Empyrical Analysis of Different Union-Find Heuristics

Víctor Franco Sanchez

2022

Abstract
In this paper we'll analyze the efficacy and efficiency of different heuristics for union and compression of the Union-Find (Sometimes also called Disjoint Set) data structure. We propose two union heuristics (union by size and union by rank) and four compression heuristics (full compression, path halving, path splitting and path inversion - the latter of which has not been proven optimal). Both methods are also compared with the trivial "do nothing" heuristic strategy.

1 Introduction

The Union-Find data structure is a data structure meant to store and modify a finite number of disjoint sets. At initialization, every element it's in its own set, and after initialization the data structure allows for two operations: The union operation takes as input two elements and joins both their sets into one larger set, and the find operation returns a representative element of the set, element which is always the same for every element on the same set.

This data structure is a vital component of many algorithms, such as the Kruskal algorithm for finding the minimum spanning tree, or many maze generating algorithms, and therefore it is of utmost importance for both the union and the find operations to be as efficient as possible.

The Union-Find is implemented as a directed n-ary forest in which each node represents one element, and this node stores the parent node for this element. One node is said to have no parent if it itself it's its own parent. The representative of the set is the parent node of the tree in which the element is found.

Thus, the "find" operation can just recursively look upwards until you find a node without parent and return that, and the "union" operation can just take the two representatives of both elements and set one to be the parent of the other.

These two operations can be optimized: For example, if, while doing the "find" operation, you find a way of getting the elements closer to the representative, consequent searches will be much faster. Also, when doing the "union" operation, you have a choice for which one of the two you want to be the parent. If you somehow find a way of making the parent node the one with the tallest tree, you will keep the resulting tree as flat as possible.

This paper focuses on four different ways you can improve on the "find" operation, and two different ways in which you can improve on the "union" operation, and shows empirical results for the efficiency of each combination.

2 Implementation

The code was made in C++ for efficiency's sake, and the UnionFind data structure was wrapped into a class for cleanliness and modularity.

```
typedef unsigned long int node;
class UnionFind {
    private:
        std::vector<node> parent;
        unsigned long int num;
        unsigned long int ngroups;
        #if UALG == 1
        std::vector<unsigned int> size;
        #endif
        #if UALG == 2
        std::vector<int> rank;
        #endif
    public:
        UnionFind(unsigned long int n);
        node find(node n);
        void union_op(node n, node m);
        unsigned long int TPL();
        unsigned long int TPU();
        unsigned long int nGroups();
};
```

The constructor initializes the data structure, setting each element's parent to itself and initializing the ranks or sizes of every element if required for the current version.

```
UnionFind(unsigned long int n){
    parent = std::vector<node>(n);
    num = n;
    ngroups = n;
    for(int i = 0; i < n; ++i) parent[i] = i;
    #if UALG == 1
    size = std::vector<unsigned int>(n, 1);
    #endif
    #if UALG == 2
    rank = std::vector<int>(n, 1);
    #endif
}
```

2.1 Find method

If no compression algorithm is used, the find operation is rather simple, and much what you'd expect: You just go up, from parent to parent, looking for the element with no parent.

```
#if CALG == 0
node find(node n){
    while(parent[n] != n) n = parent[n];
    return n;
}
#endif
```

2.1.1 Full Compression

For full compression, every node in the path up to the representative changes its parent to the representative itself, making every node at a distance of 1 to the representative. To do so, if the parent of the element has no element, we can return such. Otherwise, recursively call the find function on the parent, set the parent equal to the representative found and return this representative.

```
#if CALG == 1
node find(node n){
    if(parent[parent[n]] == parent[n]) return parent[n];
    return (parent[n] = find(parent[n]));
}
#endif
```

Remember that in C++, the expression "$a = b$" assigns b to a and evaluates to b, and therefore, if $b = 3$, $(a = b) + 2$ evaluates to 5, all while assigning a to 3.

