Ivan Perez
80602918
Lab 6 Report

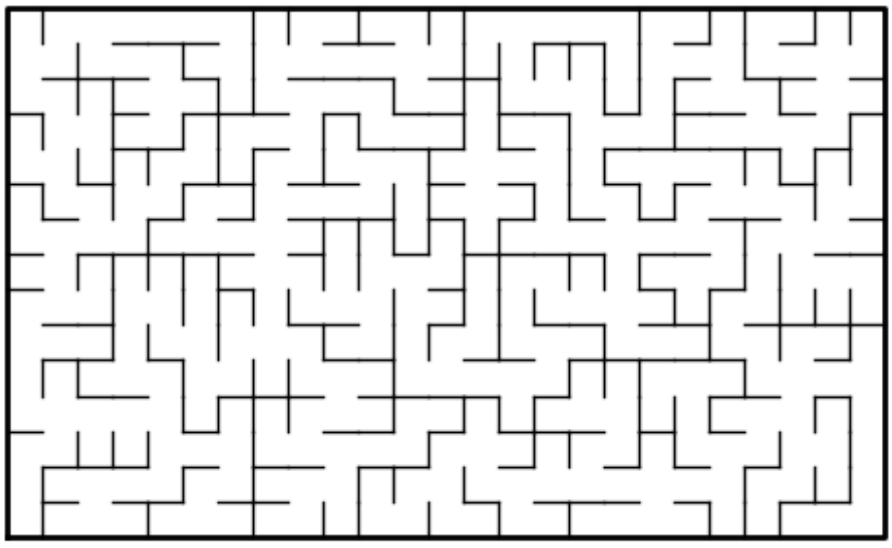
Introduction

Using the program provided to us by Mr. Fuentes, basic Disjoint set forest object methods and maze figure construction, a maze must be created. Walls can be randomly removed, which is what the code provided, but there's many probabilities of cycles, boxed out areas, or areas that can't even be reached to go from start to end. Using a forest, an ideal maze should be able to be constructed.

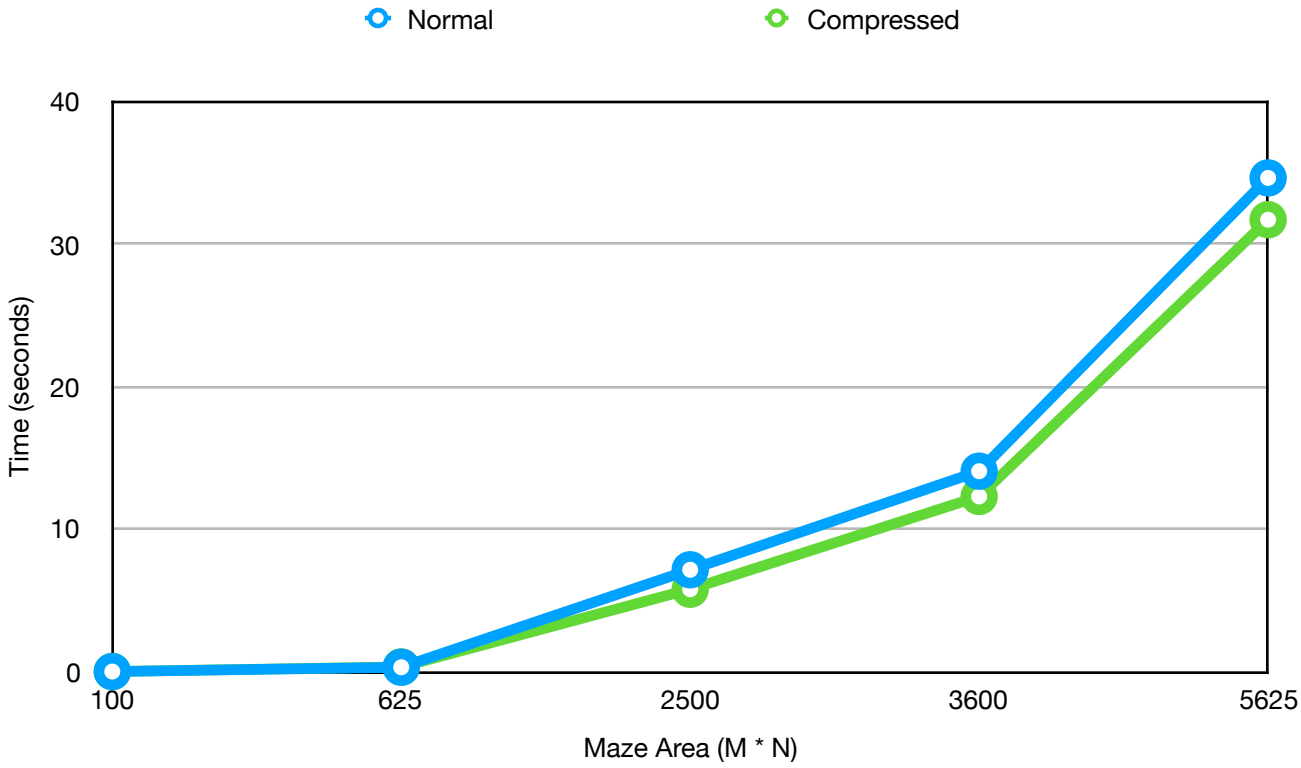
Proposed solution design and implementation

