Ericka Najera

17 April 2019

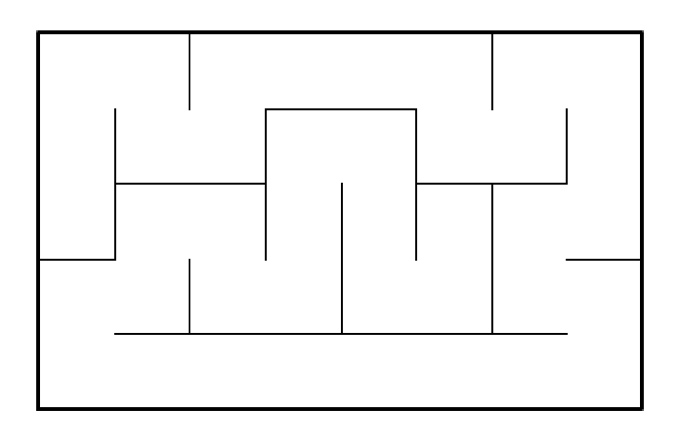
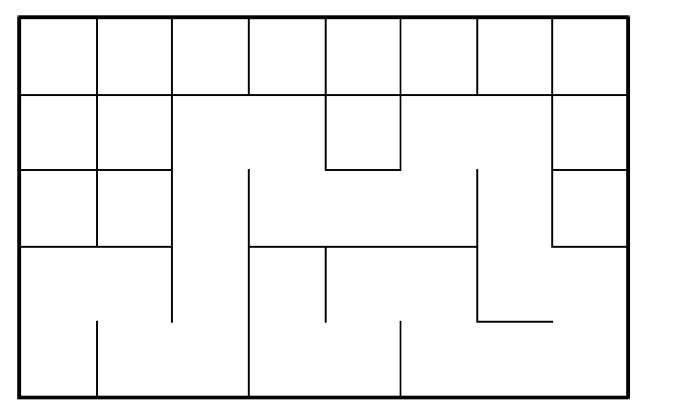
Dr. Fuentes

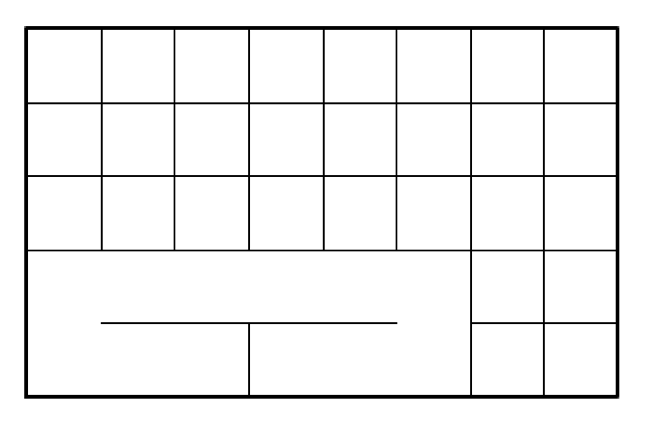
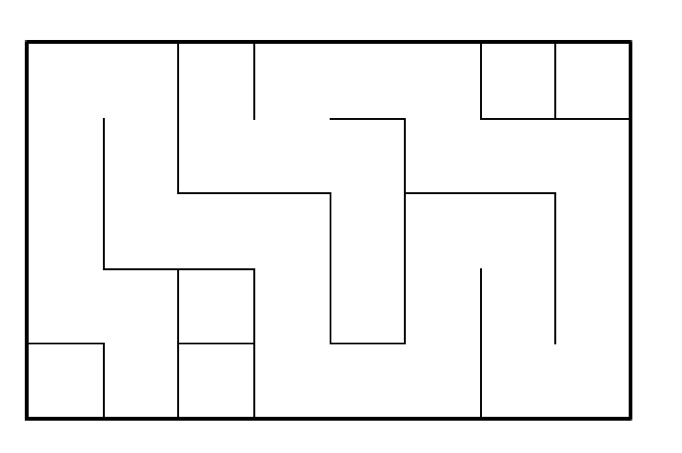
Lab 6

Lab 6 consisted of generating of taking of walls from a random point to create a path in the maze. A maze consists of having paths that lead to nowhere and one math that will reach the end of the maze. The purpose of this lab was to implement then paths of the maze using a disjoint set forest.

At the beginning I thought about joining random walls each time but then I realized that we would have to choose one cell then get the adjacent walls next to and randomly pick one of them. I honestly didn’t know where to begin or how to translate my thoughts into code. At first I would make a function to return the adjacent walls of the given cell. Then I would pick a random cell and call the function to get the adjacent walls. Then I would pick one of those two walls and make a union between them if no one exists. Then the wall I took down would become the new cell and the process would repeat. I would have to include a list to store the cells that were already visited. In order to print the maze accordingly I would convert the disjoint set forest to list and add the walls that were no in the path.

