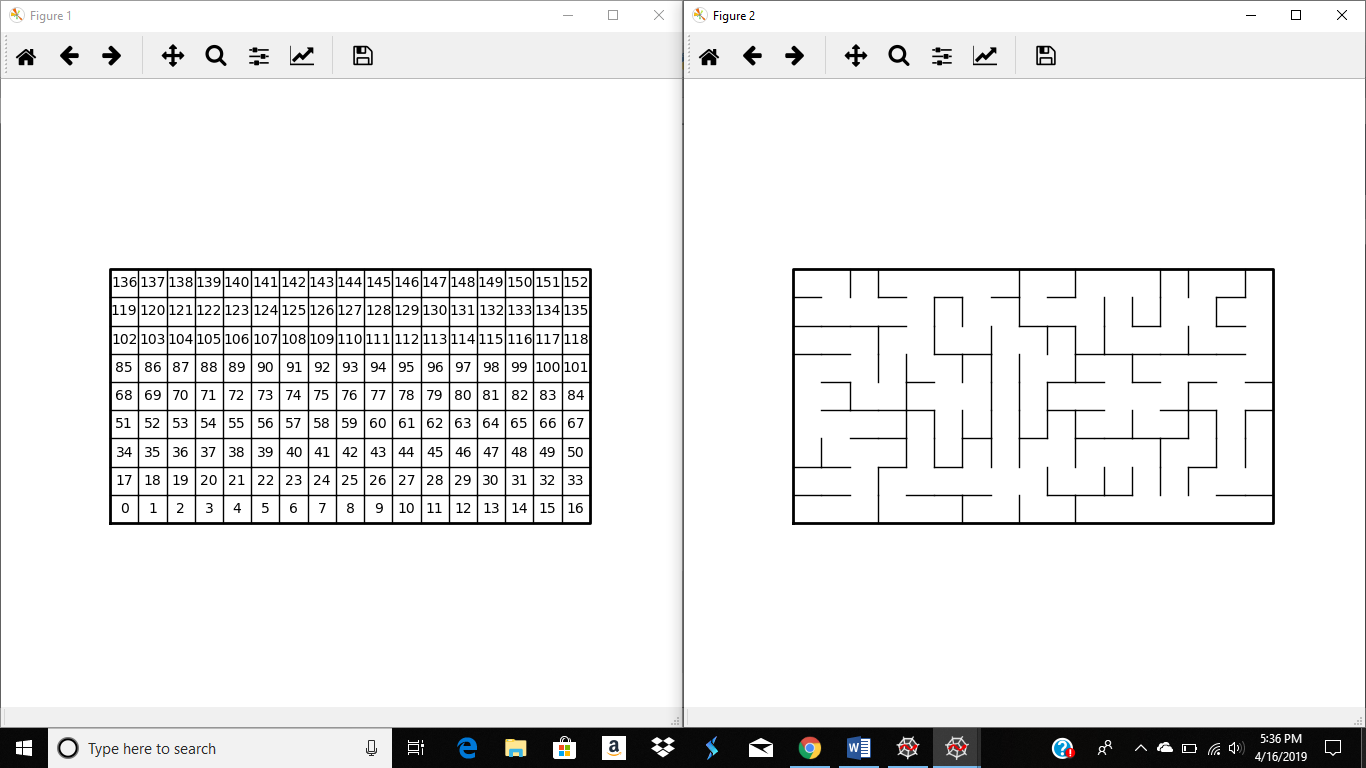
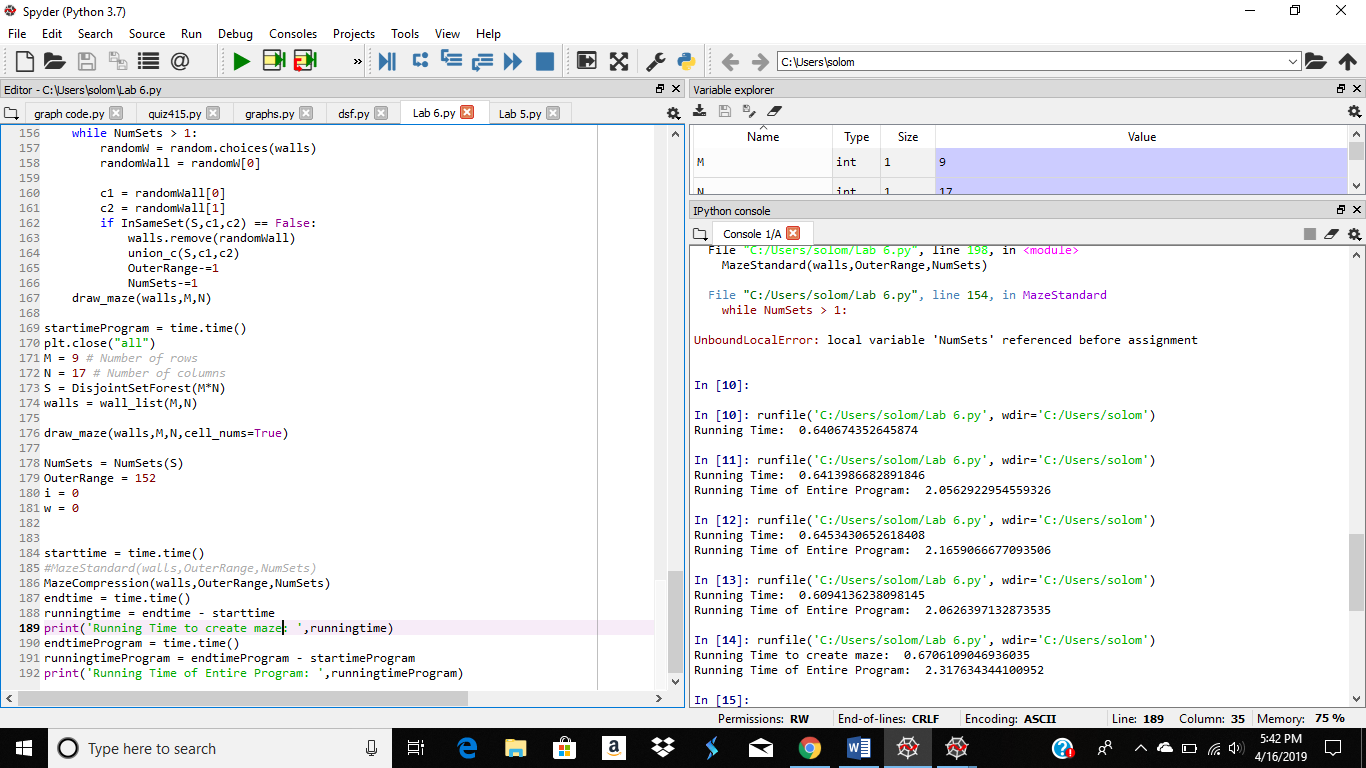
Solomon Davis Lab 6 Report