2.1.2 Path Halving

Path Halving is the idea of skipping every two parents, and setting the parent of every node in the process to its grandparent, thus essentially reducing the total path length in half. This method seems to do worse than full compression, but it shows two improvements over the algorithm: Mainly, the use of additional memory is O(1) instead of O(n), and because only one pass is needed instead of two it's also much more cache friendly. Not only that, but it's been proven to be asymptotically as good as full compression.

```
#if CALG == 2
node find(node n){
    while(parent[parent[n]] != parent[n])
        n = (parent[n] = parent[parent[n]]);
    return parent[n];
}
#endif
```

2.1.3 Path Splitting

Path splitting is the idea of splitting the path to the node in two, one path being the even elements and another being the odd elements, thus also splitting the distance to the representative in half. The improvements of this method over full compression are the same as with path halving.

```
#if CALG == 3
node find(node n){
    while(parent[parent[n]] != parent[n]){
        node tmp = parent[n];
        parent[n] = parent[parent[n]];
        n = tmp;
    }
    return parent[n];
}
#endif
```

2.1.4 Path Inversion

Many definitions for path inversion exist in the literature. For this implementation, consider the following definition:

Let $n_k n_{k-1} \dots n_4 n_3 n_2 n_1$ be the path from the element you're asking to find (n_k) to the representative (n_1). Nodes n_2 and n_3 will now point to n_1 as a parent, and nodes $n_{k-1} \dots n_4$ will now point to node n_k as a parent, which will consequently point to n_1 as a parent. This method is quite tempting, since it also requires only O(1) memory, and instead of halving the distance, it sets a maximal path length of 2. Also, notice how, for the same sequence, the depth of each node will be (at least) just as good using this method as if you were using path splitting or path halving. However, apparently, the proof for optimality does not extend for this strategy, since path inversions break the required invariant that every node has a greater rank than their children. That being said, it's difficult to see why this strategy should underperform when compared to the other strategies, and this is why this strategy is also being considered for this analysis.

```
#if CALG == 4
node find(node n){
    node a = parent[n], b = parent[a], c = parent[b];
    if(a == b) return a;
    if(b == c) return (parent[n] = b);
    while(parent[c] != c){
        parent[a] = n;
        a = b; b = c; c = parent[c];
    }
    parent[n] = c;
    return (parent[a] = c);
}
#endif
```

2.2 Union method

The union function just finds the representatives of the two elements to join, decreases the number of groups off the structure if necessary and then executes some union algorithm following some criterion.

```
void union_op(node n, node m){
    n = find(n);
    m = find(m);
    if(n == m) return;
    --ngroups;
    // Union Algorithm
}
```

The simplest, silliest criterion would be to just set the parent of the first element to the second element, regardless of anything else:

```
// Union Algorithm
#if UALG == 0
parent[n] = m;
#endif
```

2.2.1 Union by Size

The idea of the union by size criterion is to make the parent the "largest" tree, in the sense that "largest" means the one with the most number of children. This makes sense, since the more nodes a tree has, the more likely it is to be taller, but there's no theoretical guarantee that the largest tree is actually the tallest.

One important thing to note is that the only elements that need to have the correct size are the representatives of each group, every other element is irrelevant. This is important, since the compression algorithms will move elements around, changing the sizes of intermediate trees, but they will never change the size of the tree of the representative.

```
// Union Algorithm
#if UALG == 1
if(size[n] > size[m]) {
    parent[m] = n;
    size[n] += size[m];
} else {
    parent[n] = m;
    size[m] += size[n];
}
#endif
```

2.2.2 Union by Rank

If the size does not guarantee optimality, why don't we just store height? Well, the point is that it's difficult to determine the height of a tree, once the compression algorithms are introduced. Nevertheless, we can instead store an upper-bound to the height, called the "rank", which will be equal to the height as if no compression algorithm was run. This union heuristic, together with either full path compression, path halving or path splitting give an optimal amortized cost of $O(\alpha(n))$, being α the inverse of $Ack(n, n)$, being Ack the Ackermann function. α grows so absurdly slowly that α(Number of atoms in the universe) is less than 5, making the cost, in practice, constant.

```
// Union Algorithm
#if UALG == 2
int dif = rank[n] - rank[m];
if(dif < 0) parent[n] = m;
else if(dif > 0) parent[m] = n;
else {
    parent[n] = m;
    ++rank[m];
}
#endif
```

2.3 Metric Collection

The number of pointer updates was collected explicitly, increasing a counter at every time a pointer was updated. For collecting the runtime instead of the metrics, this piece of code was removed.

