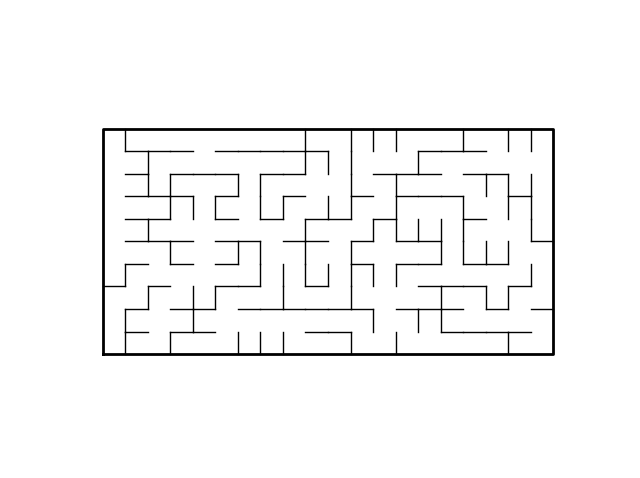
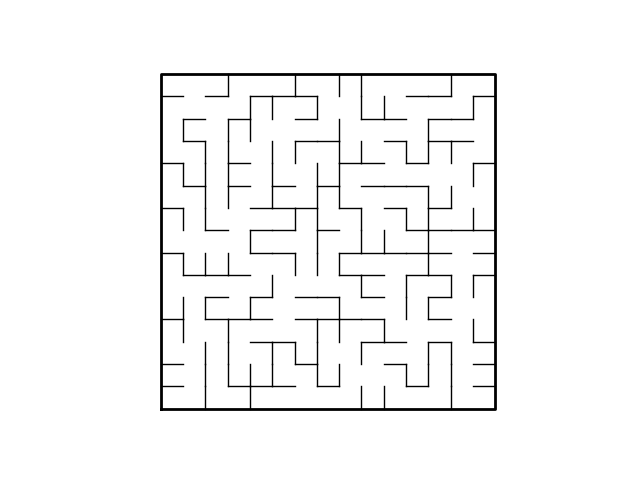
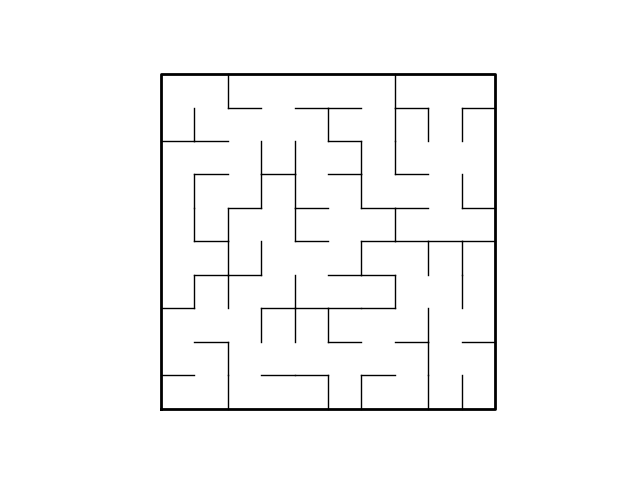
This lab asked of us to modify code given to us by professor Fuentes to create mazes at random using Disjoint Set Forests, or DSF for short. We had to modify the code to create a completely solvable maze using either standard union or union by size which takes makes the smaller list point to the larger list.

This was a very simple task to do. I created a while loop instead of the for loop that was there initially and made the while loop run until there was only one set in the entire maze and print the new maze. I also added an if statement that would prevent walls of items in the same set from being removed to avoid getting a mostly blank maze or random open patches with no walls.

Testing this was quick and simple, the majority of runtimes took less than a second to output images that allowed me to see if it was properly working, and I would manually change the maze size to be different shapes and sizes, say a 10x10, 15x10, 20x20 and so on. The below images are 10x20, 15x15, and 10x10 mazes created by the program.

I also added a small menu that allows the user to create a maze using normal union and union by size if they want a choice.

Consistently the runtime of Union by size would run in less than one hundredth of a second, however there was one time in my testing in which it took nearly a fifth of a second to complete, likely due to random choosing many pairs within the same set, which is unlikely but can occur, and the normal form of union took roughly double the time of the Union by size form creating the charts below. The y axis is the time in seconds and the x is which trial I ran.

Appendix----

"""

Course 2302(Data Structures)

Instructor: Olac Fuentes

Teaching Assistant: Anithdita Nath, Mali Zargaran

Lab 5

Last Edited on 4.11.2019

@author: Seth

Code for drawing mazes: Olac Fuentes

"""

import matplotlib.pyplot as plt

import numpy as np

import random

import time

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\_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 = 10

maze\_cols = 10

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

#walls is a 138 x2 matrix

# [[0,1]

# [0,15]

# [1,2]]ETC.

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

choice = 0

choice = input('1.Union by size with compression\n2.Normal union\nSelection: ')

if int(choice) == 2:

print('You have selected normal union')

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

#constructs maze via normal union

start = time.time()

while len(dsfToSetList(S)) != 1: #Remove walls until they are all in one set

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

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

if find(S, (walls[d])[0]) != find(S, (walls[d])[1]):

union(S, (walls[d])[0], (walls[d])[1])

walls.pop(d)

end = time.time()

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

print('Time to create maze: ', end - start)

# numS=dsfToSetList(S)

# print(numS)

# print(len(numS))

if int(choice) == 1:

print('You have selected Union by size and compression')

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

#constructs maze via union by size and compression

start = time.time()

while len(dsfToSetList(S)) != 1: #Remove walls until there is only one set

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

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

if find(S, (walls[d])[0]) != find(S, (walls[d])[1]):#if points are already in same set do not remove

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

walls.pop(d)

end = time.time()

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

print('Time to create maze: ', end - start)

# numS=dsfToSetList(S)

# print(numS)

# print(len(numS))

“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.”

- Seth Abel Flores