CS 2302 - MW 1:30 Spring 2019

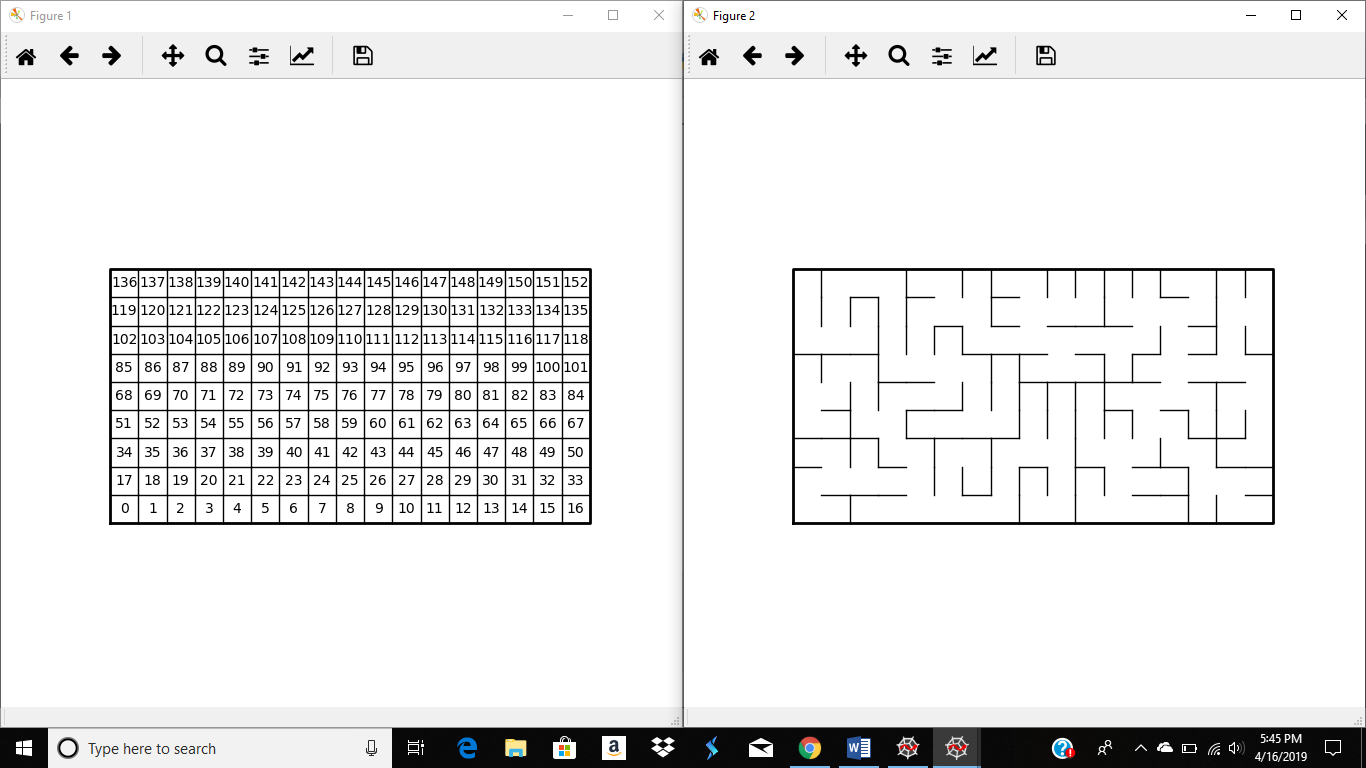
In this lab we were supposed to randomly choose a wall from the maze and remove the wall from the maze. This is supposed to be done until the number of sets in the disjoint forest was 1. To carry this task out you need to decrease the number of sets and number range for the random wall variable. After this was done the runtime for the creation of the maze using the standard union and union compression. This can be seen the pictures below.

