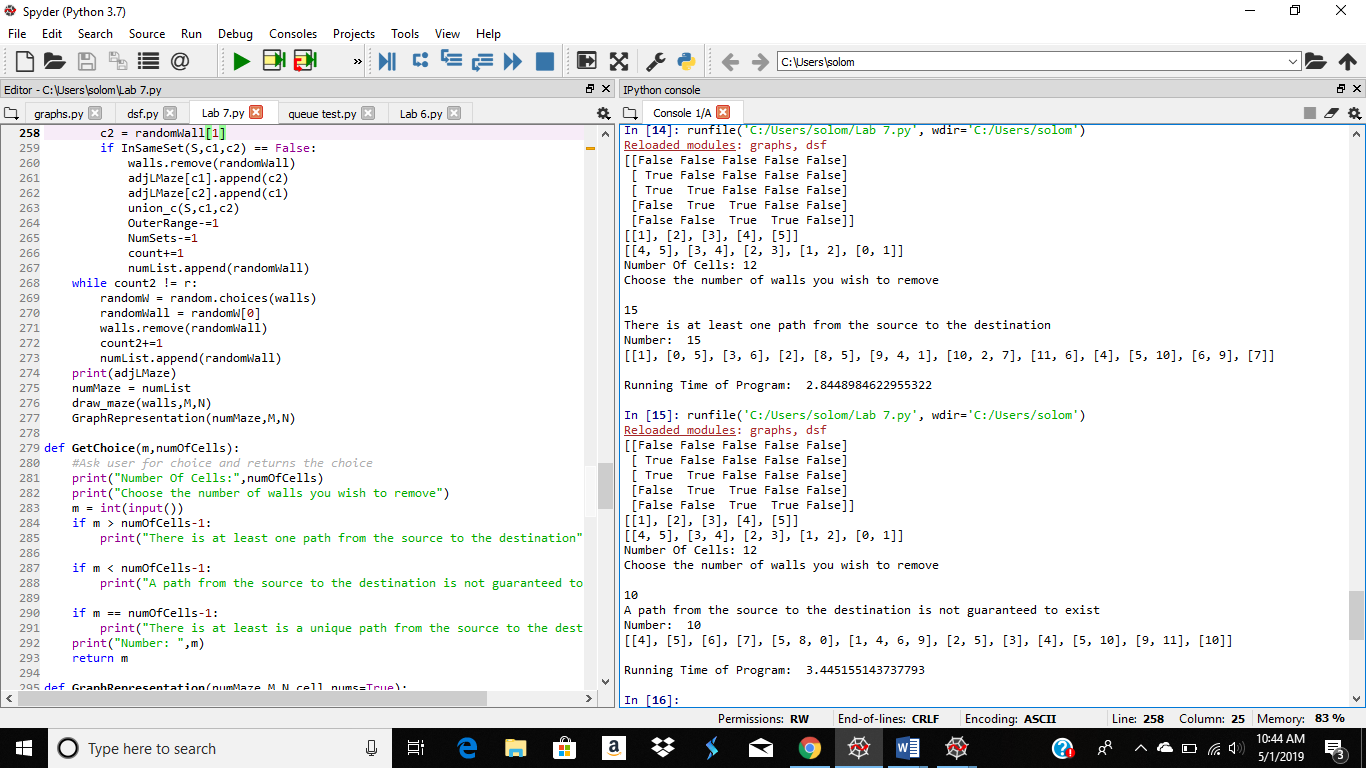
Solomon Davis Lab 7 Report

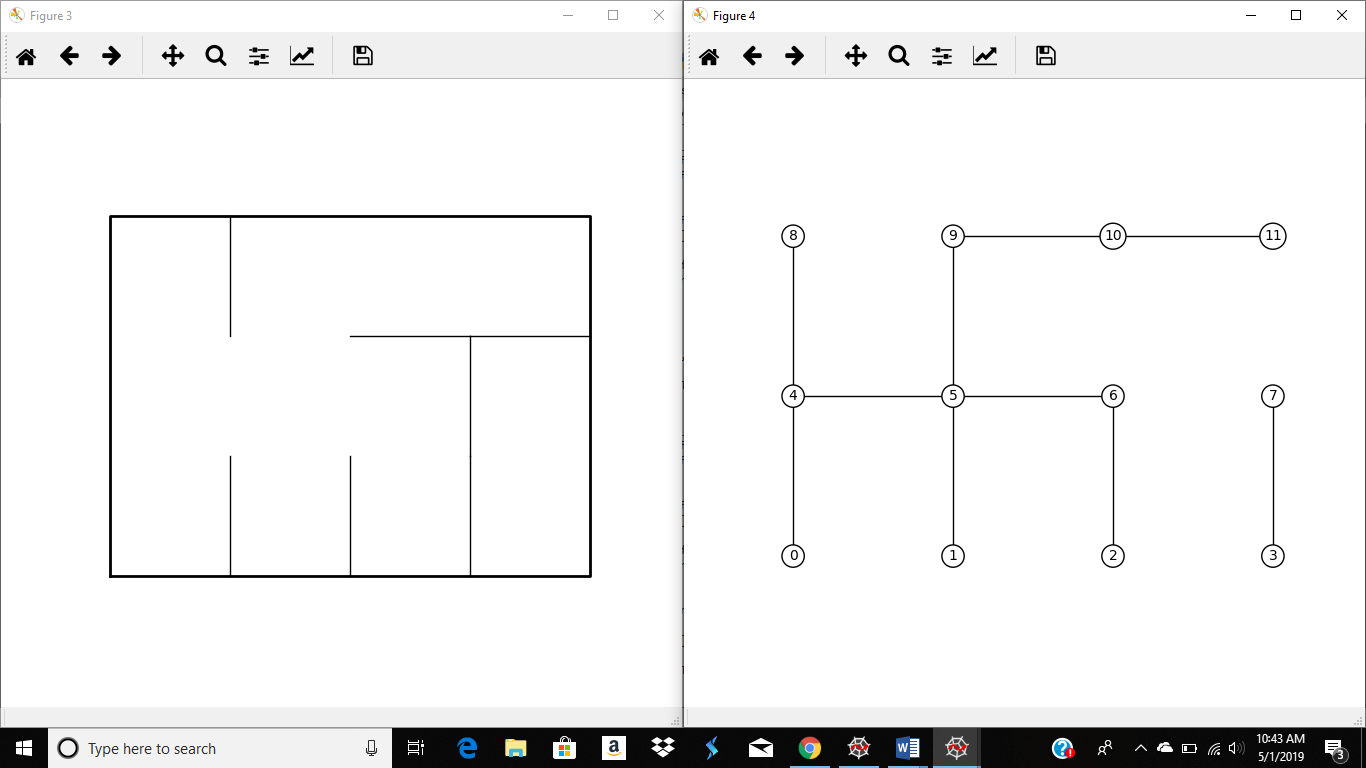
CS 2302 - MW 1:30 Spring 2019

In lab 7 the lab is supposed to use the maze from lab 6 and solve it using breadth first search and depth first search. The lab should also be able to ask the user how many cells to remove and then remove the number of walls from the maze. I was unable to solve the maze using the depth first search and breadth first search, but I was able to draw the path from using numbers and the list of walls removed as well as display the adjacency list. The output for this is below.

