Lab 6 is all about building a maze using disjoint set forest. In computer science disjoint set forest is a data structure that tracks a set of elements partitioned into a disjoint subsets. It provides a near- constant time operation to add, merge, and to determine if sets are in the same set. Disjoint set forest consists of two basic methods find and union. The method find is used to see if to elements are in the same subset. The method union is used to combine to elements that are not in the same subset. For this lab I will be using union, union compression and union by size compression to create a maze that is a collection of cells separated by walls in such a way that there is exactly one simple path meaning no cell is visited more than once.

The way I create my maze is by using a while loop. The while loop will keep going as long as the number of sets in the disjoint forest set is not 1. Inside the loop I created a variable called choice that stores a random set from the disjoint set forest using the imported random.choice method. After that I created a variable named index to store the index of the wall in the disjoint set forest by using .index method. After I use an If statement to see if the wall is part of the single subset that we want. If the wall is part of it we do nothing and go to the next wall. If the wall is not part of it we pop the wall and connect the walls by using union, union compression, and union by size compression. After that we can subtract a set from the number of subsets left. This will go on till the number of sets are just one. We can then print the newly created maze using the draw\_maze method provide to us by our teacher.

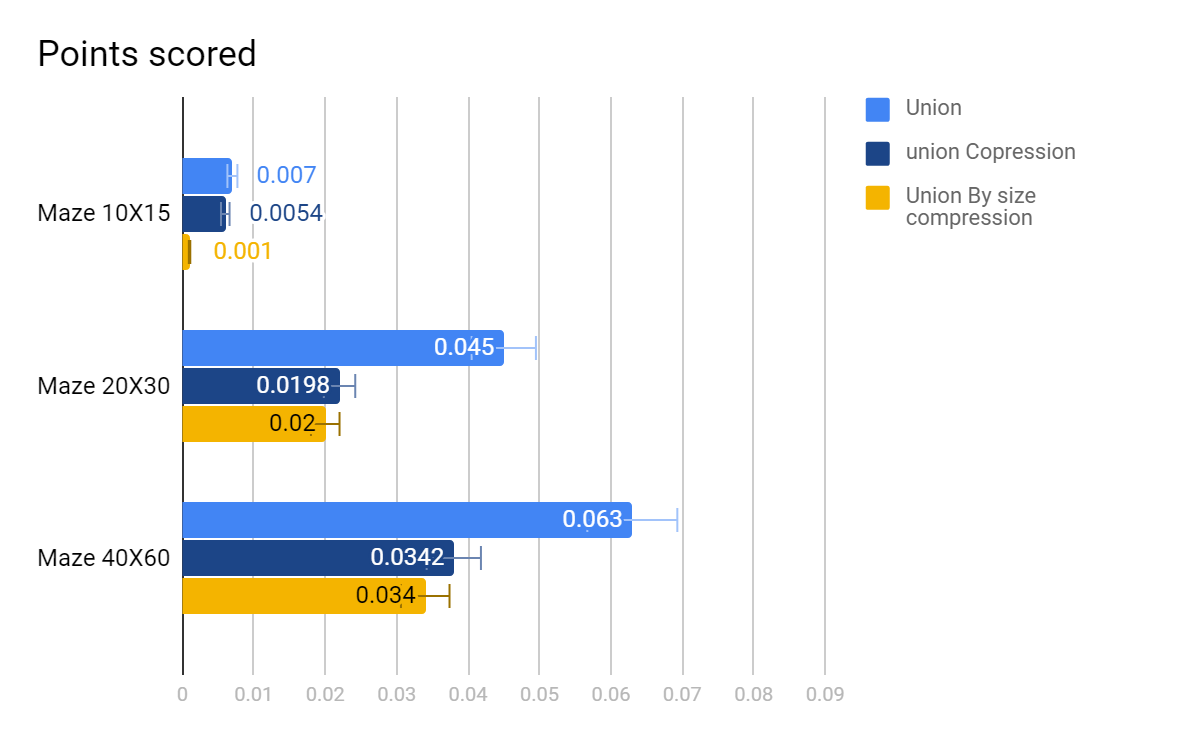
For the first maze I used the simple method union. This method uses a list of individual subsets and joins to subsets if they are different. In other words if element A and element B are not in the same subset we add element B to elements A subset. The way the pointers work is that each element points to its parent that will lead us to the root element meaning the start of the subset. So if we have three elements that are connected A, B, and C and A is our root then C would point to B and B would point to A while A does not point anywhere else as it is our root.

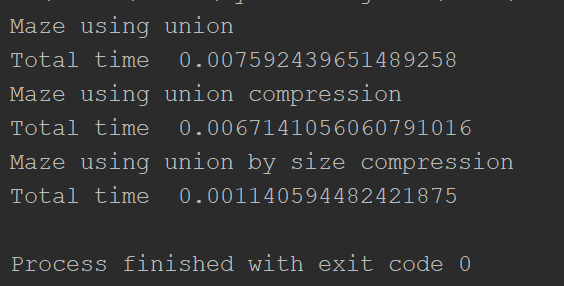
For the second maze I used union compression. Union compression is similar to normal union but instead of pointing to the parent it will always keep each element pointing to the root. For example if we go back to our example of having three elements A,B, and C instead of having C point to B and B point to A, C and A will both point to the root A. This way instead of having to follow the path to the root we will get to the root right away.

