*Machine Used: astro Compiler Options: makefile only*

**0**. In both union\_find0.cpp and union\_find1.cpp, the 'test' function recieves an object of type Union\_Find. In both cases, this class is the superclass for all other classes defined: Quick\_Find, Quick\_Union, Weighted\_Quick\_Union, and Weighted\_Quick\_Union\_with\_Path\_Compression. In each of these subclasses, there are new implementations of Union\_Find's member functions 'find' or 'union', sometimes both. In union\_find0.hpp, the 'find' and 'union' are declared as virtual in Union\_Find. This allows each of Union\_Find's subclasses to override these functions if needed, and manifests a virtual function table to keep track of overriden member functions.

This means that when a class that inherits from Union\_Find is passed into the 'test' function as a Union\_Find object, there is a virtual function table that the compiler uses to call the appropriate overriden member function when a call is made.

However, in union\_find1.hpp, 'find' and 'union' are not declared as virtual in the Union\_Find class, so when the subclasses are passed through the 'test' function as Union\_Find objects, the compiler has no relevant virtual function table to reference and treats the passed object as a Union\_Find class when a call is made - calling Union\_Find's implementation of 'union' and 'find'.

**1.**

|  |  |  |
| --- | --- | --- |
| Example 1A: Vertices: 100000, Edges: 100000 | | |
| - | Union Time | Connection time |
| Quick Find | 9131 ms | 1 ms |
| Quick Union | 1377 ms | 16830 ms |
| Weighted Quick Union | 4 ms | 6 ms |
| WQU with path compression | 5 ms | 5 ms |

|  |  |  |
| --- | --- | --- |
| Example 1B: Vertices: 100000, Edges: 50000 | | |
| - | Union Time | Connection time |
| Quick Find | 3640 ms | 0 ms |
| Quick Union | 1 ms | 5 ms |
| Weighted Quick Union | 2 ms | 2 ms |
| WQU with path compression | 2 ms | 2 ms |

|  |  |  |
| --- | --- | --- |
| Example 1C: Vertices: 50000, Edges: 100000 | | |
| - | Union Time | Connection time |
| Quick Find | 2756 ms | 1 ms |
| Quick Union | 3163 ms | 14915 ms |
| Weighted Quick Union | 4 ms | 6 ms |
| WQU with path compression | 4 ms | 4 ms |

|  |  |  |
| --- | --- | --- |
| Example 1D: Vertices: 200000, Edges: 200000 | | |
| - | Union Time | Connection time |
| Quick Find | 36893 ms | 4 ms |
| Quick Union | 6498 ms | 79446 ms |
| Weighted Quick Union | 11 ms | 14 ms |
| WQU with path compression | 12 ms | 12 ms |

Using one hundred thousand vertices and edges as our baseline graph (example 1A), we can immediately identify a few predicted behaviors based on the different implementations of union-find. It is immediately clear that weighted quick union with path compression is the most efficient implementation by far, beating out quick-find and regular quick-union by several magnitudes. However, the decreased time for execution that came as a result of path compression seems to be very marginal; much of the improvement for our ‘winning’ algorithm comes from the augmentation of weight to the connection and union process.

Beyond that, we may observe some other expected behavior – in a very sparse graph where edges has been decreased from the baseline (example 2A), all execution times improve, but the family of quick-union algorithms gain the most dramatic bump in performance, quick-find less so. This is because connection time is directly correlated with the number of edges in a graph, and quick-union’s most time intensive step is during connection.  
  
Conversely, when the graph is more dense (example 1C), we observe that the under-optimized implementation of connection in quick-union results in very poor performance overall. At the same time, quick-find experiences behavior slightly better than it did in the sparse graph due to the fewer vertices that mean fewer array lookups.

We’ll look at scalability in #2.

**2.**

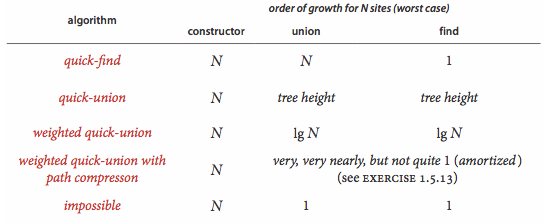
|  |  |  |
| --- | --- | --- |
| Example 2A: Vertices: 100000, Edges: 100000 | | |
| - | Union Time | Connection time |
| Quick Find | 9068 ms | 0 ms |
| Quick Union | 1489 ms | 17355 ms |
| Weighted Quick Union | 3 ms | 4 ms |
| WQU with path compression | 3 ms | 4 ms |