When the choice is the less than the number of cells

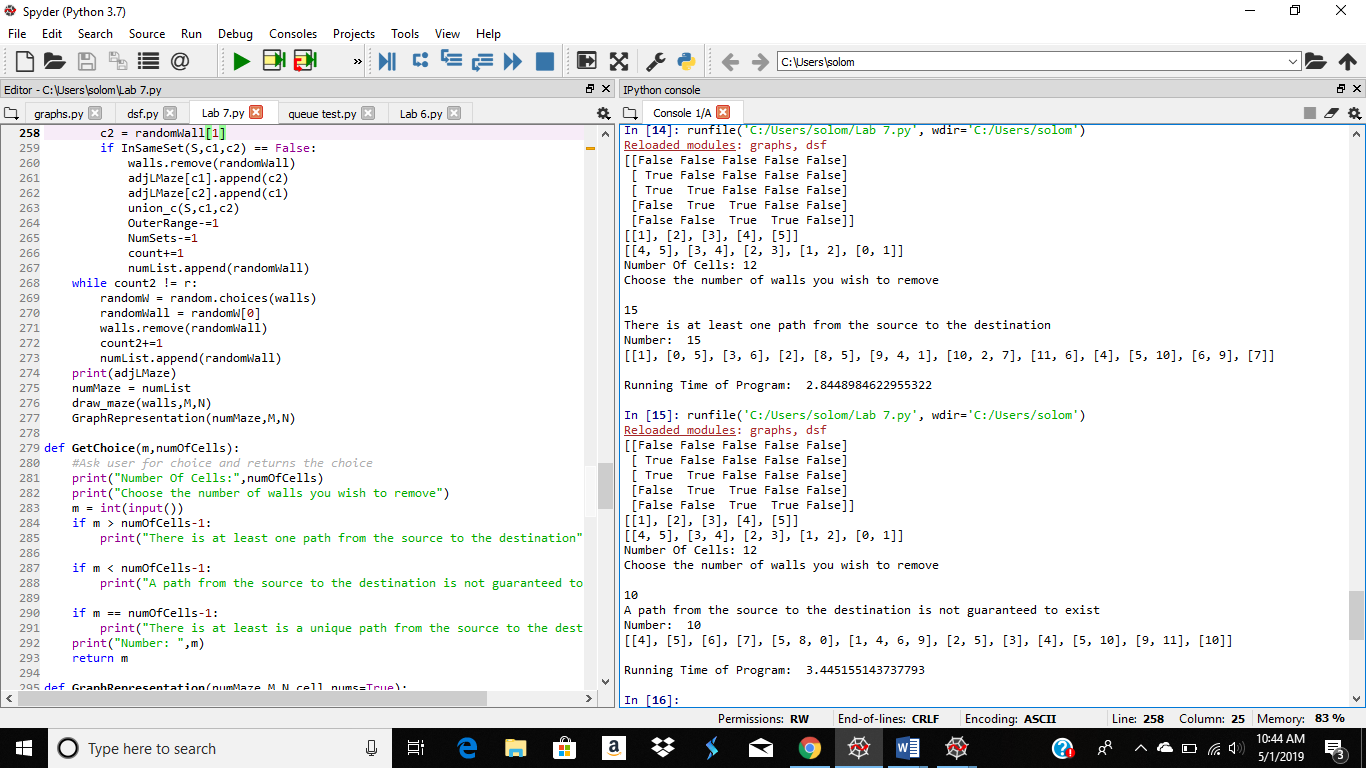
Number of cells = 12 Choice = 10

