Lab 6 report

Nicole Favela

CS2301

April 16, 2019

**Introduction:**

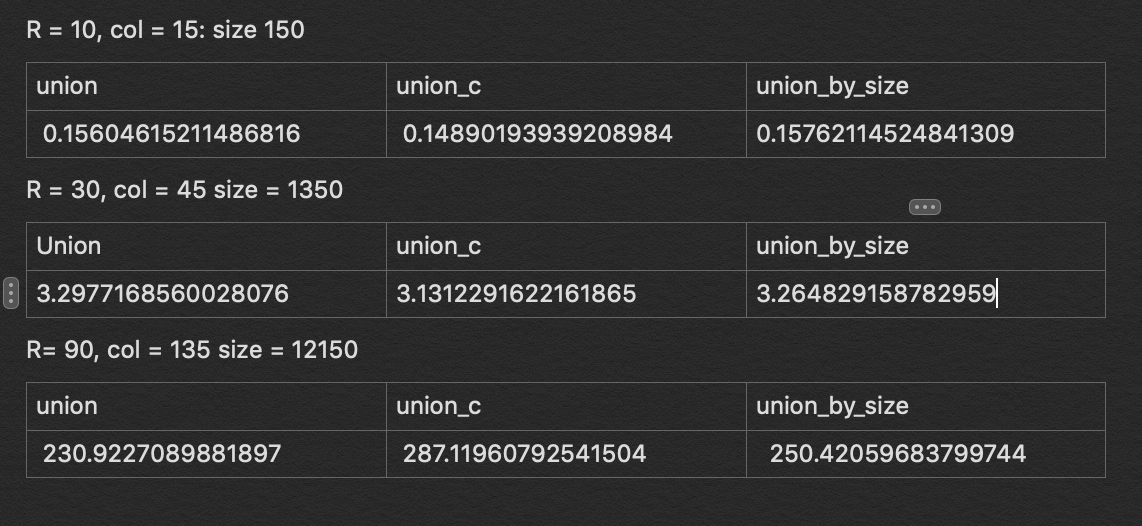
The purpose of this assignment is to use disjoint set forests to create a maze and to compare the running times of union\_by\_size and standard union. I used the code provided which draws the mazes which consists of cells (a two dimensional array) and then removes walls randomly. I was tasked with changing the code to break the walls in adjacent cells if and only if they were not in the same set.

**Proposed solution and implementation:**

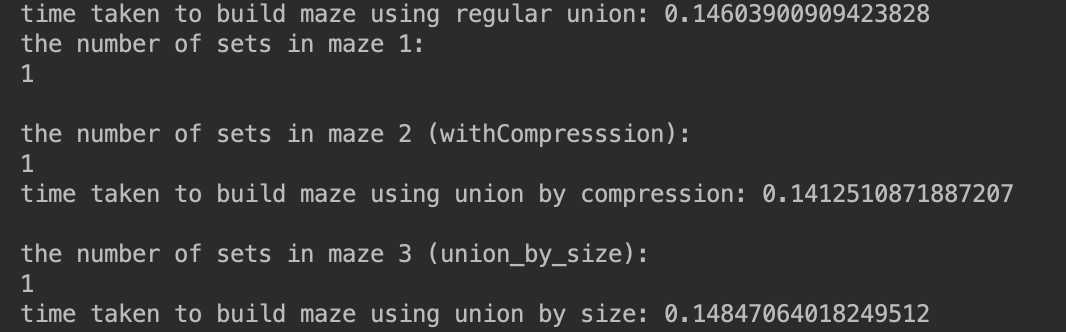
I knew from the first introduction to disjoint set forests described in class that I needed to union components in the dsf and if they are not already in the same set, I needed to break or pop a wall. I needed to make sure at the end of the operations, that there was only 1 set left (the set containing all 150 elements). To count the number of sets, I used a function called numSets which goes through all the elements in S (the array for the dsf) and counts the elements that are -1 (a root) or point to other elements. I also added union\_by\_size which was provided. Find\_c is similar as it works with union\_by\_size and union\_c and goes straight to the root. In my function buildMaze, I am getting a random integer called d and using that to select a wall from an array of walls. C1 and c2 are elements in the dsf at positions 0 and 1 respectively. Inside the loop I am checking if c1 and c2 are in the set. If they are not in the set, I am joining them and popping a wall. This continues while the number of sets is more than 1. BuildMaze\_withCompression does the same thing except using union by compression. I also used union\_by\_size in my function called buildMaze\_with\_union\_by\_size. Union by compression points every element to the root if they are in the same set. Union by size is similar, except it makes the root of the smaller tree point to the root of the larger tree.

