CS2302 - Data Structures

Spring 2019

Lab No.6 Report

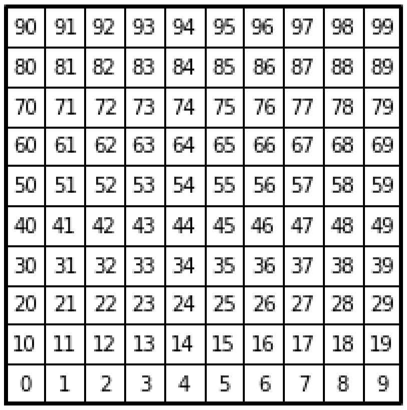
Introduction

This lab will cover the use of Disjoint Set Forest (DSF). In this structure. A disjoint-set forest consists of a number of elements each of which stores an id, a parent pointer, and either a size or a "rank" value.

In this project we made use of this data structure in order to create a randomly generated maze. The maze cells must be connected to other cells in such a matter that there exists 1 path that takes us to all the cells in the maze.

Design & Implementation

To create a randomly generated maze, DSF is the optimal data structure. The maze first starts as a grid of cells. Then we need to remove a wall from each cell until each cell is connected by one unique path.



90	91	92	93	94	95	96	97	98	99
80	81	82	83	84	85	86	87	88	89
70	71	72	73	74	75	76	77	78	79
60	61	62	63	64	65	66	67	68	69
50	51	52	53	54	55	56	57	58	59
40	41	42	43	44	45	46	47	48	49
30	31	32	33	34	35	36	37	38	39
20	21	22	23	24	25	26	27	28	29
10	11	12	13	14	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9

Figure 1: Starting grid for the maze.

Union by Size with Path Compression:

Union by size with path compression refers to a method that unites the root of a smaller DSF to the larger one. Making the united tree point directly at the index where the root is located instead of a child index.

Standard Union:

Each cell can be represented as its own DSF root. Once we unite two cells it means the root of the united cell will now point to the root of the cell to which it was united. When each cell belongs to a single set then it will mean our maze will be completed.

For this there will be a list of all the walls in our grid. Once the method selects a random wall it will then proceed to find their roots in the DSF. If the roots are different then the root of the cells will be united, and the wall will be removed from the list.

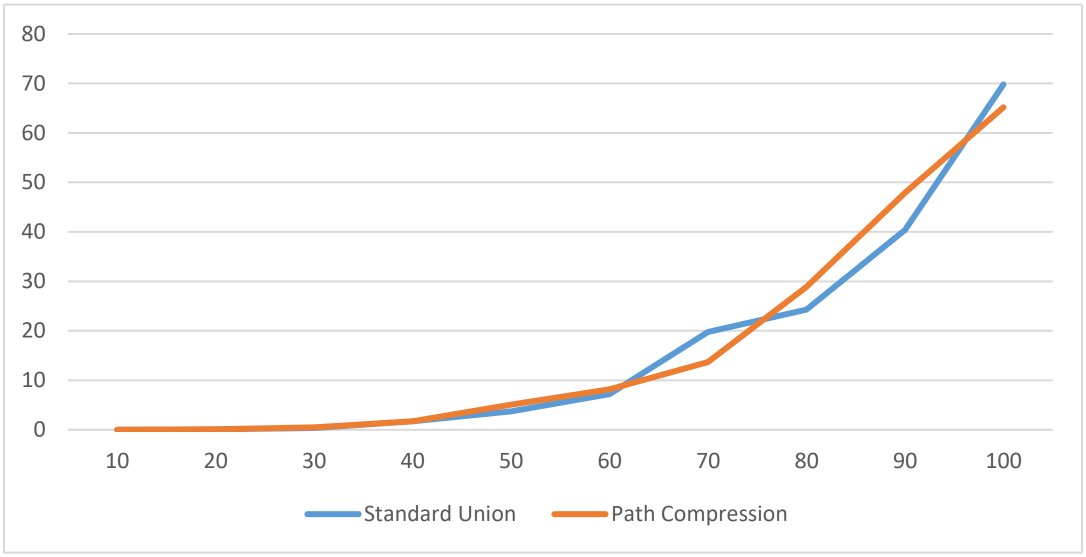
Design:

Both methods follow the same bases, with the only difference being how one calls a method to perform a union by size and the other a call to a method with standard union.

There is a user interface that asks the user for input when choosing the size of the maze and which union to perform.

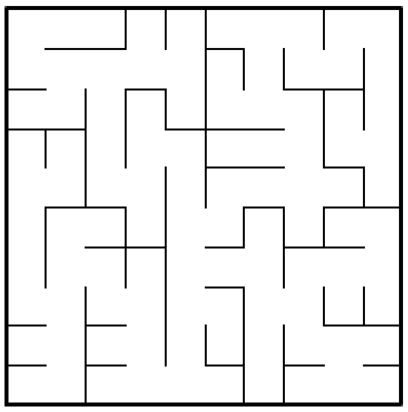
Running Times

Different tests were performed in order to obtain the running times of each type of union took to complete.