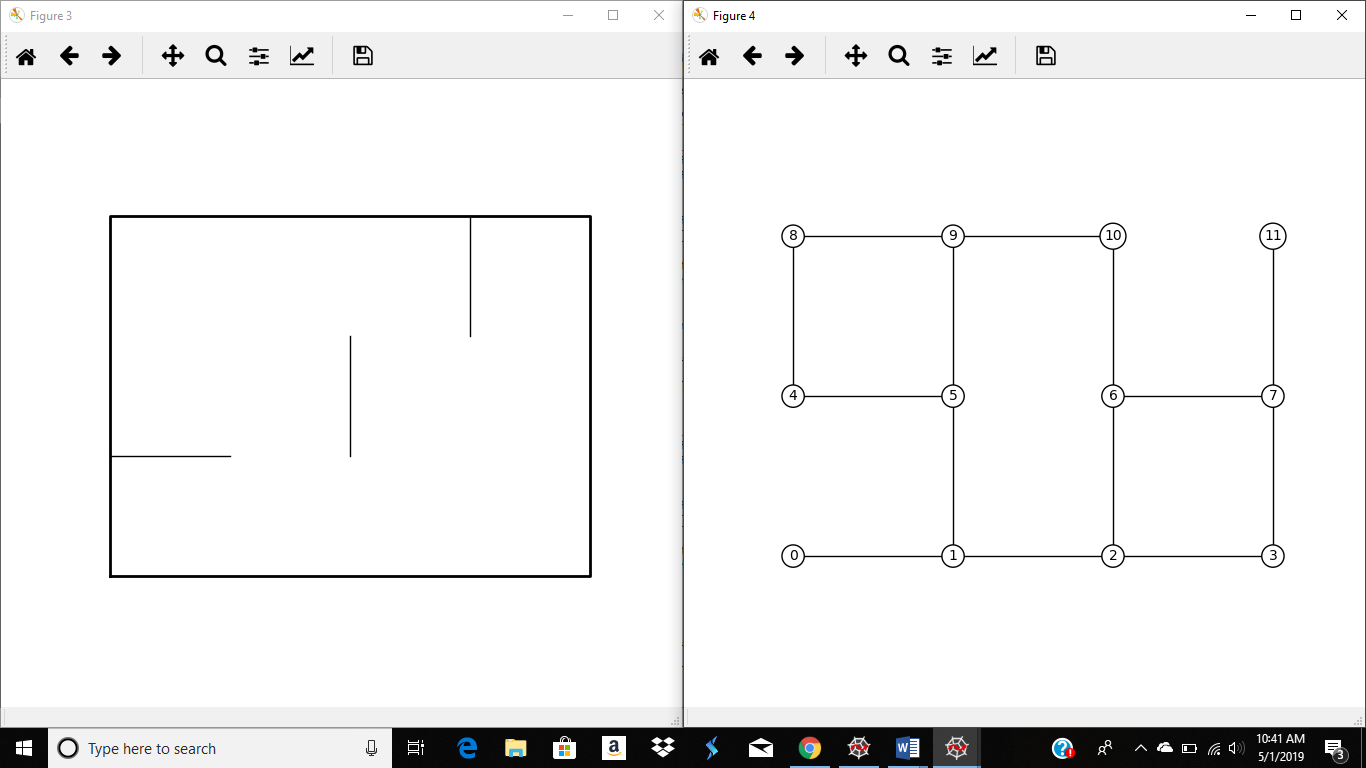




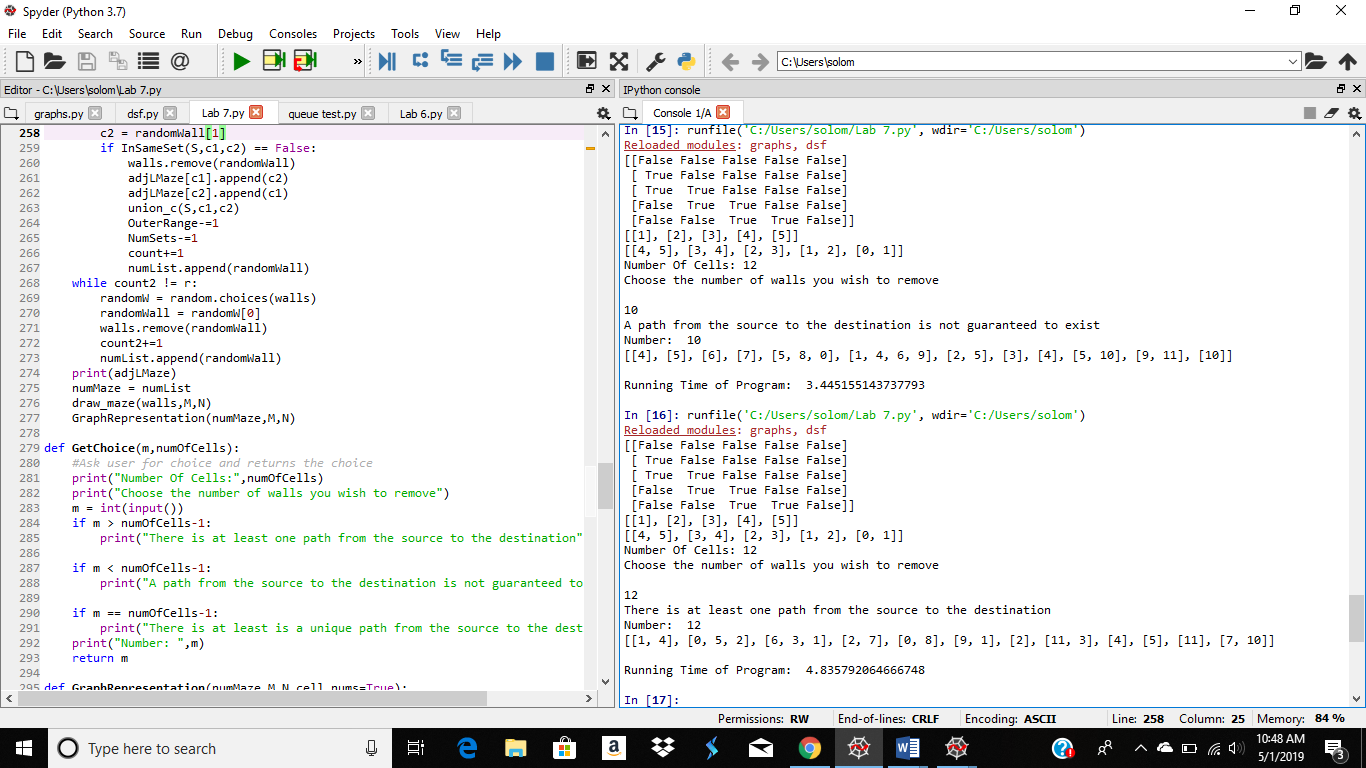
When the choice is more than the number of cells

