**Introduction**

**Task is to create a maze in which there is exactly one unique path to each cell and their will always be a path from start to destination. Our task is to remove the walls in order to create that maze.**

**Conclusion**

The union method takes considerably longer than the union by size compression because each time it compresses it takes considerably shorter to get to the root

**Academic Certification statement**

I certify that the attached written or electronically transmitted material is my own work. I further certify that (a) I have formally cited or otherwise fully acknowledged the quotations, ideas, and wording used here from other sources, whether published or unpublished, in written or electronic form, (b) I have engaged in no falsification or misrepresentation of data or experience in this submission, (c) I have disclosed any collaboration not specifically authorized by the instructor.

I understand that the relative autonomy of the legal profession carries with it special obligations of self-regulation as outlined in the Columbia Law School Bulletin and that any violation of this certification will subject me to discipline, including possible suspension or expulsion by the Law School, declination to certify for admission to the Bar, and sharing of information about discipline with Bar admissions committees. I further understand that the Law School is required to notify Bar admissions committees if a student has been the subject of disciplinary proceedings, regardless of the outcome of those proceedings.

**Apendix**

# 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

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

###################################################################################

# Implementation of disjoint set forest

# Programmed by Olac Fuentes

# Last modified March 28, 2019

from scipy import interpolate

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))

###################################################################################

plt.close("all")

maze\_rows = 10

maze\_cols = 15

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

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

for i in range(len(walls)//2): #Remove 1/2 of the walls

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

print('removing wall ',walls[d])

walls.pop(d)

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