CS2302 Lab 6 Report – Andres Silva

Disjoint ser forest (DSFs) can serve as a representation of paths and steps, in this lab, this was taken literally in the form of the implementation of a maze. By creating a grid of n x m size and deleting random lines (walls) of said grid, a path might be formed resulting in a maze, but that will rarely be the case. To ensure that an actual path exists between the cells of the grid, a DSF can be used to represent each square (cell) of the gird.

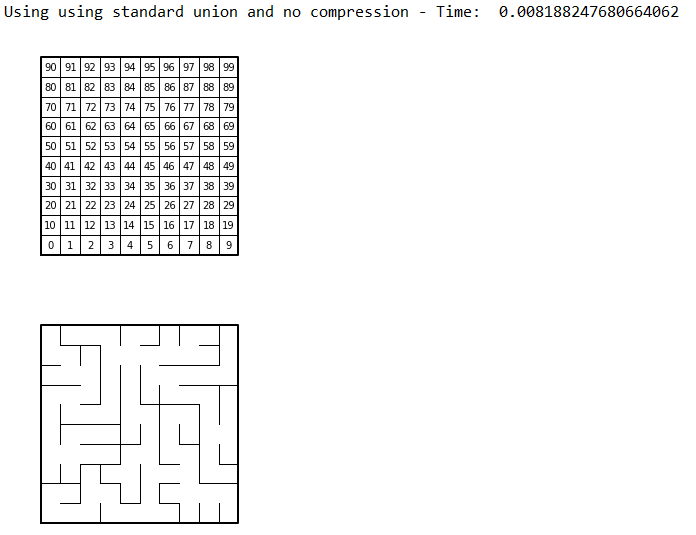
By representing each cell as a set and a connection between two cells the union of two sets, it will then be possible to guarantee a path between all cells and thus have a solvable maze. This was achieved by randomly removing walls of cells that belong to different sets until a singular set was left.

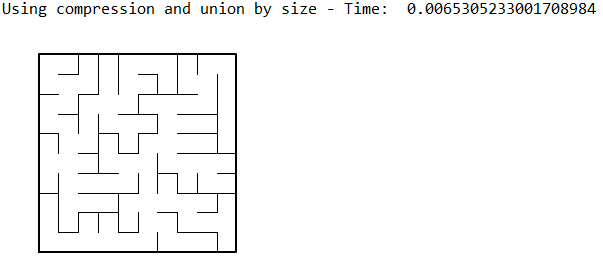
In more detail, the DSF class contained a method that counts the roots of a DSF, it is known that a DSF has as many sets as roots, which are always negative, so the method counts the negative elements in the provided DSF. Then a while loop, which executes if there is more than one set, will select a random wall and check if the cells of the wall belong to different sets. If they do belong to different sets, then the sets are joined, and the wall deleted for the wall list.

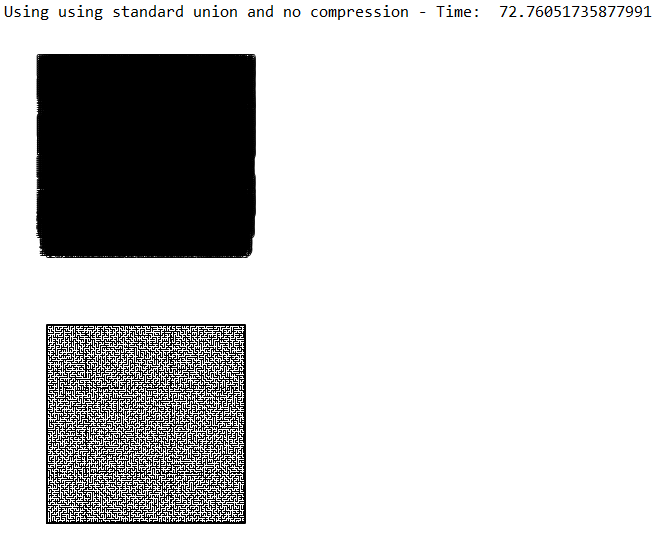
A second method with the same function was created, but one implementing path compression and union by size, for time comparisons. These comparisons were performed by making the size of the maze n x n and starting from n = 10 to n =100 and recording the time it took the DSF to reach a singular set.

Here is the time comparison between the two methods, which might resemble O(n^2) for both.

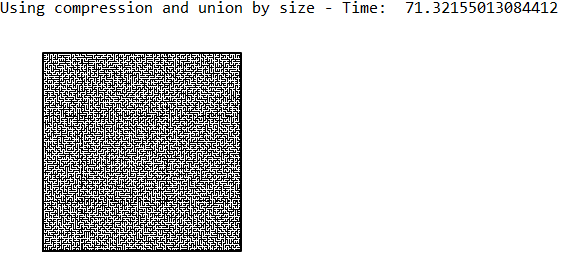
**Outputs:**

n = 10.



n = 100.

This is the initial numerated gird with 1000 sets



**Academic Honesty Statement**

“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.”

* Andres Silva.

**Appendix**Top of Form

Bottom of Form

Top of Form

Bottom of Form

|  |  |
| --- | --- |
|  | # -\*- coding: utf-8 -\*- |
|  | """ |
|  | Created on Mon Apr 8 22:29:30 2019 |
|  |  |
|  | @author: andre |
|  | """ |
|  |  |
|  | # 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 matplotlib.pyplot as plt |
|  | import numpy as np |
|  | import random |
|  | from dsf import \* |
|  | import time |
|  |  |
|  | 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: #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 |
|  | if find(S,W[d][0]) != find(S,W[d][1]): #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 |
|  | 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]) #and union by size |
|  | W.pop(d) |
|  |  |
|  | plt.close("all") |
|  |  |
|  | r = 100 #Define number of rows |
|  | c = r #Define number of columns |
|  |  |
|  | W = wall\_list(r,c) |
|  | S = DisjointSetForest(r \* c) |
|  |  |
|  |  |
|  | #draw\_maze(W,r,c,cell\_nums=True) |
|  |  |
|  | start = time.time() |
|  |  |
|  | #Maze\_normal(r,c,S,W) |
|  | Maze\_C(r,c,S,W) |
|  |  |
|  | end = time.time() |
|  |  |
|  | #print("Using using standard union and no compression - Time: ", end - start) |
|  | print("Using compression and union by size - Time: ", end - start) |
|  |  |
|  | draw\_maze(W,r,c) |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |