INFO 6205 Program Structures & Algorithms

Assignment - 3

Task:

Step 1: (a) Implement height-weighted Quick Union with Path Compression. For this, you will flesh out the class UF_HWQUPC. All you have to do is to fill in the sections marked with // TO BE IMPLEMENTED ... // ...END IMPLEMENTATION.

(b) Check that the unit tests for this class all work. You must show "green" test results in your submission (screenshot is OK).

Step2: Using your implementation of UF_HWQUPC, develop a UF ("union-find") client that takes an integer value n from the command line to determine the number of "sites." Then generates random pairs of integers between 0 and n-1, calling connected() to determine if they are connected and union() if not. Loop until all sites are connected then print the number of connections generated. Package your program as a static method count() that takes n as the argument and returns the number of connections; and a main() that takes n from the command line, calls count() and prints the returned value. If you prefer, you can create a main program that doesn't require any input and runs the experiment for a fixed set of n values. Show evidence of your run(s).

Step3:Determine the relationship between the number of objects (*n*) and the number of pairs (*m*) generated to accomplish this (i.e. to reduce the number of components from *n* to 1). Justify your conclusion.

Output:

After implementing the mergeComponents() find(), and doPathCompression() methods in UF_HWQUPC.java, I ran the UF_HWQUPC test and passed them successfully.(Screenshot Attached)

Then created main method and count () function in UF_HWQUPC class which takes in an integer as an argument and generates random pairs between 0 and n-1. Checked if they are connected or not and its unions them.

It continues to do it till the number of components go from n to 1. It returns the generated pairs required to bring the components from n to 1. We call the number of generated pairs for a given n as say ‘m’. Since the value of m changed each time for a given n, it is averaged (avg) as the m values over 100 runs. I started from n = 100 and till n = 4850 (total of 20 values of n).

The output values are given below:

For n= 100 number of generated pairs = 263 for 100 runs
Coefficient for n=100 and m = 263 is= 0.5710972437027761

For n= 350 number of generated pairs = 1108 for 100 runs
Coefficient for n=350 and m = 1108 is= 0.5404148873380976

For n= 600 number of generated pairs = 2121 for 100 runs
Coefficient for n=600 and m = 2121 is= 0.5526088593326193

For n= 850 number of generated pairs = 3153 for 100 runs
Coefficient for n=850 and m = 3153 is= 0.5499305839727094

For n= 1100 number of generated pairs = 4087 for 100 runs
Coefficient for n=1100 and m = 4087 is= 0.5305468822646682

For n= 1350 number of generated pairs = 5359 for 100 runs
Coefficient for n=1350 and m = 5359 is= 0.550736238000797

For n= 1600 number of generated pairs = 6545 for 100 runs
Coefficient for n=1600 and m = 6545 is= 0.5544536018164035

For n= 1850 number of generated pairs = 7390 for 100 runs
Coefficient for n=1850 and m = 7390 is= 0.530988431000219

For n= 2100 number of generated pairs = 8479 for 100 runs
Coefficient for n=2100 and m = 8479 is= 0.5278145470974566

For n= 2350 number of generated pairs = 9761 for 100 runs
Coefficient for n=2350 and m = 9761 is= 0.5351102457677069

For n= 2600 number of generated pairs = 10927 for 100 runs
Coefficient for n=2600 and m = 10927 is= 0.5344715441051838

For n= 2850 number of generated pairs = 12141 for 100 runs
Coefficient for n=2850 and m = 12141 is= 0.5355072565843942

For n= 3100 number of generated pairs = 13485 for 100 runs
Coefficient for n=3100 and m = 13485 is= 0.541101484734575

For n= 3350 number of generated pairs = 14761 for 100 runs
Coefficient for n=3350 and m = 14761 is= 0.5428635005078937

For n= 3600 number of generated pairs = 15816 for 100 runs
Coefficient for n=3600 and m = 15816 is= 0.5365124095648859

For n= 3850 number of generated pairs = 17229 for 100 runs
Coefficient for n=3850 and m = 17229 is= 0.5420491685942553

For n= 4100 number of generated pairs = 18085 for 100 runs
Coefficient for n=4100 and m = 18085 is= 0.5302454957452811

For n= 4350 number of generated pairs = 19569 for 100 runs
Coefficient for n=4350 and m = 19569 is= 0.5369608108526568

For n= 4600 number of generated pairs = 20967 for 100 runs
Coefficient for n=4600 and m = 20967 is= 0.5404488153056506