The total path length was computed by iterating over the vector of elements and finding how many iterations it took to find the parent - without applying any sort of compression. Of course, during runtime collection, this function is never called.

Neither of these implementations is particularly efficient, but because these were used for metric acquisition and not for runtime measurements, this was deemed acceptable.

2.4 Instance Generation

To generate random instances, a number of elements N and a random seed is requested as input. Then, random pairs of elements are generated, and we check whether they're present in a C++ Set, and if they're not we output the pair and store the pair in the set. We repeat until we ensure there is only one set remaining, thing that we ensure using a Union-Find (In fact, the exact same Union-Find we're benchmarking).

Originally, the instance generator generated all possible pairs, shuffled them and then chose the first bunch up until only one set was remaining. The problem with this approach is that it uses $O(N^2)$ memory, and for large values of N my computer couldn't handle it. The current solution, although there is no guarantee of termination, uses only $O(N+m)$ memory, being m the number of outputted pairs, which is worst expected case $O(N \log N)$. With this upgrade, much larger instances can be generated.

20 instances for each size ($N = 1000, 5000, 10000$) were generated, and these 60 instances were the same instances used for every test.

3 Metric Comparison

There were two metrics collected: The *Total Path Length* (TPL), corresponding to the sum of the distance of every element to the representative of its set, and the *Total Pointer Updates* (TPU) which is the number of times a pointer changes due to a compression algorithm. The pointer updates due to unions were omitted since by the end there would always be exactly $N-1$ pointer updates corresponding to unions regardless of union and compression strategy, being N the number of elements of the Union-Find.

The first obvious observation is that, if no compression is run, obviously the path lengths will be maximal and the number of pointer changes during compression will be zero. This is obviously true for every tried size of problem, being the sizes $N = 1000$ (Figure 1), $N = 5000$ (Figure 2) and $N = 10000$ (Figure 3). However, these plots are not really that useful, considering the absurdly high path length squashes the data in the plot.

Figure 1: Metrics for all the heuristics for N = 1000. Red: “Do nothing” compression algorithm, Yellow: Full Path Compression, Green: Path Halving, Cyan: Path Splitting, Blue: Path Inversion.

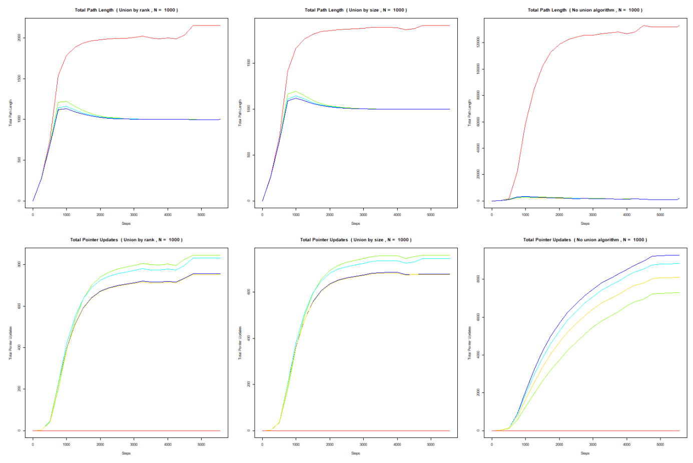


Figure 2: Metrics for all the heuristics for N = 5000. Red: “Do nothing” compression algorithm, Yellow: Full Path Compression, Green: Path Halving, Cyan: Path Splitting, Blue: Path Inversion.