**Runtimes and samples outputs:**

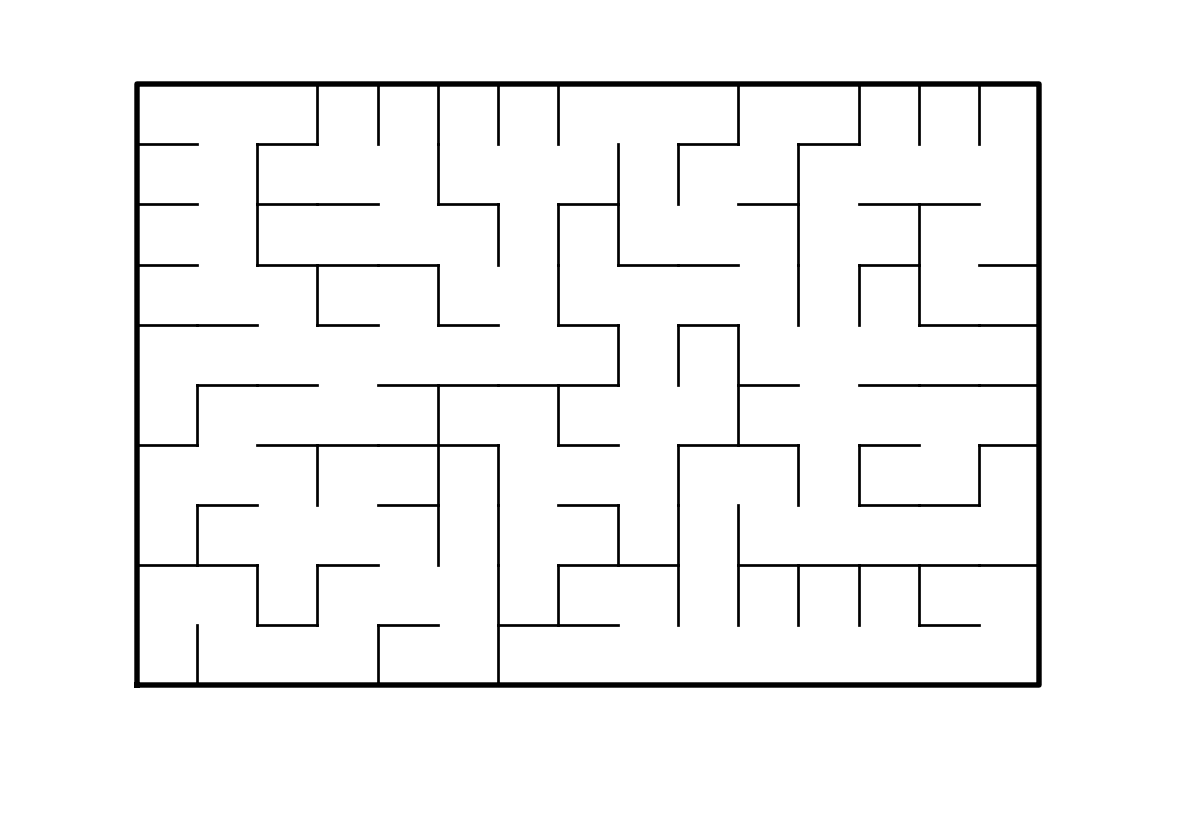
For my sample runtimes, I tested my methods each with varying sizes of rows and columns. For all union functions, the runtimes are O(1) since they use the find operations (find and find\_c). They simply look up elements by index to find the sets they belong to. The only difference in the union functions are the number of conditionals they contain which are all constant operations. For my mazes (3 total), I used size = 150, size = 1350, and size = 12150 by altering the rows and columns by a factor of 3 each time. All buildMaze funtions were timed from the start to after the maze is drawn. The buildMaze functions I created all operated in O(n) runtime. The draw\_maze function is also operating in O(n) since it is set to default false for cell\_nums. When draw\_maze is called, it never goes into the second condition because cell\_nums is false in that case. The table of the inputs and time taken to execute the functions is as shown below. The times are recorded in seconds using time.time().



