

Lab 6

University of texas at el paso

Javier Soto | CS2302 | 10:30 AM

# INTRODUCTION

The purpose of lab 6 is to implement a disjoint set Forrest in the creation of a maze to ensure that there is only one path from beginning to end. We then need to alter our implementation of the disjoint set Forrest to optimize run time and compare to the original implementation.

## Proposed solution

My solution to this problem was to create a disjoint set Forrest where each index of the Forrest represents a cell in the maze. The Forrest will be of size equal to the size of the maze with every index being a root when initialized. Every time a wall is deleted form the maze, I will union the two cells adjacent to the wall. Once all cells belong to the same set the deleting of wall will cease.

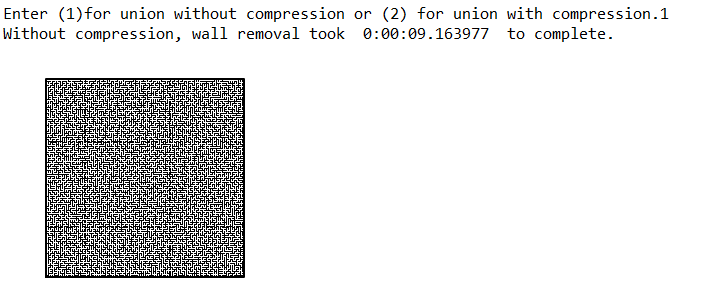
For the deletion of self, I have implemented a loop that checks if all items are in the same set. To check if all items are in the same set I simply count the number of sets by iteration though the Forrest and counting roots. This process takes O(n) every time it executes to improve this I can implement an attribute to the disjoint set Forrest object called numSets that keeps track of the number of sets in the Forrest so that I do not need to cost linear time for every check.

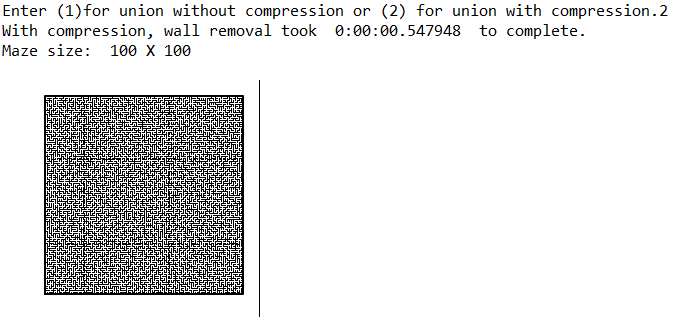
While there still exists more than one set in the Forrest, a random wall is selected. If the adjacent cells are not in the same sets then the indexes of the Forrest are unioned with wither method mentioned above and the wall is deleted. If the adjacent cells are already in the set, then nothing is done and the loop continues. At the end of the loop if the size of the maze is smaller than 500x500 then the maze is drawn using professor Fuentes draw method.

To implement compression, the union with compression method will always use the find with compression method. In the find with compression method every time a index is visited, if the content of that index is not a root or pointing to a root then the root of that index is found and the index is now set to point at the root. As my experimental results will show when the paths are compressed the run time of the programs are dramatically improved.

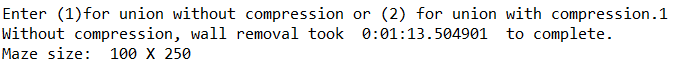
## Experimental results

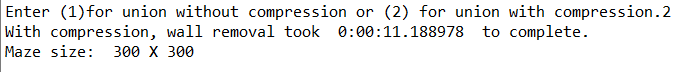
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 400 | 2500 | 10K | 25k | 90k |
| without compression | 0.0156 | 0.5203 | 9.1639 | 73.5049 |  |
| with compression | 0.0155 | 0.078 | 0.5479 | 1.6383 | 11.1889 |











