Pallak Singh (NUID: 001063095) - Section 02

INFO 6205
Program Structures & Algorithms
Fall 2020
Assignment No 3

- **Task**

  We were given the following tasks:
  1. Complete the implementation of class UF_HWQUPC.java, i.e implement the find(), mergeComponents() and doPathCompression() method that perform find and union for height weighted quick union with path compression.
  2. Develop a client that takes an integer n, generates random pairs from 0 to n-1 and returns the connections that are generated.
  3. Derive the relationship between the generated pairs (m) and the number of objects (n).

  Please find the code for the client in UFClient.java present in the /union_find directory.

- **Output**

  After implementing the mergeComponents() and find() methods in UF_HWQUPC.java, I created a class called UFClient.java. In UFClient.java, there is a method count() which takes in an integer as an argument and generates random pairs between 0 and n-1. With these pairs, it checks whether they are connected, if not, it unions them. It does this 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 'm'. Since the value of m changed each time for a given n, I averaged the m values over 100 runs. I started from n = 100 and doubling it till n = 819,200 (total of 14 values of n). The console output is given below:

Console Output (for n = 100 to n = 819,200)

```
For n: 100 the number of generated pairs are: 195 for 200 runs
For n: 200 the number of generated pairs are: 450 for 200 runs
For n: 400 the number of generated pairs are: 1017 for 200 runs
For n: 800 the number of generated pairs are: 2279 for 200 runs
For n: 1600 the number of generated pairs are: 5020 for 200 runs
For n: 3200 the number of generated pairs are: 10754 for 200 runs
For n: 6400 the number of generated pairs are: 23897 for 200 runs
For n: 12800 the number of generated pairs are: 51440 for 200 runs
For n: 25600 the number of generated pairs are: 110856 for 200 runs
For n: 51200 the number of generated pairs are: 239598 for 200 runs
For n: 102400 the number of generated pairs are: 508426 for 200 runs
For n: 204800 the number of generated pairs are: 1073363 for 200 runs
For n: 409600 the number of generated pairs are: 2296621 for 200 runs
For n: 819200 the number of generated pairs are: 4840474 for 200 runs
```

Once I had the n and the generated pairs (m), I plotted a n vs m graph which is shown below:

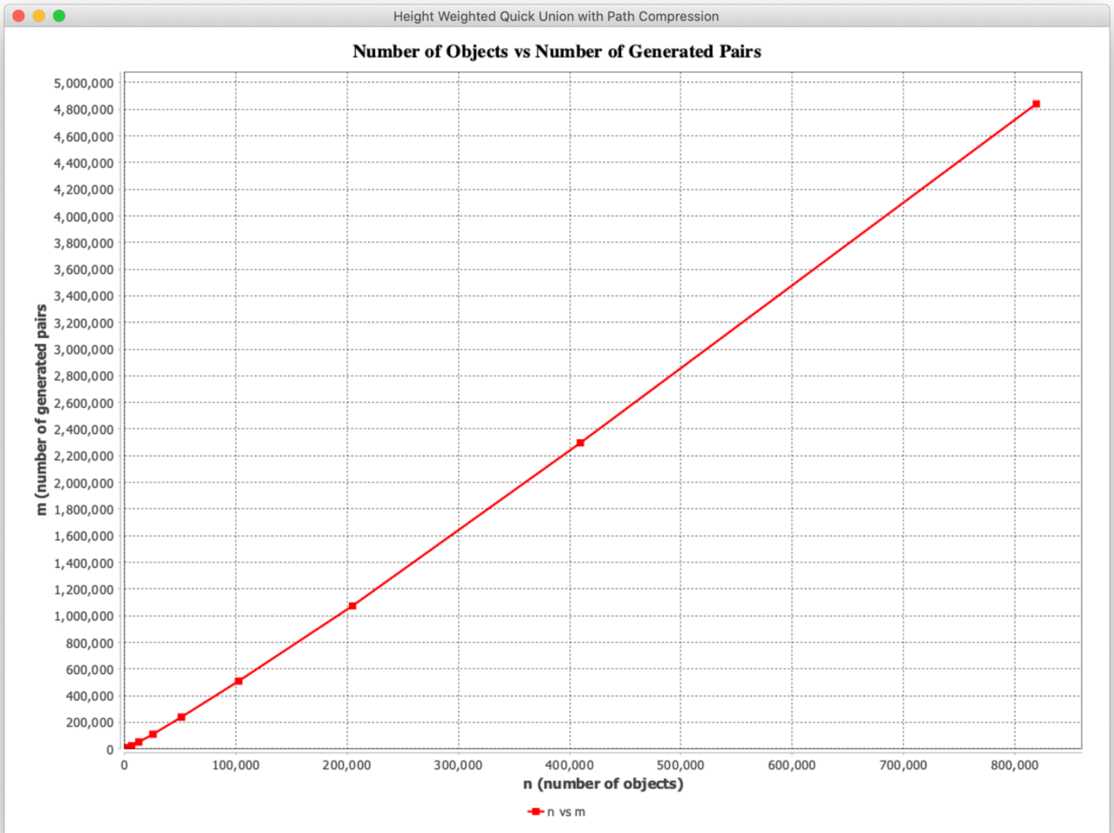

Fig 2.1: Plotting the Number of Objects (n) vs Number of Generated Pairs (m) for the range n = 100 to n = 819200 doubling n at each run

While giving us a rough idea of the nature of the relationship, I could see that there were lot of points being concentrated towards the origin due to the range of n being too large, so I ran the client again for n = 1000 and incremented it by 500 till n = 39,500 (total of 80 values of n)

Console Output (for n = 1000 to n = 39,500)

```
For n: 100 the number of generated pairs are: 261 for 200 runs
For n: 600 the number of generated pairs are: 2146 for 200 runs
For n: 1100 the number of generated pairs are: 4155 for 200 runs
For n: 1600 the number of generated pairs are: 6407 for 200 runs
For n: 2100 the number of generated pairs are: 8657 for 200 runs
For n: 2600 the number of generated pairs are: 10767 for 200 runs
For n: 3100 the number of generated pairs are: 13324 for 200 runs
For n: 3600 the number of generated pairs are: 16247 for 200 runs
For n: 4100 the number of generated pairs are: 18487 for 200 runs
For n: 4600 the number of generated pairs are: 20934 for 200 runs
For n: 5100 the number of generated pairs are: 23074 for 200 runs
For n: 5600 the number of generated pairs are: 25916 for 200 runs
For n: 6100 the number of generated pairs are: 28054 for 200 runs
For n: 6600 the number of generated pairs are: 30730 for 200 runs
For n: 7100 the number of generated pairs are: 33559 for 200 runs
For n: 7600 the number of generated pairs are: 36036 for 200 runs
```

```
For n: 8100 the number of generated pairs are: 38897 for 200 runs
For n: 8600 the number of generated pairs are: 41807 for 200 runs
For n: 9100 the number of generated pairs are: 44236 for 200 runs
For n: 9600 the number of generated pairs are: 46869 for 200 runs
For n: 10100 the number of generated pairs are: 49804 for 200 runs
For n: 10600 the number of generated pairs are: 52619 for 200 runs
…
…
…
For n: 34600 the number of generated pairs are: 189887 for 200 runs
For n: 35100 the number of generated pairs are: 190309 for 200 runs
For n: 35600 the number of generated pairs are: 195850 for 200 runs
For n: 36100 the number of generated pairs are: 197675 for 200 runs
For n: 36600 the number of generated pairs are: 199216 for 200 runs
For n: 37100 the number of generated pairs are: 206449 for 200 runs
For n: 37600 the number of generated pairs are: 208790 for 200 runs
For n: 38100 the number of generated pairs are: 214446 for 200 runs
For n: 38600 the number of generated pairs are: 217597 for 200 runs
For n: 39100 the number of generated pairs are: 216087 for 200 runs
For n: 39600 the number of generated pairs are: 218136 for 200 runs
```

Once I had the n and the generated pairs (m) within a smaller range of n, I once again plotted a n vs m graph which is shown below:

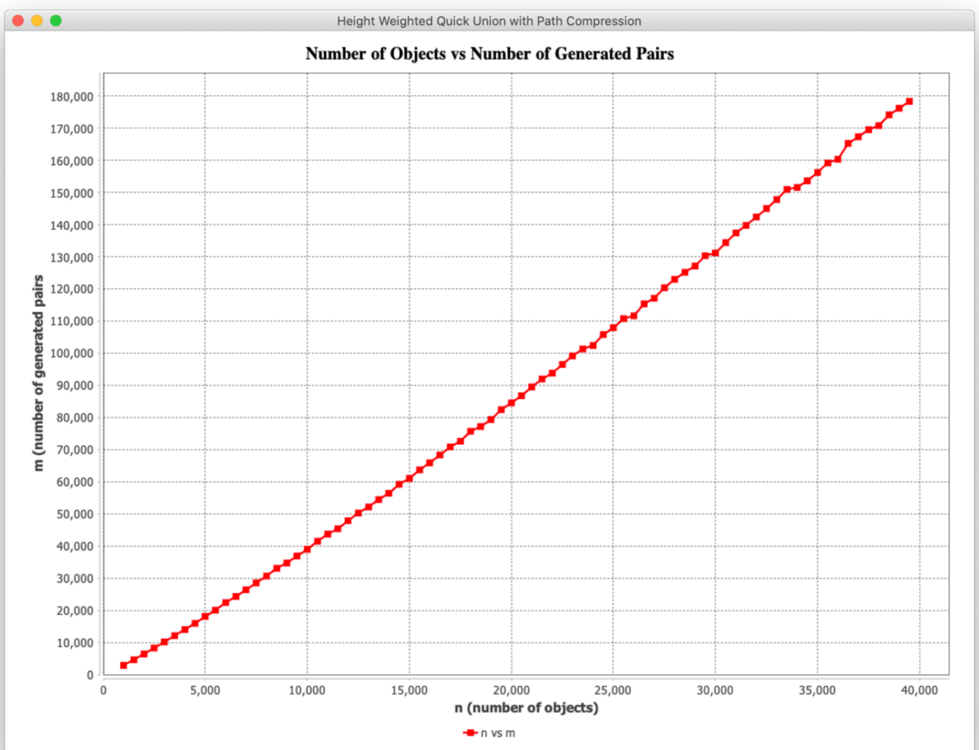

Fig 2.2: Plotting the Number of Objects (n) vs Number of Generated Pairs (m) for the range n = 100 to n = 39600 with increments of 500

- **Relationship Conclusion**

Observing the n vs m graph, we can see that there is almost a linear relationship between the number of objects and the number of generated pairs. However, dividing m over n gave results that kept growing as n increased. I have recorded the $\frac{m}{n}$ results for range of n = 100 to n = 4100 in the table below:

n	m	$\frac{m}{n}$
100	263	2.630
600	2155	3.591
1100	4206	3.823
1600	6544	4.09
2100	8813	4.196
2600	11002	4.231
3100	13200	4.258
3600	15690	4.358
4100	18485	4.508

Table 3.1: Results recording $\frac{m}{n}$ for range n = 100 to n = 4100 with increments of 500

These results weren't that surprising considering the logarithmic nature of height weighted quick union. In height (or even weight) weighted quick union, the height of a node with a tree having total of 2^k nodes is at most k or in other words, the height of a tree having total of n nodes is at most $\log(n)$. Since in this mechanism we keep track of the height of each tree and connect the smaller tree to the larger rather than doing it arbitrarily, we are guaranteed a logarithmic performance. 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, I hypothesized that there must be a n*log(n) factor in the relationship between m and n. To check this, I ran my client again and I found $\frac{m}{n*\log(n)}$ and recorded the results for range of n = 100 to n = 3200 below:

n	m	$\frac{m}{n*\log(n)}$
100	263	0.571
600	2155	0.561
1100	4206	0.545
1600	6544	0.554
2100	8813	0.548
2600	11002	0.531
3100	13200	0.529
3600	15690	0.532
4100	18485	0.541

Table 3.2: Results recording $\frac{m}{n*\log(n)}$ for range n = 100 to n = 4100 with increments of 500

From the results above, we see that there is indeed a relation between n and m where the coefficient is given by $\frac{m}{n*\log(n)}$.
Averaging these for 80 values of n, I found the value of $\frac{m}{n*\log(n)}$ to be roughly 0.53.

Console Output:
```
For n: 100 the number of generated pairs are: 258 for 200 runs
Coefficient for n: 100 and m = 258 is: 0.5602398816551948

For n: 600 the number of generated pairs are: 2102 for 200 runs
Coefficient for n: 600 and m = 2102 is: 0.5476585678063016

For n: 1100 the number of generated pairs are: 4217 for 200 runs
Coefficient for n: 1100 and m = 4217 is: 0.5474226088842931

For n: 1600 the number of generated pairs are: 6377 for 200 runs
Coefficient for n: 1600 and m = 6377 is: 0.540221637705608

For n: 2100 the number of generated pairs are: 8505 for 200 runs
Coefficient for n: 2100 and m = 8505 is: 0.5294330372760784

For n: 2600 the number of generated pairs are: 11128 for 200 runs
Coefficient for n: 2600 and m = 11128 is: 0.5443030422625136

For n: 3100 the number of generated pairs are: 13169 for 200 runs
Coefficient for n: 3100 and m = 13169 is: 0.5284216130863639
…
…
…
For n: 38100 the number of generated pairs are: 215159 for 200 runs
Coefficient for n: 38100 and m = 215159 is: 0.5353843519408286

For n: 38600 the number of generated pairs are: 217179 for 200 runs
Coefficient for n: 38600 and m = 217179 is: 0.5327521009892199

For n: 39100 the number of generated pairs are: 220016 for 200 runs
Coefficient for n: 39100 and m = 220016 is: 0.5321612191683455

For n: 39600 the number of generated pairs are: 221864 for 200 runs
Coefficient for n: 39600 and m = 221864 is: 0.5292194396604014

Average value of the coefficient (m/n*log(n)) is: 0.5308734272744104
```