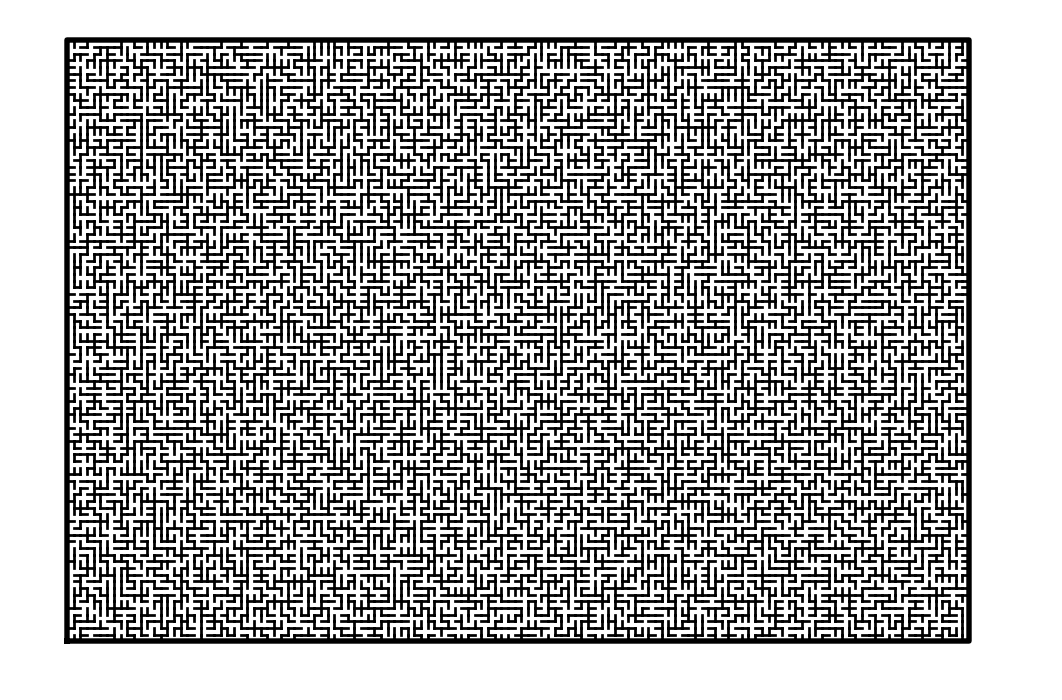
The sample console output for a maze of size 150 is shown below:



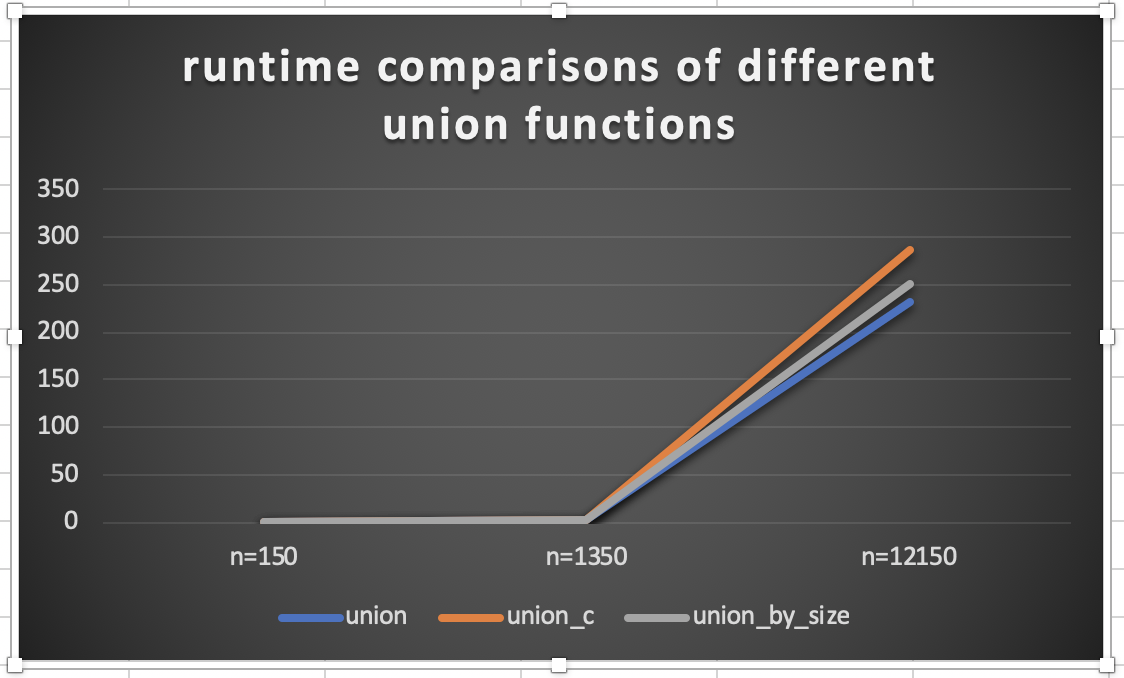
The following graph is produced by the standard union function and size of 150.



