Manik Kumar (001063023) – Section 02

INFO 6205 - Fall 2020

Program Structures & Algorithms

Assignment 4

1. Task:

- Implement two weighted quick union problems with tree height and tree size as the weight parameters.
- Implement two weighted quick union with path compression problems, a two-pass implementation(already exists in the repository), and a single pass implementation with the grandparents' fix.
- Benchmark all the above implementations and compare them.

2. Output

To produce the desired output, I coded the quick union problem to be run and benchmarked using the Benchmark_Timer class for 8 times (i.e. ran my count() function), for the number of values ranging from 10,000 to 1,280,000 elements. For each number of elements, the count() function was ran for 20 times and the resulting time in milliseconds was averaged over. Each time, the average final depth of the tree was calculated as well.

The following tables represent the values generated by the program.

Number of Objects(n)	Weighted UF by Size	Weighted UF by height	WUF with path compression	WUF with path compression and grandparents fix
10000	3.75ms	2.55ms	2.25ms	2.6ms
20000	5.15ms	5.65ms	4.1ms	3.7ms
40000	12.15ms	10.05ms	8.25ms	8.05ms
80000	28.0ms	27.1ms	18.0ms	17.9ms
160000	68.9ms	64.3ms	44.35ms	44.4ms
320000	162.6ms	147.65ms	91.45ms	92.05ms
640000	342.2ms	354.2ms	189.95ms	213.95ms
1280000	868.8ms	835.15ms	504.85ms	487.1ms

Table 1: Number of Pairs (m) and benchmark time for different implementations.

Number of Objects(n)	Average Depth Weighted UF by Size	Average Depth Weighted UF by height	Average Depth WUF with path compression	WUF with path compression and grandparents fix
10000	6	7	2	1
20000	6	7	2	2
40000	7	8	1	1
80000	7	8	2	2
160000	9	8	2	1
320000	8	9	2	2
640000	8	9	2	2
1280000	9	9	2	2

Table 2: Number of Pairs (m) and average tree depth for different implementations.

Example terminal output:

```
2020-10-12 00:02:50 INFO  Benchmark_Timer - Begin run: Benchmark for Weighted UF
based on size with 20 runs Time taken: 3.75ms
2020-10-12 00:02:50 INFO  Benchmark_Timer - Begin run: Benchmark for Weighted UF
based on height with 20 runs Time taken: 2.55ms
2020-10-12 00:02:51 INFO  Benchmark_Timer - Begin run: Benchmark for Weighted UF Path
compression  with 20 runs Time taken: 2.25ms
2020-10-12 00:02:51 INFO  Benchmark_Timer - Begin run: Benchmark for Weighted UF Path
compression Grandparent fix with 20 runs
...
...
...
2020-10-12 00:04:12 INFO  Benchmark_Timer - Begin run: Benchmark for Weighted UF Path
compression  with 20 runs Time taken: 504.85ms
2020-10-12 00:04:23 INFO  Benchmark_Timer - Begin run: Benchmark for Weighted UF Path
compression Grandparent fix with 20 runs Time taken: 487.1ms

Depth for WUF with size and n = 10000 is = 6
Depth for WUF with height and n = 10000 is = 7
Depth for WUF and CP and n = 10000 is = 2
Depth for WUF and CP with Grandparent Fix and n = 10000 is = 1
...
...
...
Depth for WUF with size and n = 1280000 is = 9
Depth for WUF with height and n = 1280000 is = 9
Depth for WUF and CP and n = 1280000 is = 2
Depth for WUF and CP with Grandparent Fix and n = 1280000 is = 2
```

3. Observation

From the outputs above, we could plot the following graph to depict the relationship between objects(n) and time taken.