From these results we can conclude that:

$$m \approx k * n * \log(n)$$

Where $k \approx 0.53$

- **Relationship Evidence**

  In order to prove the relationship

$$m \approx 0.53 * n * \log(n)$$

I plotted the graph for n vs m and n vs n*log(n)*0.53. The graph is provided below:

Pallak Singh (NUID: 001063095) - Section 02

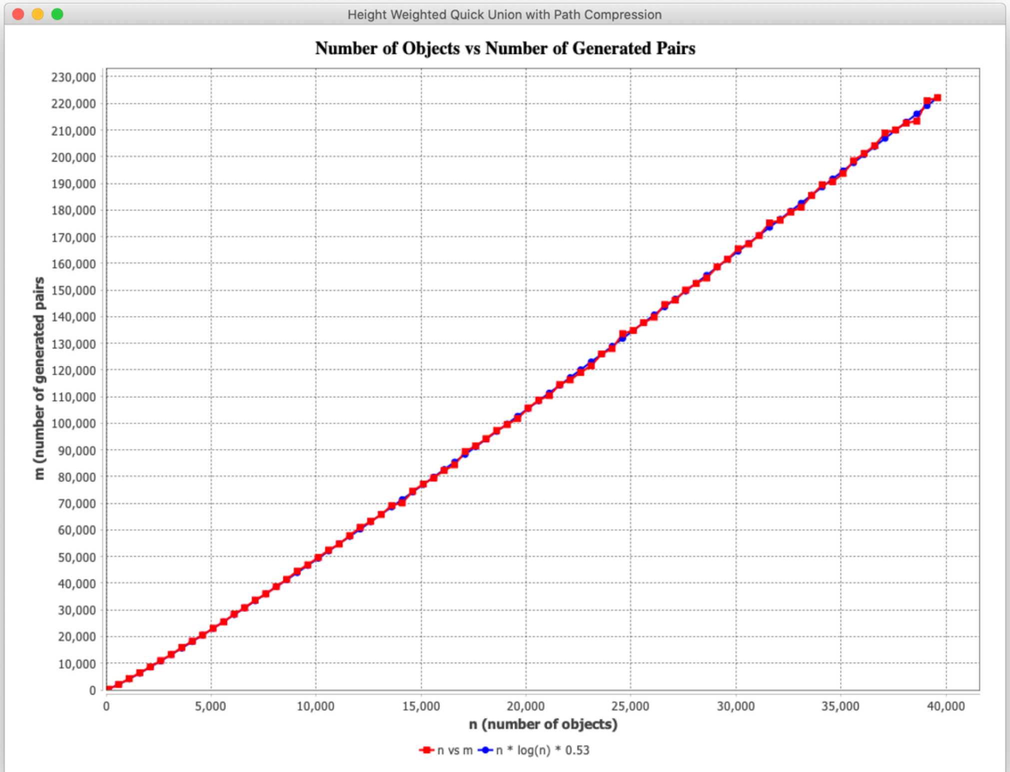

*Fig 4.1: Plotting the Number of Objects (n) vs Number of Generated Pairs (m) and the Number of Objects (n) vs n*log(n)*0.53 for the range n = 100 to n = 39600 with increments of 500*

The overlapping line graph for both n vs m and n vs n*log(n)*0.53 supports our conclusion that the relationship b etween n and m can be defined by:

$$m \approx 0.53 * \ n * \log{(n)}$$

- **Screenshot of Unit test passing**

UF_HWQUPC_Test.java

Finished after 0.042 seconds

Runs: 13/13 Errors: 0 Failures: 0

edu.neu.coe.info6205.union_find.UF_HWQUPC_Test [Runner: JUnit 4] (0.001 s)
- testIsConnected01 (0.001 s)
- testIsConnected02 (0.000 s)
- testIsConnected03 (0.000 s)
- testFind0 (0.000 s)
- testFind1 (0.000 s)
- testFind2 (0.000 s)
- testFind3 (0.000 s)
- testFind4 (0.000 s)
- testFind5 (0.000 s)
- testToString (0.000 s)
- testConnect01 (0.000 s)
- testConnect02 (0.000 s)
- testConnected01 (0.000 s)