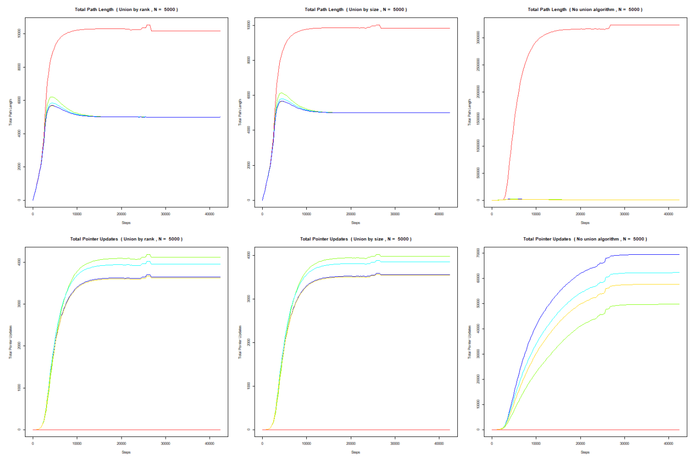
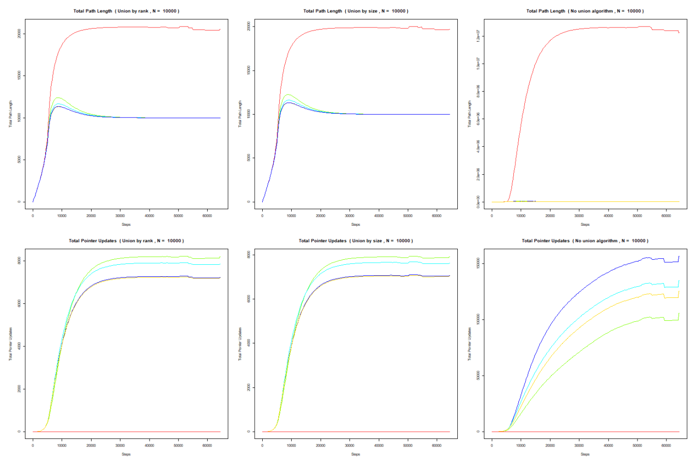


Figure 3: Metrics for all the heuristics for N = 10000. Red: “Do nothing” compression algorithm, Yellow: Full Path Compression, Green: Path Halving, Cyan: Path Splitting, Blue: Path Inversion.



For better clarity, we can plot the data removing the ”No compression” algorithm for better clarity. In these plots (Figure 4, 5 and 6) we can see how there’s a bump in the path length after between 10% and 20% of steps, probably because as more and more pairs are added, the less likely it is that we need to make a union, and therefore the path length only decreases by the path compression algorithms. Also, obviously, the number of pointer updates only steadily increases over the execution of the program - or at least it would if it weren’t for the fact that the executions were of an uneven number of steps, and thus the averaging of the data points did add some weird artifacts.

Nonetheless, we can actually observe some trends, like the fact that for both union by rank and union by length, full path compression and path inversion seem to be quite similar, and that path halving seems to do both the most number of pointer updates and tends to end up with the longest total path length, thing which doesn’t make much sense, but regardless.

This data is still pretty difficult to decipher, since the difference between the functions is quite tiny.

Figure 4: Metrics for all the heuristics for N = 1000 - Ignoring the do nothing compression algorithm. Yellow: Full Path Compression, Green: Path Halving, Cyan: Path Splitting, Blue: Path Inversion.

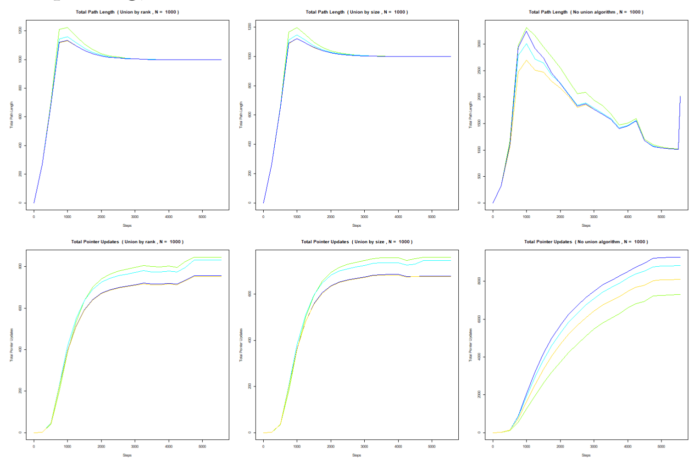


Figure 5: Metrics for all the heuristics for N = 5000 - Ignoring the do nothing compression algorithm. Yellow: Full Path Compression, Green: Path Halving, Cyan: Path Splitting, Blue: Path Inversion.

