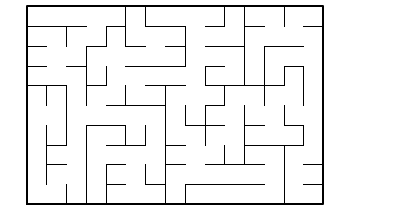
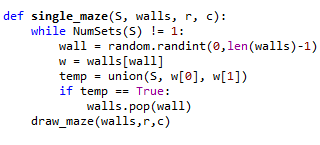
Lab 6

The purpose of this lab is to modify a program to create a maze that is linked to a Disjoint Set Forest. What we need to do is create a maze that consists of only one set, so that starting from 0, you theoretically could reach any other tile within the maze. We also need to do this with using union by size using compression, and from there compare running times of the two mazes.

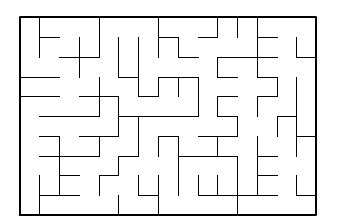
Before editing the code to become what I needed, I had to take some time to understand the process of creating a maze. Basically, given the dimensions of the maze, you create a list that basically consists of cells that have a wall in between them. When you then input that list into the draw\_maze method, which then graphs the block. Removing an item from the list would then cause the corresponding wall not to show when graphing the block.

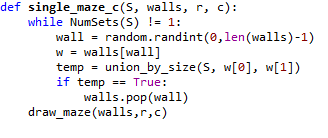
Once I understood that, I got to work on creating the randomized maze. I create a DSF using the size of the maze (Multiply dimensions) and from there created a method receiving the DSF, the block of walls, and the dimensions. Within a while statement that does not end until the number of sets is equal to one, I select a random wall from the list. I then use the two cells specified in that wall in a union for the DSF. When they are part of different sets, they will join together. If they are part of the same set nothing will happen. When two sets are combined though, I then pop the wall that was previously selected. I then have the method draw the maze, because it makes it more convenient when calling on the method. The displayed maze should have access to every tile from 0 (which will not necessarily be the root).





Lastly, when setting up the maze with union by size-compression, I basically used the exact same code, except I call on union\_by\_size instead. For the running times, I placed the time.time() function before and after I created the randomized method. The Running times for the method should vary based on the dimensions of the maze, but in comparison to each other, the two randomly generated mazes will take the same amount of time to create on average.













In conclusion, I’ve learned how to create mazes that I can possibly use in future game development. I would like to experiment and set up more stuff with it, such as how would I be able to edit it so that the path from start to end passes by a certain number of tiles. I also learned how to better work with DSF, as I had the concept down, but not the implementation.

Appendix:

#Patrick Brannan

#Last Edited 4/23/2019

#Purpose of program is to create a random maze of given dimensions

import matplotlib.pyplot as plt

import numpy as np

import random

import time

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):

#print('ROW:', r)

for c in range(maze\_cols):

#print('COLUMN:', c)

cell = c + r\*maze\_cols

#print('CELLS:', cell)

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, S):

# 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 DisjointSetForest(size):

return np.zeros(size,dtype=np.int)-1

def find(S,i):

if S[i]<0:

return i

return find(S,S[i])

def find\_c(S,i):

if S[i]<0:

return i

r = find\_c(S,S[i])

S[i] = r

return r

def union(S,i,j): #joins set if different

ri = find(S,i)

rj = find(S,j)

if ri!=rj:

S[rj] = ri

return True

else:

return False

def union\_by\_size(S,i,j):

ri = find\_c(S,i)

rj = find\_c(S,j)

if ri!=rj:

if S[ri]>S[rj]:

S[rj] += S[ri]

S[ri] = rj

else:

S[ri] += S[rj]

S[rj] = ri

return True

else:

return False

def NumSets(S):

count =0

for i in range(len(S)):

if S[i]<0:

count += 1

return count

#THE TWO METHODS FOR CREATING THE MAZES

def single\_maze(S, walls, r, c):

while NumSets(S) != 1:

wall = random.randint(0,len(walls)-1) #holds the wall value for later

w = walls[wall]

temp = union(S, w[0], w[1])

if temp == True:

walls.pop(wall)

draw\_maze(walls,r,c)

#Exact same as previous method except it uses Union by size

def single\_maze\_c(S, walls, r, c):

while NumSets(S) != 1:

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

w = walls[wall]

temp = union\_by\_size(S, w[0], w[1])

if temp == True:

walls.pop(wall)

draw\_maze(walls,r,c)

plt.close("all")

rows = 10

col = 15

print('CREATING MAZES WITH DIMENSIONS', rows, 'X', col)

#Important to Note that I edited the union methods to return T/F

#Start of randomized maze

M = DisjointSetForest(rows\*col)

walls = wall\_list(rows,col,M)

draw\_maze(walls,rows,col,cell\_nums=True) #Prints the base block for maze

start = time.time()

single\_maze(M, walls, rows, col)

end = time.time()

run\_time\_1 = end - start

#Start on maze with size/compression - differentiates it from first maze

M\_c = DisjointSetForest(rows\*col)

walls\_c = wall\_list(rows,col,M\_c)

start = time.time()

single\_maze\_c(M\_c, walls\_c, rows, col)

end = time.time()

run\_time\_2 = end - start

print('Running time for creating a random maze:', run\_time\_1)

print('Running time for maze with union by size and compression:', run\_time\_2)

***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 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.***

***-Patrick Brannan***