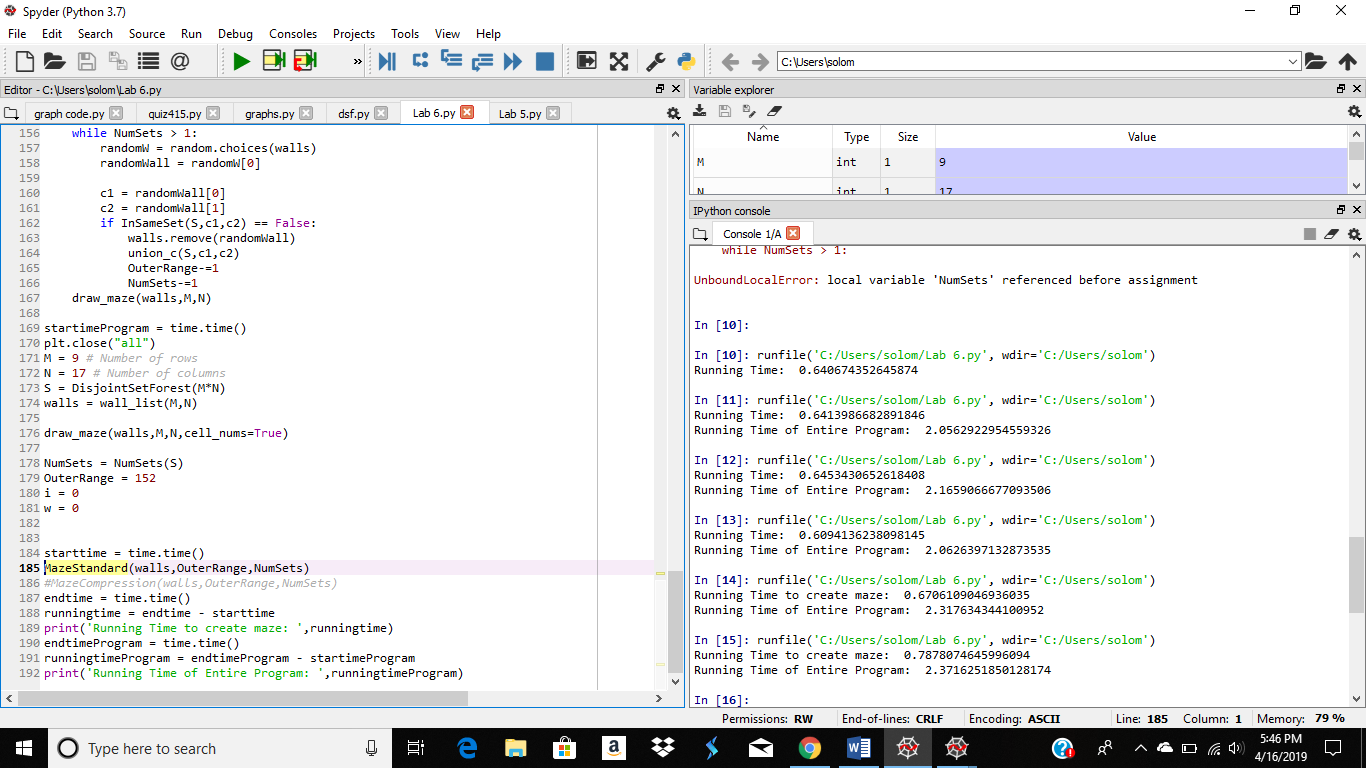
Before and after picture of maze using compression with runtime





