Tyler Salas

1:30-2:50 Class

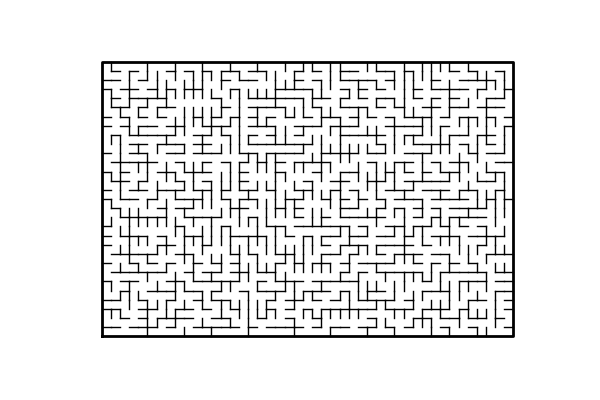
Dr. Fuentes

80596221

Lab 6 Report

Lab 6, the lab assigned this week, aimed to implement the use of disjoint set forests. All we had to do was follow the psuedo-code shared in the lab sheet on the class website. Dr. Fuentes also provided us with code that makes an mxn box that contains cells for each number. In the code shared, a random wall would be removed in the range of half the size of the list. This would create a box with random sections of walls removed.

The idea was to make the provided code create a maze where all cells in the image will be linked together, thus creating a maze that one can visit all cells and create a path from start to finish. I simply followed the psuedocode that was provided and ended with the image of a maze.￼



I start with making a list of walls, this is what the draw\_maze function will take and what will be modified to create the maze. The size of the maze can be changed by modifying the variables maze\_rows and maze\_cols in lines 101 and 102. A disjoint set forest is then created to be the size of the number of cells. A while loop is then created that checks the method moreOneSet of S the created DSF. What this method does is kep track of a count variable while it iterates through the DSF. If count is ever more than one, the method terminates and True is returned.

Inside the while loop, a random wall is chosen and declared to c1. An if statement then uses the find method on both the first and second elements of that wall. This finds the root of both cells in the DSF. If they are not equal to each other, the if statement executes with the following. It will first union both members of the list at point c1 and will pop that wall, thus reducing the amount of sets by one and creating a path from one adjacent cell to the other. The draw\_maze function is then called and the maze is created.

Regular – Uses the find and union method with no modifications

Modified – Uses find with compression and union by size

**Run Times 10x15**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| Regular | 0.0099728107452392 | 0.0119671821594238 | 0.0109703540802008 |
| Modified | 0.0129649639129638 | 0.0120017528533935 | 0.0109698772430419 |

**Run Times 20x30**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| Regular | 0.2124304771423339 | 0.277271032333374 | 0.2114875316619873 |
| Modified | 0.1944489479064941 | 0.1745004653930664 | 0.2084429264068605 |

**Run Times 30x45**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Test 1 | Test 2 | Test 3 |
| Regular | 1.3873121738433838 | 1.0313031673431396 | 1.0561771392822266 |
| Modified | 1.4034206867218018 | 1.1102344989776611 | 1.1519241333007812 |

As we can see, when we look at the statistics given to us by the run times in the tables above, it seems that there doesn’t seem to be such a large variation in the amount of time it takes to create each list. The while loop that runs the majority of work done would be a O(n) because it must go through each new set in the declared DSF. The moreOneSet function that the while loop checks is also O(n). The find function that is used is a O(n) as it has the possibility of iterating the whole array, the same is true for find with compression as long as it is a new element, though it would help if more operations were done later with the same DSF.

Overall, this lab helped me to understand Disjoint Set Forests and it was quite pleasurable to see the end result of the maze.

# 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 time

import matplotlib.pyplot as plt

import numpy as np

import random

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])

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 find\_c(S,i):

if S[i] <= 0:

return i

s = i

while S[i] >= 0:

i = S[i]

root = i

while S[s] >= 0:

p = S[s]

S[s] = root

s = p

return root

'''

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 union(S,i,j):

# Joins i's tree and j's tree, if they are different

ri = find\_c(S,i)

rj = find\_c(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

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 moreOneSet(S):

count = 0

for i in range(len(S)):

if S[i] < 0:

count += 1

if count > 1:

return True

return False

def sameSet(S,i,j):

i = find\_c(S,i)

#print(i)

j = find\_c(S,j)

#print(j)

if i == j:

return True

return False

plt.close("all")

maze\_rows = 30

maze\_cols = 45

start = time.time()

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

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

while moreOneSet(S):

c1 = random.randint(0,len(walls)-1)

if find\_c(S,walls[c1][0]) != find\_c(S,walls[c1][1]):

union\_by\_size(S,walls[c1][0],walls[c1][1])

walls.pop(c1)

end = time.time()

print(end-start)

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

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.



-Tyler Salas