My proposed solution after seeing the numbers for each box, was to randomly select numbers and remove walls for each union of the values. Using the wall list, I picked a random wall, checked to make sure the values weren't already in the same set, and remove the wall if they were not in the same set. This prevents cycles. To make sure I removed the correct amount of walls, I used a while loop to remove walls until there was only one set in the set forest. I created two different methods to create mazes, one of them utilizes compressed find and union by size instead of the regular methods to increase efficiency. I also created test methods that call the maze creation method but also displays the runtime of different variables.

Experimental results



Above is an example of a maze created with the maze creation method that utilizes disjoint set forests. I used a 15x25 sized maze just to demonstrate the lack of cycles and boxed out areas and how its an ideal maze. Below is a graph of running times of maze creation using compressed and normal methods of find and union. The x axis is the area of the maze created, for example the start is 100 which is a 10x10 sized maze. Initially the values were similar but as the mazes got bigger, compression took less time as the mazes got larger. Often times, the compressed method would initially even take longer than the normal methods but in the long run the compressed versions were more efficient.



Conclusions

This lab helped visualize a way disjoint set forests work and implemented them in a fun maze creation way. It showed how efficient an algorithm can be even with small things such as making a maze. We initially started by removing random walls within a maze causing a variety of problems but by implementing this algorithm, things became infinitely better and more consistent. I learned an easier way to visualize sets and how forests aid in utilizing sets.

Appendix

```
#################################################################
###############
#
# Course: CS2302 (MW - 1:30pm)
# Author: Ivan Perez
# Assignment: Lab 6 - Disjoint Set Forests
# Instructor: Olac Fuentes
# T.A.: Anindita Nath
# Last Modified: Apr 17, 2019
# Purpose: This code creates a maze by using disjoint set
forests. One of the
# ways uses compression while the other doesnt. It also includes
methods to
# display the maze as a figure and contains other methods needed
for the
# Disjoint set Forests
#
#################################################################
###############

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 NumSets(S):
    # counts the number of sets in the forest
    count =0
    for i in range(len(S)):
        if S[i]<0:
            count += 1
    return count

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:
        S[rj] = ri


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_by_size(S,i,j):
    # if i is a root, S[i] = -number of elements in tree (set)
    # Makes root of smaller tree point to root of larger tree
    # Uses path compression
    ri = find_c(S,i)
    rj = find_c(S,j)
    if ri!=rj:
        if S[ri]>S[rj]: # j's tree is larger
            S[rj] += S[ri]
            S[ri] = rj
        else:
            S[ri] += S[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


def createMaze_Comp(M, N):
    #Creates Maze with Compression
    dsf = DisjointSetForest(M*N)
    walls = wall_list(M, N)

    while NumSets(dsf)> 1:
        d = random.randint(0,len(walls)-1)
        i = walls[d][0]
        j = walls[d][1]
        if find_c(dsf, i) != find_c(dsf, j):
        # if values are not part of the same set
            walls.pop(d) # removes wall
            union_by_size(dsf, i , j ) #unites the two sets
    return walls

def createMaze(M, N):
    # Creates Maze without Compression
    dsf = DisjointSetForest(M*N)
    walls = wall_list(M, N)

    while NumSets(dsf)> 1:
        d = random.randint(0,len(walls)-1)
        i = walls[d][0]
        j = walls[d][1]
        if find(dsf, i) != find(dsf, j):
        # if values are not part of the same set
```

```
            walls.pop(d)
            union(dsf, i , j )
    return walls

##////////////////////////// Test
Code ////////////////////////////////##
plt.close("all")

# Displays how a maze is made using the method (test
Demonstration)
walls = createMaze_Comp(15, 25)
draw_maze(walls,15,25)

def Test(M,N):
    # Method to create maze and print runtime
    Start = time.time()
    createMaze(M, N)
    End = time.time()
    t = End - Start
    print('Maze Area: ', M*N)
    print('Runtime: ', t)

def Test_Comp(M,N):
    # Method to create maze and print runtime using compression
    Start = time.time()
    createMaze_Comp(M, N)
    End = time.time()
    t = End - Start
    print('Maze Area: ', M*N)
    print('Runtime: ', t)

print('Mazes made without compression')
Test(10, 10)
Test(25, 25)
Test(50, 50)
Test(60, 60)
Test(75, 75)
print()

print('Mazes made with compression')
Test_Comp(10, 10)
Test_Comp(25, 25)
Test_Comp(50, 50)
Test_Comp(60, 60)
Test_Comp(75, 75)
print()
```

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.