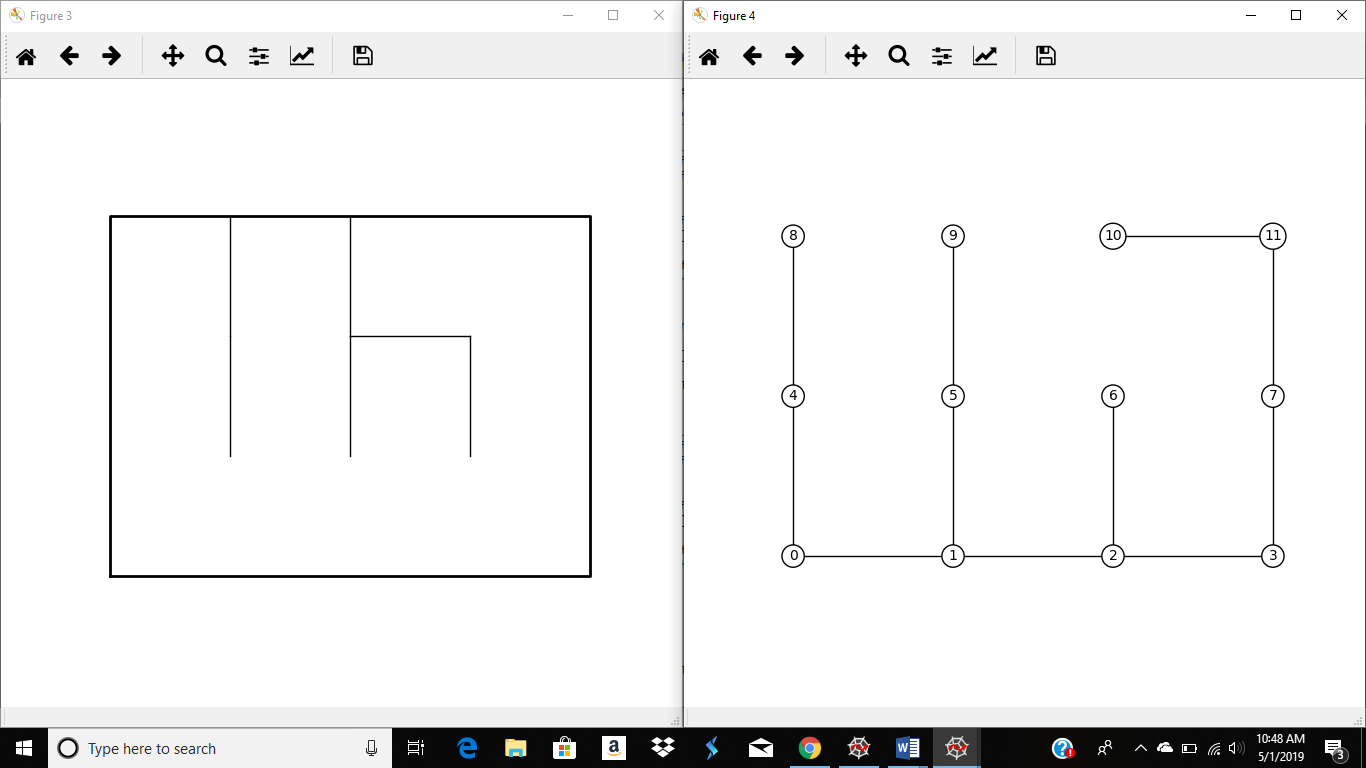
Number of cells = 12. Choice = 15





When the choice is equal to the number of cells

Number of cells = 12. Choice = 12



#Course: CS2302 - Spring 2019

#Author: Solomon Davis

#Lab Number: 7

#Instructor: Olac Fuentes

#Last Modified: May 1, 2019

#Due Date: April 29, 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. Depending on the number chosen by the user the number of

#walls will be removed. The adjacency list, maze with removed walls, and number

#maze will be dsiplayed to the user.

import matplotlib.pyplot as plt

import numpy as np

import random

import time

from scipy import interpolate

#import graphs

import queue

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 draw\_graph\_representation(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')

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)

def RemovingWalls(m,walls,OuterRange,NumSets,numList,numOfCells):

numMaze = wall\_list(M,N)

count = 0

adjLMaze = []

for i in range(numOfCells):

adjLMaze.append([])

while count != m:

randomW = random.choices(walls)

randomWall = randomW[0]

c1 = randomWall[0]

c2 = randomWall[1]

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

walls.remove(randomWall)

adjLMaze[c1].append(c2)

adjLMaze[c2].append(c1)

union\_c(S,c1,c2)

OuterRange-=1

NumSets-=1

count+=1

numList.append(randomWall)

print(adjLMaze)

numMaze = numList

#BreadthFirstSearch(adjLMaze,numMaze,M,N)