## Appendix

**# -\*- coding: utf-8 -\*-**

**"""**

**Created on Mon Apr 8 17:56:23 2019**

**@author: yatha**

**"""**

**# 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 random**

**import dsf**

**from datetime import datetime as dt**

**def remove\_walls\_c(W, sets,Print):**

**numSets= dsf.num\_of\_sets(sets)**

**start = dt.now()**

**while numSets > 1 and W is not None:**

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

**#print(numSets)**

**if (dsf.in\_same\_set(sets,walls[d][0],walls[d][1])) == False:**

**dsf.union\_c(sets,walls[d][0],walls[d][1])**

**numSets -= 1**

**walls.pop(d)**

**stop = dt.now() -start**

**if Print == True:**

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

**print('With compression, wall removal took ', stop,' to complete.')**

**def remove\_walls(W, sets,Print):**

**numSets= dsf.num\_of\_sets(sets)**

**start = dt.now()**

**while numSets > 1 and W is not None:**

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

**#print(numSets)**

**if (dsf.in\_same\_set(sets,walls[d][0],walls[d][1])) == False:**

**dsf.union(sets,walls[d][0],walls[d][1])**

**numSets -= 1**

**walls.pop(d)**

**stop = dt.now() -start**

**if Print == True:**

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

**print('Without compression, wall removal took ', stop,' to complete.')**

**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 = 50**

**maze\_cols = 50**

**Print = False**

**if maze\_rows < 500 and maze\_cols <500:**

**Print = True**

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

**sets = dsf.DisjointSetForest(maze\_rows\*maze\_cols)**

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

**try:**

**choice = int(input("Enter (1)for union without compression or (2) for union with compression."))**

**if choice ==1:**

**remove\_walls(walls,sets,Print)**

**elif choice == 2:**

**remove\_walls\_c(walls,sets,Print)**

**else:**

**print('invalid input')**

**except ValueError:**

**print("exception reached")**

**print()**

**print("Maze size: " , maze\_rows, "X", maze\_cols)**

**# -\*- coding: utf-8 -\*-**

**"""**

**Created on Thu Apr 11 09:41:51 2019**

**@author: yatha**

**"""**

**# Implementation of disjoint set forest**

**# Programmed by Olac Fuentes**

**# Last modified March 28, 2019**

**import matplotlib.pyplot as plt**

**import numpy as np**

**from scipy import interpolate**

**def num\_of\_sets(S):**

**count =0**

**for i in range(len(S)):**

**if S[i]<0:**

**count +=1**

**return count**

**def in\_same\_set(S,i,j):**

**return find(S,i) == find(S,j)**

**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: # i 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: #i and S[i] are neighbors; draw straight line**

**y0 = [0,0,0,0,0]**

**else: #i and S[i] are not neighbors; draw arc**

**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)**

**'''**

**if \_\_name\_\_ == "\_\_main\_\_":**

**plt.close("all")**

**S = DisjointSetForest(8)**

**print(S)**

**draw\_dsf(S)**

**union(S,7,6)**

**print(S)**

**draw\_dsf(S)**

**union(S,0,2)**

**print(S)**

**draw\_dsf(S)**

**union(S,6,3)**

**print(S)**

**draw\_dsf(S)**

**union(S,5,2)**

**print(S)**

**draw\_dsf(S)**

**union(S,4,6)**

**print(S)**

**draw\_dsf(S)**

**print('Sets encoded by DSF:',dsfToSetList(S))**

**T = DisjointSetForest(8)**

**union(T, 7 , 0 )**

**union(T, 1 , 6 )**

**union(T, 3 , 0 )**

**union(T, 0 , 6 )**

**union(T, 3 , 4 )**

**union(T, 2 , 5 )**

**union(T, 6 , 0 )**

**union(T, 0 , 3 )**

**union(T, 4 , 2 )**

**union(T, 1 , 7 )**

**print(T)**

**draw\_dsf(T)**

**print('Sets encoded by DSF:',dsfToSetList(T))**

**U = DisjointSetForest(8)**

**for i in range(len(U)):**

**union(U, i , 0 )**

**print(U)**

**draw\_dsf(U)**

**Uc = DisjointSetForest(8)**

**for i in range(len(Uc)):**

**union\_c(Uc, i , 0 )**

**print(Uc)**

**draw\_dsf(Uc)**

**Us = DisjointSetForest(8)**

**for i in range(len(Us)):**

**union\_by\_size(Us, i , 0 )**

**print(Us)**

**draw\_dsf(Us)**

**'''**

Academic dishonesty

I, Javier Soto, certify that this script and lab report are of my own unless otherwise documented above.