For size 12150 using union\_c:



The graphs of all 3 functions are as shown below. Note that the graphs of all 3 are very similar and almost identical up to the point n = 1350.



**Conclusion:**

This project taught me how to use disjoint set forests to implement and visualize a maze. Using the code provided helped me to be able to draw a maze and utilize the wall list in order to break selected walls. It also showed me that the runtimes of union, union\_c, and union\_by\_size are very similar since the find methods simply look up values at specified indices and the unions function all use conditionals and value reassignments to perform operations in O(1) time. It also helped me to see the difference between union functions.

**Appendix source (source code):**

#CS2302

#Nicole Favela

#last modified: April 14, 2019

#Lab6

#purpose: to build a maze using disjoint set forests and to compare runtimes of union by size and regular union functions

#instructor: Olac Fuentes

#TAs: Anindita Nath and Maliheh Zargaran

# 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 random

from matplotlib import pyplot as plt

import numpy as np

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])

#Find with path compression

def find\_c(S,i):

if S[i]<0:

return i

r = find\_c(S,S[i])

S[i] = r

return r

#counts the number of sets in the dsf

def numSets(S):

count = 0

for i in range (len(S)):

if S[i] <0 or S[i] == i:

count+=1

return count

#joins sets

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

#union by compression

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

#union by size

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

#builds maze

def buildMaze():

while numSets(S) > 1:

#d is a random integer in range of walls

d = random.randint(0, len(walls) - 1)

#wall is an list location in walls at d

wall = walls[d]

#grabs coordinates for cells

c1 = wall[0]

c2 = wall[1]

#if not in same set

if find(S,c1)!= find(S,c2):

union(S,c1,c2) #make part of same set

walls.pop(d) #remove wall

#builds maze with union with compression

def buildMaze\_withCompresssion():

while numSets(S) > 1:

# d is a random integer in range of walls

d = random.randint(0, len(walls) - 1)

# wall is an list location in walls at d

wall = walls[d]

# grabs coordinates for cells

c1 = wall[0]

c2 = wall[1]

#if not in same set

if find\_c(S,c1)!= find\_c(S,c2):

union\_c(S,c1,c2) #make part of same set

walls.pop(d) #remove wall

def buildMaze\_with\_union\_by\_size():

while numSets(S) > 1:

# d is a random integer in range of walls

d = random.randint(0, len(walls) - 1)

# wall is an list location in walls at d

wall = walls[d]

# grabs coordinates for cells

c1 = wall[0]

c2 = wall[1]

#if not in same set

if find\_c(S,c1)!= find\_c(S,c2):

union\_by\_size(S, c1, c2) #make part of same set

walls.pop(d) #remove wall

plt.close("all")

maze\_rows = 10

maze\_cols = 15

#creates dsf of 150

S = DisjointSetForest(maze\_rows\*maze\_cols)

#creates array of walls

walls = wall\_list(maze\_rows,maze\_cols)

#draws empty maze

draw\_maze(walls,maze\_rows,maze\_cols,cell\_nums=True)

#records time taken to build maze

start = time.time()

buildMaze()

draw\_maze(walls,maze\_rows,maze\_cols)

end = time.time()

print('time taken to build maze using regular union:', end-start)

#shows that the maze is one set

print('the number of sets in maze 1:')

print(numSets(S))

print()

#creates dsf for maze for union by size

S = DisjointSetForest(maze\_rows\*maze\_cols)

#creates array of walls

walls = wall\_list(maze\_rows,maze\_cols)

start = time.time()

buildMaze\_withCompresssion()

draw\_maze(walls,maze\_rows,maze\_cols)

end = time.time()

print('the number of sets in maze 2 (withCompresssion):')

print(numSets(S))

print('time taken to build maze using union by compression:', end-start)

print()

S = DisjointSetForest(maze\_rows\*maze\_cols)

#creates array of walls

walls = wall\_list(maze\_rows,maze\_cols)

start = time.time()

#builds maze using union by size

buildMaze\_with\_union\_by\_size()

draw\_maze(walls,maze\_rows,maze\_cols)

end = time.time()

print('the number of sets in maze 3 (union\_by\_size):')

print(numSets(S))

print('time taken to build maze using union by size:', end-start)

plt.show()

“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.”

-Nicole Favela