draw\_maze(walls,M,N)

GraphRepresentation(numMaze,M,N)

#graphs.draw\_graph(adjLMaze)

def RemovingWalls2(m,walls,OuterRange,NumSets,numList,numOfCells):

count = 0

adjLMaze = []

for i in range(numOfCells):

adjLMaze.append([])

while count != m:

randomW = random.choices(walls)

randomWall = randomW[0]

c1 = randomWall[0]

c2 = randomWall[1]

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

walls.remove(randomWall)

adjLMaze[c1].append(c2)

adjLMaze[c2].append(c1)

union\_c(S,c1,c2)

OuterRange-=1

NumSets-=1

count+=1

numList.append(randomWall)

print(adjLMaze)

numMaze = numList

draw\_maze(walls,M,N)

GraphRepresentation(numMaze,M,N)

def RemovingWalls3(m,walls,OuterRange,NumSets,numList,numOfCells):

count = 0

count2 = 0

adjLMaze = []

for i in range(numOfCells):

adjLMaze.append([])

P = NumSets - 1

r = m - NumSets

while count != P:

randomW = random.choices(walls)

randomWall = randomW[0]

c1 = randomWall[0]

c2 = randomWall[1]

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

walls.remove(randomWall)

adjLMaze[c1].append(c2)

adjLMaze[c2].append(c1)

union\_c(S,c1,c2)

OuterRange-=1

NumSets-=1

count+=1

numList.append(randomWall)

while count2 != r:

randomW = random.choices(walls)

randomWall = randomW[0]

walls.remove(randomWall)

count2+=1

numList.append(randomWall)

print(adjLMaze)

numMaze = numList

draw\_maze(walls,M,N)

GraphRepresentation(numMaze,M,N)

def GetChoice(m,numOfCells):