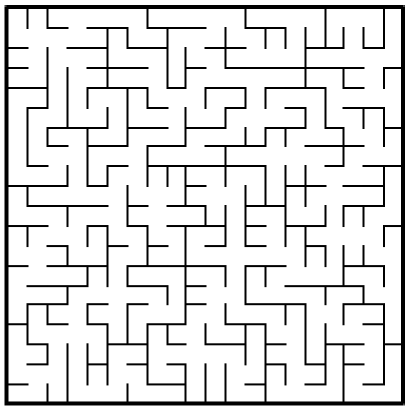


Outputs

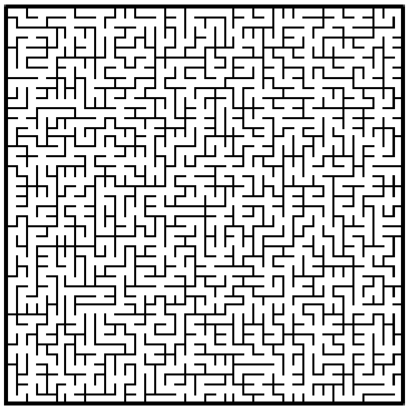
```
Choose the number cells for the length and height of the maze: 10
1 for normal DSF or 2 for compressed DSF:

Choice: 1
Using standard union and no compression, time: 0.004013538360595703

Drawing maze...
```



```
Choose the number cells for the length and height of the maze: 20
1 for normal DSF or 2 for compressed DSF:

Choice: 1
Using standard union and no compression, time: 0.07483506202697754

Drawing maze...
```



```
Choose the number cells for the length and height of the maze: 40
1 for normal DSF or 2 for compressed DSF:

Choice: 2
Using compression and union by size, time: 1.400815725326538

Drawing maze...
```



Appendix

```
from scipy import interpolate

import matplotlib.pyplot as plt

import numpy as np

import random

import time


"""

Disjoint Set Forest ---------------------------------------------------------

"""

def DisjointSetForest(size):

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



def dsfToSetList(S):

    #Returns aa list containing the sets encoded in S

    sets = [ [] for i in range(len(S)) ]

    for i in range(len(S)):

        sets[find(S,i)].append(i)

    sets = [x for x in sets if x != []]

    return sets


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 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_dsf(S):
    scale = 30
    fig, ax = plt.subplots()
    for i in range(len(S)):
        if S[i]<0: # There is a root
            ax.plot([i*scale,i*scale],[0,scale],linewidth=1,color='k')
            ax.plot([i*scale-1,i*scale,i*scale+1],[scale-2,scale,scale-2],linewidth=1,color='k')
        else:
            x = np.linspace(i*scale,S[i]*scale)
            x0 = np.linspace(i*scale,S[i]*scale,num=5)
            diff = np.abs(S[i]-i)
            if diff == 1:
                y0 = [0,0,0,0,0]
            else:
                y0 = [0,-6*diff,-8*diff,-6*diff,0]
            f = interpolate.interp1d(x0, y0, kind='cubic')
            y = f(x)
            ax.plot(x,y,linewidth=1,color='k')
            ax.plot([x0[2]+2*np.sign(i-S[i]),x0[2],x0[2]+2*np.sign(i-S[i])],[y0[2]-
1,y0[2],y0[2]+1],linewidth=1,color='k')
        ax.text(i*scale,0, str(i), size=20,ha="center", va="center",
         bbox=dict(facecolor='w',boxstyle="circle"))
    ax.axis('off')
    ax.set_aspect(1.0)


def num_of_sets(S): #return number of roots in dsf
    num = 0
    for i in range(len(S)):
        if S[i] < 0:
            num += 1
    return num

"""
Maze Functions ---------------------------------------------------------------
"""
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 position
            x0 = (w[1]%maze_cols)
            x1 = x0
            y0 = (w[1]//maze_cols)
            y1 = y0+1
        else: # Horizontal Wall postion
            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)
```

```
# Creates a list with all the walls in the maze
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: # If not last column
                w.append([cell,cell+1]) # Wall between adjacent columns
            if r!=maze_rows-1: # If not last row
                w.append([cell,cell+maze_cols]) # Wall between adjacent rows
    return w

def Maze_normal(r,c,S,W):
    while num_of_sets(S) > 1:
        d = random.randint(0,len(W)-1) # Random index in wall list
        if find(S,W[d][0]) != find(S,W[d][1]): # Check if the roots are different
            union(S,W[d][0],W[d][1]) # Join the sets
            W.pop(d) # Delete wall

def Maze_C(r,c,S,W):
    while num_of_sets(S) > 1:
        d = random.randint(0,len(W)-1) # Random index in wall list
        if find_c(S,W[d][0]) != find_c(S,W[d][1]): # Use path compression
            union_by_size(S,W[d][0],W[d][1])
            W.pop(d)

"""
Main ------------------------------------------------------------------------
"""
plt.close("all")
```

```
c = int(input("Choose the number cells for the length and height of the maze: "))
W = wall_list(c,c)
S = DisjointSetForest(c * c)

choice = 0
print("1 for normal DSF or 2 for compressed DSF:")
while 1: # Continue until 1 or 2 is input
    try:
        choice = int(input("Choice: "))
        if choice == 1 or choice == 2:
            break
        else:
            print("Choose 1 or 2")
    except:
        print("Choose 1 or 2")

if choice == 1:
    start = time.time()
    Maze_normal(c,c,S,W)
    end = time.time()
    print("Using standard union and no compression, time:",end - start)

if choice == 2:
    start = time.time()
    Maze_C(c,c,S,W)
    end = time.time()
    print("Using compression and union by size, time:",end - start)

print("\nDrawing maze...")
draw_maze(W,c,c)
```

I certify that this essay is original work prepared by me, Jesus A Hernandez.