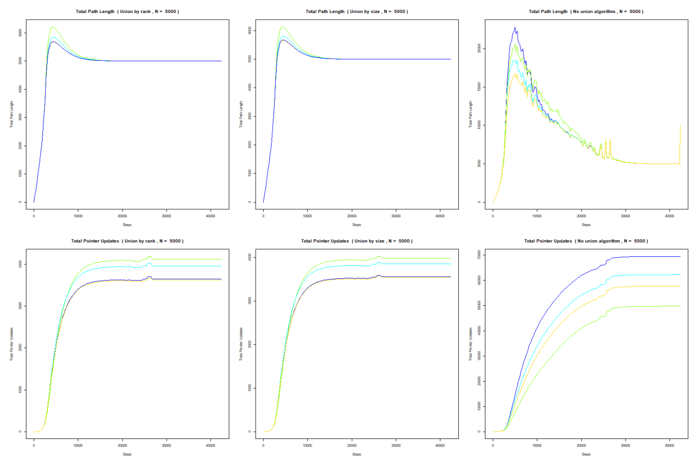
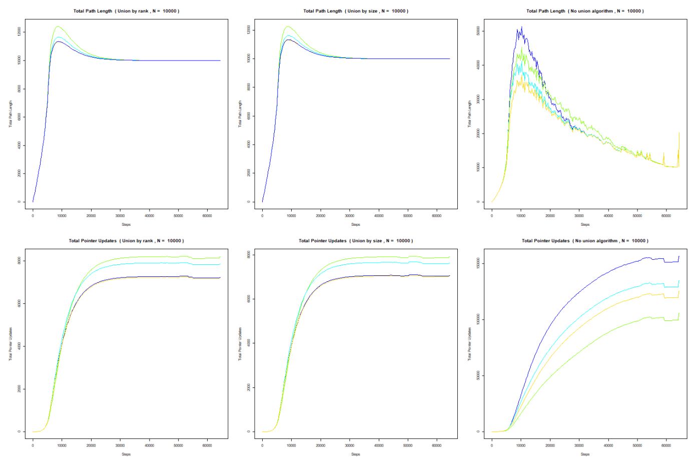


Figure 6: Metrics for all the heuristics for N = 10000 - Ignoring the do nothing compression algorithm. Yellow: Full Path Compression, Green: Path Halving, Cyan: Path Splitting, Blue: Path Inversion.



To more clearly see the results, we will subtract the full compression algorithm metrics from every function to create a baseline for the functions and that way see the differences more clearly (Figures 7, 8, 9). Now we can see that for union by rank and union by size, path inversion performs almost identically to full path compression both in total path length and in total pointer updates, while path halving uses the most number of pointer updates and ends up with the longest path length, and path splitting falls somewhere in between. For no union criterion, however, there seems to be some sort of change of roles, since it seems that path inversions have the greatest peaks in total path length, and end up using the most number of pointer updates. Also quite surprisingly, if no union criterion is chosen, path halving goes from using the most pointer changes to the least, using even less than full path compression.

Figure 7: Metrics for all the heuristics for N = 1000 - Ignoring the do nothing compression algorithm and subtracting the Full Compression metric. Yellow: Full Path Compression, Green: Path Halving, Cyan: Path Splitting, Blue: Path Inversion.