#Ask user for choice and returns the choice

print("Number Of Cells:",numOfCells)

print("Choose the number of walls you wish to remove")

m = int(input())

if m > numOfCells-1:

print("There is at least one path from the source to the destination")

if m < numOfCells-1:

print("A path from the source to the destination is not guaranteed to exist")

if m == numOfCells-1:

print("There is at least is a unique path from the source to the destination")

print("Number: ",m)

return m

def GraphRepresentation(numMaze,M,N,cell\_nums=True):

fig, ax = plt.subplots()

for w in numMaze:

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

x0 = (w[1]%N)

x1 = x0-1

y0 = (w[1]//N)

y1 = y0

else:#horizontal wall

x0 = (w[0]%N)

x1 = x0

y0 = (w[1]//N)

y1 = y0-1

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

if cell\_nums:

for r in range(M):

for c in range(N):

cell = c + r\*N

ax.text(c,r, str(cell), size=10,

ha="center", va="center",bbox=dict(facecolor='w',boxstyle="circle"))

ax.axis('off')

ax.set\_aspect(1.0)

def BreadthFirstSearch(adjLMaze,numMaze,M,N,cell\_nums=False):

#EL = graphs.adj\_list\_to\_edge\_list(adjLMaze)

#print(EL)

#print(numMaze)

numQueue = queue.Queue()

visited = []

#u = 0

numQueue.put(0)

#u = numMaze[0]

#print(len(numMaze))

#u = 0

#match = 0

count = 0

#cnt = 0

while count != 10:

print("Before",numQueue.queue)

v = numQueue.get()

print("After",numQueue.queue)

#print("Visited",visited,len(visited),v)

for r in range(len(adjLMaze)):

if v == r:

for p in range(len(adjLMaze[v])):

if len(visited) == 0:

visited.append(v)

else:

for f in range(len(visited)):

match = 0

if adjLMaze[v][p] != visited[f]:

match +=0

else:

match +=1

if match !=0:

numQueue.put(adjLMaze[v][p])

visited.append(v)

count+=1

#if v == len(adjLMaze):

#if len(visited) == 0:

# visited.append(v)

#else:

# for f in range(len(visited)):

# match = 0

# if v != visited[f]:

# match += 0

# else:

# match += 1

#print("value of f",f+1)

#cnt+=1

#if match != 0:

# visited.append(v)

#print("cnt",cnt)

#print(numQueue.queue)

#for r in range(len(numMaze)):

#print(r)

#numQueue.put(u)

#u = numMaze[r]

#numQueue.get(numMaze[r])

#for i in visited:

# print('HELLO')

#if u != visited[i]:

# visited.append(numMaze[i])

#def DepthFirstSearch(numMaze,M,N,cell\_nums=False)

#G = graphs.random\_graph

m = ''

numList = []

startimeProgram = time.time()

plt.close("all")

M = 3 # Number of rows

N = 4 # Number of columns

numOfCells = M \* N

m = GetChoice(m,numOfCells)

S = DisjointSetForest(M\*N)

walls = wall\_list(M,N)

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

GraphRepresentation(walls,M,N,cell\_nums=True)

NumSets = NumSets(S)

OuterRange = numOfCells - 1

i = 0

w = 0

if m == numOfCells - 1:

RemovingWalls(m,walls,OuterRange,NumSets,numList,numOfCells)

if m < numOfCells - 1:

RemovingWalls2(m,walls,OuterRange,NumSets,numList,numOfCells)

if m > numOfCells -1:

RemovingWalls3(m,walls,OuterRange,NumSets,numList,numOfCells)

endtimeProgram = time.time()

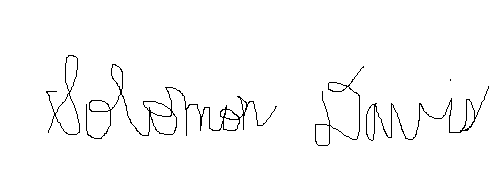
runningtimeProgram = endtimeProgram - startimeProgram

print('')

print('Running Time of 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