When I did that in my code the path would work but every time I would run my code extra walls would be removed and there would be empty holes in my maze. Then I tried making a np array of the adjacent walls of that index and every time I would make a union I would delete the adjacent walls of both indexes to avoid repetition. Since each cell should have at least one wall every time there was one adjacent wall left I would remove it again to avoid having empty spaces. As for the union by height and union my compression I thought about just repeating the code for both of those but some how when I changed the union functions and find functions accordingly it gave me an error. In my program I couldn’t manage to remove other paths to create a full maze and in every time I run my code it gives better mazes than others.





Running Times:

Union: ﻿479000 nanoseconds

T(n) = 2n = (O)n

Since I could not get the code to work for union by size and by compression I could not get the running times to compare, but I think the union by compression is the faster since path connects to only the root directly.

Appendix:

﻿import matplotlib.pyplot as plt

import numpy as np

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

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

# 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 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

def findAdjacent(walls,n):

#returns a list of the adjacent walls to n

adjacent = []

for i in walls:

if i[0]==n:

adjacent.append(i[1])

if i[1]==n:

adjacent.append(i[0])

return adjacent

def adjacents(walls,maze\_rows,maze\_cols ):

#creates a 2d array with a the adjacent walls of that index list

adj = [ ]

for i in range(maze\_rows\*maze\_cols):

adj.append(findAdjacent(walls,i))

return adj

def findWall(walls,c,r):

#return the index of the wall that unites c and r

for i in range(len(walls)):

if walls[i]==[c,r] or walls[i] == [r,c]:

return i

return None

def path(S):

#searches a disjoint set list and returns the longest path

lengths =0

index = 0

for s in range(len( S)):

if len(S[s])>lengths:

lengths = len(S[s])

index= s

p = []

p.append(S[index][0])

p+=S[index]

return p

def roots(S):

#returns a list of the roots in a disjoint set

root = []

for s in range(len(S)):

if len(S[s])==1:

r = S[s][0]

root.append(r)

return root

def maze(S,walls,adjacents):

#generates a random cell to start off

#collects the adjacent cells and randomly picks a wall

#makes a union with the two cells

#keeps joining the cells until there are no more cells to pick

#returns the disjoint set forest

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

visited =[]

for i in range(len(S)\*2):

if adjacents[d]!=[]:

r = random.randint(0,len(adjacents[d])-1)

if find(S,d)!=find(S,adjacents[d][r]):

union(S,d,adjacents[d][r])

walls.pop(findWall(walls,d,adjacents[d][r]))

visited.append(d)

r = adjacents[d][r]

adj = adjacents[d]

adj2 = adjacents[r]

cell = r

adj.remove(r)

adj2.remove(d)

if len(adj)==1:

other = adj[0]

adj3=adjacents[other]

adj3.remove(d)

adjacents[other]=adj3

adj=[]

adjacents[d]=adj

adjacents[r]=adj2

d = cell

else:

d = visited.pop()

return S

def mazebyHeight(S,walls,adjacents):

#generates a random cell to start off

#collects the adjacent cells and randomly picks a wall

#makes a union with the two cells

#keeps joining the cells until there are no more cells to pick

#returns the disjoint set forest

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

visited =[]

while len(adjacents)>1:

if adjacents[d]!=[]:

r = random.randint(0,len(adjacents[d])-1)

if find(S,d)!=find(S,adjacents[d][r]):

union\_by\_size(S,d,adjacents[d][r])

walls.pop(findWall(walls,d,adjacents[d][r]))

visited.append(d)

r = adjacents[d][r]

adj = adjacents[d]

adj2 = adjacents[r]

cell = r

adj.remove(r)

adj2.remove(d)

if len(adj)==1:

other = adj[0]

adj3=adjacents[other]

adj3.remove(d)

adjacents[other]=adj3

adj=[]

adjacents[d]=adj

adjacents[r]=adj2

d = cell

else:

d = visited.pop()

return S

def mazebyCompression(S,walls,adjacents):

#generates a random cell to start off

#collects the adjacent cells and randomly picks a wall

#makes a union with the two cells

#keeps joining the cells until there are no more cells to pick

#returns the disjoint set forest

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

visited =[]

while len(adjacents)>1:

if adjacents[d]!=[]:

r = random.randint(0,len(adjacents[d])-1)

if find\_c(S,d)!=find\_c(S,adjacents[d][r]):

union\_c(S,d,adjacents[d][r])

walls.pop(findWall(walls,d,adjacents[d][r]))

visited.append(d)

r = adjacents[d][r]

adj = adjacents[d]

adj2 = adjacents[r]

cell = r

adj.remove(r)

adj2.remove(d)

if len(adj)==1:

other = adj[0]

adj3=adjacents[other]

adj3.remove(d)

adjacents[other]=adj3

adj=[]

adjacents[d]=adj

adjacents[r]=adj2

d = cell

else:

d = visited.pop()

return S

plt.close("all")

maze\_rows = 5

maze\_cols = 8

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

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

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

adj = adjacents(walls,maze\_rows,maze\_cols)

S= maze(S,walls,adj)

S= dsfToSetList(S)

pa=path(S)

walls.append(pa)

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

I Ericka Najera certify that this project is entirely my own work. I wrote, debugged, and teste 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

