this lab taught us how to efficiently implement disjoint sets and modify data in order to efficiently use such structure. I learned how mazes are built to the point the have a solution which is very interesting. I really enjoyed this lab and the implementation and future possibilities of applications of disjoint sets.

Appendix

```
#Course: CS 2302 Data Structures | Spring 2019
#Author: Maria Fernanda Corona Ortega
#Assignment: Lab 5
#Instructor: Olac Fuentes
#Purpose of Code: 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
#Last Modification: 4/1/2019

import matplotlib.pyplot as plt
import numpy as np
import random
from scipy import interpolate

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): #Find with path compression
    if S[i]<0:
        return i
    r = find_c(S,S[i])
    S[i] = r
    return r

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:
        S[rj] = ri

def union_c(S,i,j):
    # Joins i's tree and j's tree, if they are different
    # Uses path compression
    ri = find_c(S,i)
    rj = find_c(S,j)
    if ri!=rj:
        S[rj] = ri


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 = 20
maze_cols =  20

walls = wall_list(maze_rows,maze_cols)

draw_maze(walls,maze_rows,maze_cols,cell_nums=True)

size = maze_rows*maze_cols

S = DisjointSetForest(size)

for i in range(size*2 + 1):#While S has more than one set
    ran = random.randint(0,len(walls)-1)
    if find_c(S, walls[ran][0]) != find_c(S, walls[ran][1]):
#        print('root of', walls[ran][0],'is',find_c(S, walls[ran][0]))
#        print('root of', walls[ran][1],'is',find_c(S, walls[ran][1]))
#        print('removing wall ',walls[ran])
#        print('union between sets', walls[ran][0], 'and ', walls[ran]
[1])

        union_c(S,walls[ran][0],walls[ran][1])
#        print('root of', walls[ran][0],'is NOW',find_c(S, walls[ran]
[0]))
#        print('root of', walls[ran][1],'is NOW',find_c(S, walls[ran]
[1]))
```

```
        walls.pop(ran)
#         print()
#         print(len(walls))
#         print(walls)
#         print(S)

print(walls)

draw_maze(walls,maze_rows,maze_cols)
print(S)
```

Academic Honesty Certification

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.

Maria F. Corona Ortega

09/09/2017