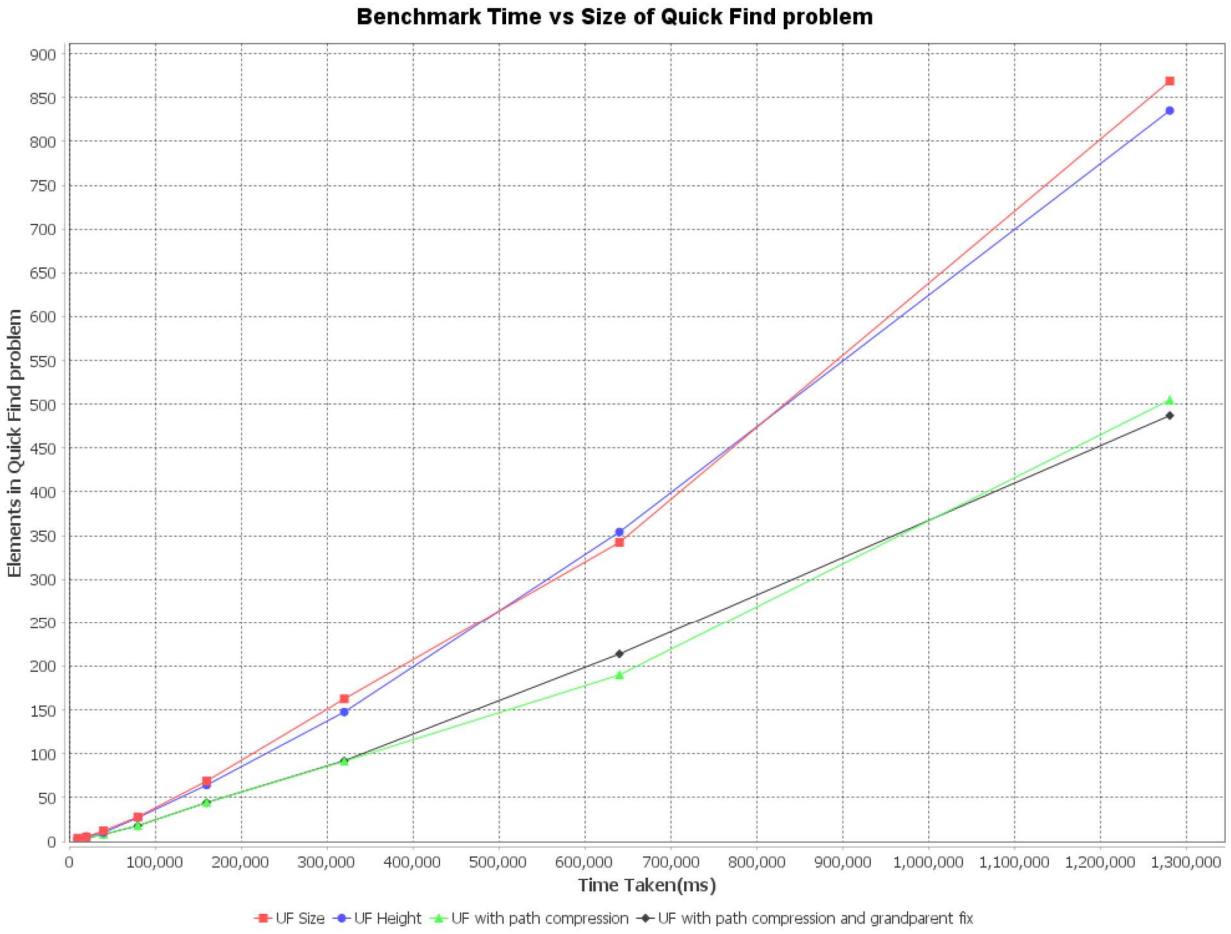

Figure 2: Number of Objects vs Time taken in different implementations.

The first observation we can make is that the plot for weighted quick union with tree size is extremely similar to the plot for weighted quick union with tree height. This is because merging by height is essentially the same as merging by the size of the tree. This is because of the direct positive relationship between the height and the size of the tree; height will be *log(n)* if the size of the tree is *n*. And the find operation looks for the parent, *i.e.* it only really traverses the height of the tree, which will always be *log(n)* regardless of if we are merging by tree size or tree height.

The second observation we make is that both our weighted quick union with path compression cases have better performance than the cases without path compression.

This is because weighted quick union with path compression can be solved in *n log*(n)* time, where *log*(n)* is an iterated log function. This can be proven through the inverse Ackerman equation.

$$log*(n) = \begin{cases} 0 & if\ n \leq 1 \\ 1 + log*(\log(n)) & if\ n > 1 \end{cases}$$

The third observation we make is that the in case of weighted quick union with path compression, comparing the implementation that exists in the repository with the implementation with the grandparents' fix, the performance and the final average depth of the tree is extremely similar. This is because the depth of the tree is extremely tiny and similar in both cases as the algorithms keep the tree almost completely flat in both the cases, hence the find operation takes very similar amounts of time. Hence, they do not have a significant performance difference.

4. Conclusion

- Weighted quick union with size and quick union by height are essentially the same thing.
- Weighted quick union with path compression is significantly more performant than simply a weighted quick union.
- Weighted quick union with path compression with a two-pass implementation has similar performance to weighted quick union with path compression with path halving (one pass, or grandparent implementation).

Files used in this assignment are:
UF_Alternative_Benchmarking.java
UF_HWQUPC.java
WeightedUF.java