|  |  |  |
| --- | --- | --- |
| Example 2B: Vertices: 100000, Edges: 50000 | | |
| - | Union Time | Connection time |
| Quick Find | 3641 ms | 0 ms |
| Quick Union | 1 ms | 4 ms |
| Weighted Quick Union | 1 ms | 1 ms |
| WQU with path compression | 1 ms | 1 ms |

|  |  |  |
| --- | --- | --- |
| Example 2C: Vertices: 50000, Edges: 100000 | | |
| - | Union Time | Connection time |
| Quick Find | 2728 ms | 0 ms |
| Quick Union | 3443 ms | 15424 ms |
| Weighted Quick Union | 3 ms | 4 ms |
| WQU with path compression | 3 ms | 3 ms |

|  |  |  |
| --- | --- | --- |
| Example 2D: Vertices: 200000, Edges: 200000 | | |
| - | Union Time | Connection time |
| Quick Find | 36575 ms | 0 ms |
| Quick Union | 6893 ms | 80554 ms |
| Weighted Quick Union | 9 ms | 9 ms |
| WQU with path compression | 8 ms | 9 ms |

All of the observations about the various flavors of union-find hold for #2 and #3 – quirks between these implementations are held for #4 and #5. Looking at how these execution times should scale, lets look at a resource provided by Sedgewick and Wayne:



Take note: the above are each for one operation, and so total complexity should be multiplied by N. Our reference will be the doubling test between example 2A and 2D. Quick-find grows predictably at a quadratic rate (9068 \* 4 ~ 36575), quick-union’s growth is difficult to verify but will be quadratic in the worst case so its growth works seems reasonable. Weighted quick-union and WQU with path compression are both a little higher than doubled, with weighted quick-union being consistently a hair slower. For the most part, all four of these algorithms scale as expected.

**3.**

|  |  |  |
| --- | --- | --- |
| Example 3A: Vertices: 100000, Edges: 100000 | | |
| - | Union Time | Connection time |
| Quick Find | 9134 ms | 0 ms |
| Quick Union | 1488 ms | 17363 ms |
| Weighted Quick Union | 3 ms | 4 ms |
| WQU with path compression | 4 ms | 2 ms |

|  |  |  |
| --- | --- | --- |
| Example 3B: Vertices: 100000, Edges: 50000 | | |
| - | Union Time | Connection time |
| Quick Find | 3639 ms | 0 ms |
| Quick Union | 1 ms | 4 ms |
| Weighted Quick Union | 1 ms | 1 ms |
| WQU with path compression | 2 ms | 1 ms |

|  |  |  |
| --- | --- | --- |
| Example 3C: Vertices: 50000, Edges: 100000 | | |
| - | Union Time | Connection time |
| Quick Find | 2755 ms | 0 ms |
| Quick Union | 3441 ms | 15417 ms |
| Weighted Quick Union | 3 ms | 3 ms |
| WQU with path compression | 2 ms | 1 ms |

|  |  |  |
| --- | --- | --- |
| Example 3D: Vertices: 200000, Edges: 200000 | | |
| - | Union Time | Connection time |
| Quick Find | 36868 ms | 0 ms |
| Quick Union | 6803 ms | 80599 ms |
| Weighted Quick Union | 8 ms | 9 ms |
| WQU with path compression | 9 ms | 5 ms |

Analysis from #1 and #2 hold for #3.

**4.**

Examples 1A, 2A, 3A, and 1D, 2D, 3D illustrate that implementation 2 (aka union\_find1) out performs implementation 1 (aka union\_find0) on both weighted quick-union and WQU with path compression. Similarly, implementation 3 (aka union\_find2) out performs implementation in the same way. This ranking the implementations fairly trivial:

**Implementation 3 > Implementation 2 > Implementation 1.**

**5.**

Implementation 1 suffers because of its use of virtual functions. The use of the virtual table to gain the correct reference to a function is consuming over thousands of operations. This is an example of dynamic binding, which significantly affects performance at runtime instead of at compile time. This is evidently the flaw because timing resembles implementation 2 when functions are simply rewritten in subclasses as in implementations 2 and 3. Implementation 3 is better than 1 or 2 because of its use of inline functions for connected and find. Making these short functions inline instructs the compiler to ‘inject’ the code directly where they are referenced, and not to define a new scope or instantiate a new stack upon their call. This improves execution over 1 and 2 and explains our hierarchy.