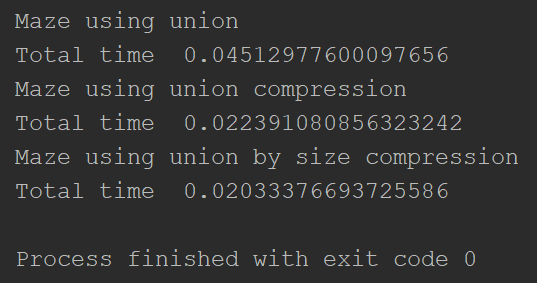
Finally for the last maze I used union by size compression. Just like before this union works the same like all the other the only difference is that instead of just adding the element to the set first set it will instead join the two sets depending on size of the sets. In other words if we have a set that's bigger that the other set the smaller sets root will point to the bigger sets root. For example if we have a set A and B where B points to A our root and set C where C is our root and use union by size compression the C will point to the bigger sets root. In this case it would be A.

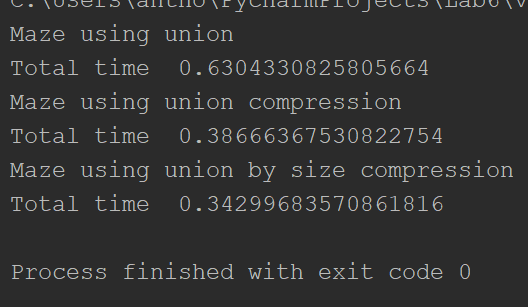
For the lab I also had to time the creation of each maze from start to ending. The results ended with the bigger the number of cells union by size compression would be faster, followed by union compression, and lastly union. I came up with this results by running my experiment with 3 different sizes of mazes. The first size was a maze that was 10X15 the results ended up been .007 sec for union, .006 sec for union compression, and .001 sec for union by size compression. The second size was a maze that was 20X30 the results ended up been .045 sec for union, .022 sec for union compression, and .020 sec for union by size compression. The last size was a maze that was 40X60 the results ended up been .063 sec for union, .038 sec for union compression, and .034 sec for union by size compression.

In conclusion the fasted way to build a maze using disjoint set forest would be to do union by size compression. Not only would it be better for a maze but it is the best way to use in case of dealing with greater size of sets.









# Lab 6

# Programmed by Anthon Herrera

# Last modified April, 14, 2019

import matplotlib.pyplot as plt

import numpy as np

from scipy import interpolate

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\_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,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 NumberOfSets(S):# will count the number of -1 that show diffrent sets

count = 0

for x in S:

if x == -1:

count += 1

return count

plt.close("all")

maze\_rows = 40

maze\_cols = 60

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

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

disjoint\_set\_forest = DisjointSetForest(maze\_rows\*maze\_cols)

number\_of\_set = NumberOfSets(disjoint\_set\_forest)

print('Maze using union')

timer\_0 = time.time()

while number\_of\_set > 1: # will create the maze using stander union of sets

choice = random.choice(walls) # selects random wall from list

index = walls.index(choice) # return index of wall

if find(disjoint\_set\_forest,choice[0]) != find(disjoint\_set\_forest,choice[1]):

walls.pop(index) # deletes the wall selected

union(disjoint\_set\_forest,choice[0],choice[1]) # add the wall to a set using union

number\_of\_set -= 1 # decreases the number of sets by one

timer\_1 = time.time()

print('Total time ', timer\_1-timer\_0)

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

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

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

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

disjoint\_set\_forest = DisjointSetForest(maze\_rows\*maze\_cols)

number\_of\_set = NumberOfSets(disjoint\_set\_forest)

print('Maze using union compression ')

timer\_0 = time.time()

while number\_of\_set > 1: #Will create the maze using union with compression

choice = random.choice(walls) # selects random wall from list

index = walls.index(choice) # return index of wall

if find(disjoint\_set\_forest,choice[0]) != find(disjoint\_set\_forest,choice[1]):

walls.pop(index) # deletes the wall selected

union\_c(disjoint\_set\_forest,choice[0],choice[1]) # add the wall to a set using union compression

number\_of\_set -= 1 # decreases the number of sets by one

timer\_1 = time.time()

print('Total time ', timer\_1-timer\_0)

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

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

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

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

disjoint\_set\_forest = DisjointSetForest(maze\_rows\*maze\_cols)

number\_of\_set = NumberOfSets(disjoint\_set\_forest)

print('Maze using union by size compression')

timer\_0 = time.time()

while number\_of\_set > 1: #Will create the maze using union by size with compression

choice = random.choice(walls) # selects random wall from list

index = walls.index(choice) # return index of wall

if find(disjoint\_set\_forest,choice[0]) != find(disjoint\_set\_forest,choice[1]):

walls.pop(index) # deletes the wall selected

union\_by\_size(disjoint\_set\_forest,choice[0],choice[1]) # add the wall to a set using union by size compression

number\_of\_set -= 1 # decreases the number of sets by one

timer\_1 = time.time()

print('Total time ', timer\_1-timer\_0)

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

#plt.show()

Academic dishonesty includes but is not limited to cheating, plagiarism and collusion. Cheating may involve copying from or providing information to another student, possessing unauthorized materials during a test, or falsifying data (for example program outputs) in laboratory reports. Plagiarism occurs when someone represents the work or ideas of another person as his/her own. Collusion involves collaborating with another person to commit an academically dishonest act. Professors are required to - and will - report academic dishonesty and any other violation of the Standards of Conduct to the Dean of Students.

* Anthony Herrera