Before and after picture using the standard union with runtime





#Course: CS2302 - Spring 2019

#Author: Solomon Davis

#Lab Number: 6

#Instructor: Olac Fuentes

#Last Modified: April 16, 2019

#Due Date: April 12, 2019

#Description: This Lab will create a maze depending on a random wall chosen in

# in the full maze. This will be done until there is exactly one set in the

#disjoint forest.

import matplotlib.pyplot as plt

import numpy as np

import random

import time

from scipy import interpolate

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 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: # Do nothing if i and j belong to the same set

S[rj] = ri # Make j's root point to i's root

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 InSameSet(S,a,b):

ri = find(S,a)

rj = find(S,b)

if ri == rj:

return True

return False

def NumSets(S):

count = 0

for i in range(len(S)):

if S[i]<0:

count+=1

return count

def SRoot(S,k):

for i in range(len(S)):

if S[i] == k:

return i

return SRoot(S[i],k)

def Singleton(S,i):

if S[i] != -1:

return False

return True

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)

def draw\_maze(walls,M,N,cell\_nums=False):

fig, ax = plt.subplots()

for w in walls:

if w[1]-w[0] ==1: #vertical wall

x0 = (w[1]%N)

x1 = x0

y0 = (w[1]//N)

y1 = y0+1

else:#horizontal wall

x0 = (w[0]%N)

x1 = x0+1

y0 = (w[1]//N)

y1 = y0

ax.plot([x0,x1],[y0,y1],linewidth=1,color='k')

sx = N

sy = M

ax.plot([0,0,sx,sx,0],[0,sy,sy,0,0],linewidth=2,color='k')

if cell\_nums:

for r in range(M):

for c in range(N):

cell = c + r\*N

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(M,N):

# Creates a list with all the walls in the maze

w =[]

for r in range(M):

for c in range(N):

cell = c + r\*N

if c!=N-1:

w.append([cell,cell+1])

if r!=M-1:

w.append([cell,cell+N])

return w

def MazeCompression(walls,OuterRange,NumSets):

#creates the maze using standard union

while NumSets > 1:

randomW = random.choices(walls)

randomWall = randomW[0]

c1 = randomWall[0]

c2 = randomWall[1]

if InSameSet(S,c1,c2) == False:

walls.remove(randomWall)

union\_c(S,c1,c2)

OuterRange-=1

NumSets-=1

draw\_maze(walls,M,N)

def MazeStandard(walls,OuterRange,NumSets):

#creates the maze using union compression

while NumSets > 1:

randomW = random.choices(walls)

randomWall = randomW[0]

c1 = randomWall[0]

c2 = randomWall[1]

if InSameSet(S,c1,c2) == False:

walls.remove(randomWall)

union\_c(S,c1,c2)

OuterRange-=1

NumSets-=1

draw\_maze(walls,M,N)

startimeProgram = time.time()

plt.close("all")

M = 9 # Number of rows

N = 17 # Number of columns

S = DisjointSetForest(M\*N)

walls = wall\_list(M,N)

draw\_maze(walls,M,N,cell\_nums=True)

NumSets = NumSets(S)

OuterRange = 152

i = 0

w = 0

starttime = time.time()

MazeStandard(walls,OuterRange,NumSets)

#MazeCompression(walls,OuterRange,NumSets)

endtime = time.time()

runningtime = endtime - starttime

print('Running Time to create maze: ',runningtime)

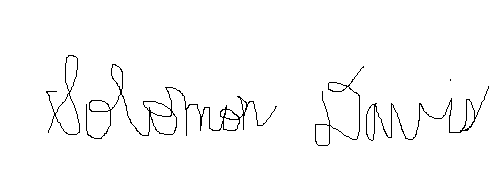
endtimeProgram = time.time()

runningtimeProgram = endtimeProgram - startimeProgram

print('Running Time of Entire Program: ',runningtimeProgram)

Academic Service Certificate:

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.



Solomon Davis