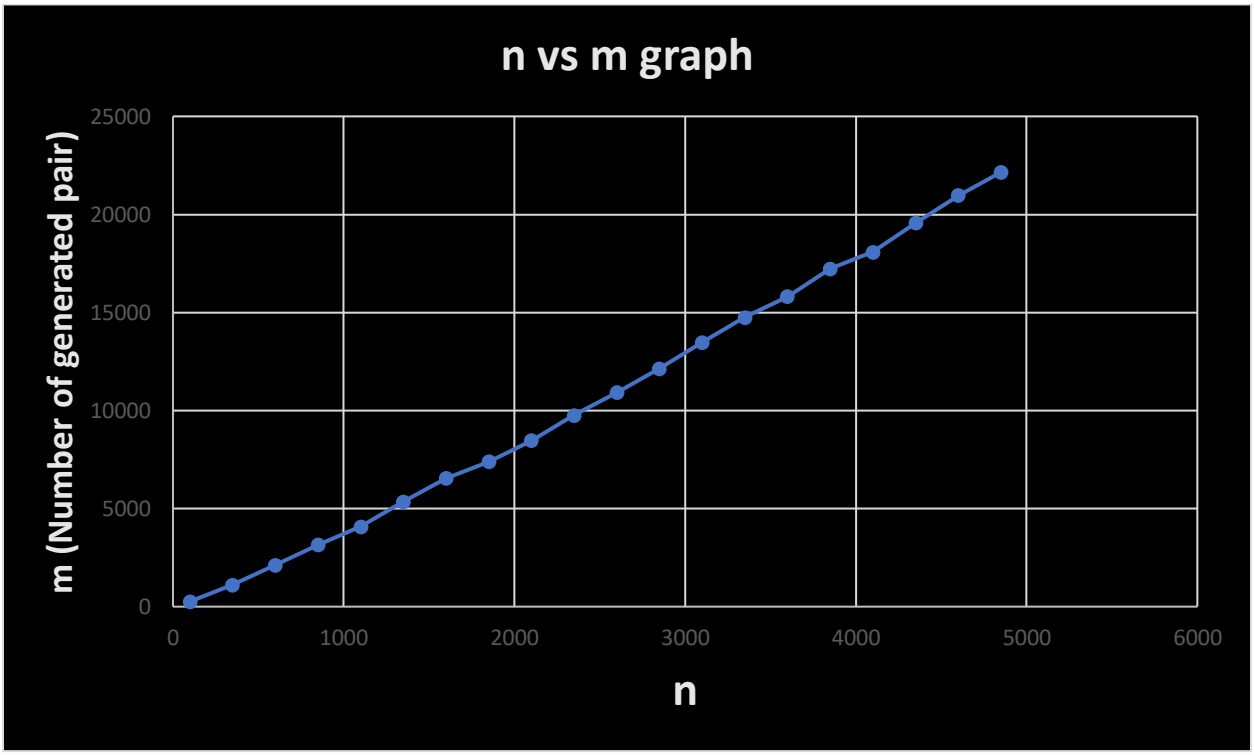
For n= 4850 number of generated pairs = 22153 for 100 runs
Coefficient for n=4850 and m = 22153 is= 0.538208087425335

*Average value of the coefficient (m/n*log(n)) is=0.5411035046856781*

Table Values of these are as follows:

n	m
100	263
350	1108
600	2121
850	3153
1100	4087
1350	5359
1600	6545
1850	7390
2100	8479
2350	9761
2600	10927
2850	12141
3100	13485
3350	14761
3600	15816
3850	17229
4100	18085
4350	19569
4600	20967
4850	22153

Graph is represented as below :



Relationship Conclusion

Observing the n vs m graph and the values of them in the table, we can see that there is almost a linear relationship between the number of objects and the number of generated pairs. However, when dividing m and n gave results that kept growing as n increased as show below for few sample tests values:

n	m	m/n
100	263	2.63
350	1108	3.165
600	2121	3.53
850	3153	3.709
1100	4087	3.715

In height weighted quick union, the height of a node with a tree having total of 2k nodes is at most k or the height of a tree having total of n nodes is at most log (n).
As we keep track of the height of each tree and connect the shorter root point to taller one rather than doing it arbitrarily, it is guaranteed a logarithmic performance. And If we assume that we perform n unions and maximum height of a node is log(n), we can conclude that the number of operations required to perform n unions would be at most n*log(n). With this conclusion, we can relate that there must be a n*log(n) factor in the relationship between m and n.
From the results we observe that there is a relation between n and m where the coefficient is given by m/n*(log(n)). Averaging these for 20 values of n, I found the value of m/n*(log(n)) to be roughly 0.54.

ie. => **m=k * n * log(n) where k=0.54 approx.**

Screenshot of tests passed