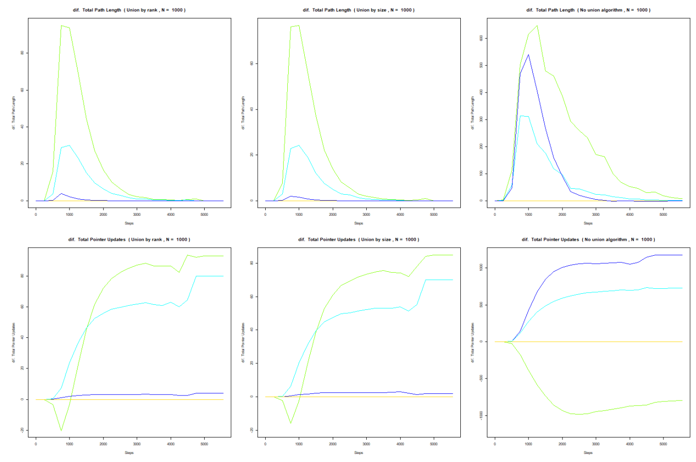


Figure 8: Metrics for all the heuristics for N = 5000 - Ignoring the do nothing compression algorithm and subtracting the Full Compression metric. Yellow: Full Path Compression, Green: Path Halving, Cyan: Path Splitting, Blue: Path Inversion.

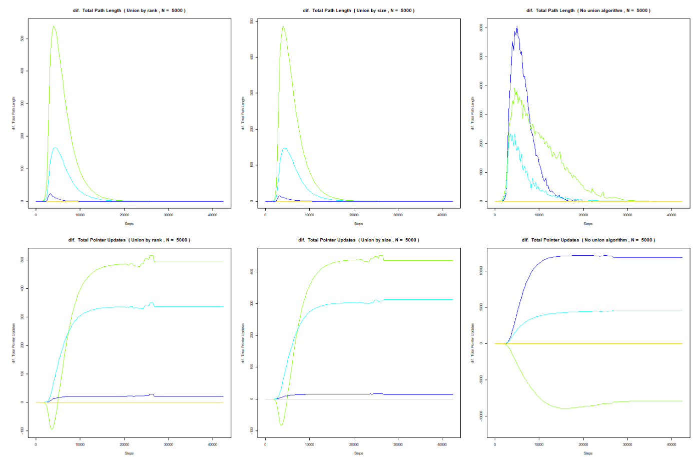
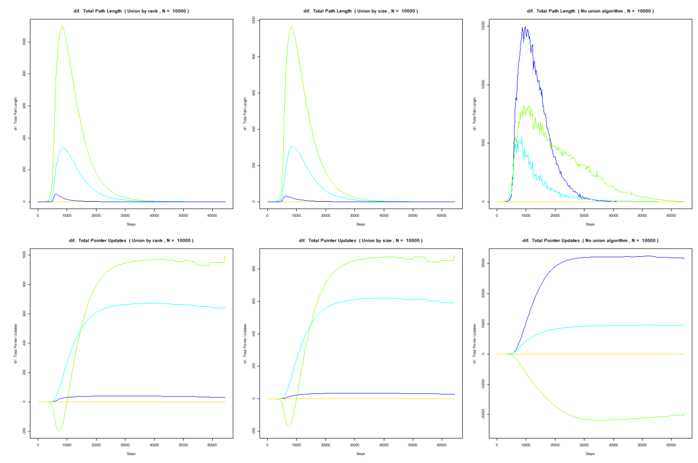


Figure 9: Metrics for all the heuristics for N = 10000 - Ignoring the do nothing compression algorithm and subtracting the Full Compression metric. Yellow: Full Path Compression, Green: Path Halving, Cyan: Path Splitting, Blue: Path Inversion.



4 Time Comparison

The following tables (Tables 1, 2 and 3) show the average running times of every method. Some unremarkable observations are the fact that no union criterion and no compression are always the slowest, but some other observations are worth making, such as the fact that path splitting seems to slightly underperform when compared to the other compression heuristics, and path inversion seems to slightly outperform the other algorithms. That being said, 20 runs per each size, only up to a size of 10000 is not that big of a data sample to extract conclusions, but still, this shows some promising results for path inversion, even without the optimality guarantees the rest of the methods have.

Due to time constraints, no statistical analysis was conducted to rule out the null hypothesis that none of these algorithms is better than any of the other, and this should be considered as future work.

Table 1: Execution times in ms, N=1000

	Full Compression	Path Inversion	Path Splitting	Path Halving	No Compression
Union By Rank	0.397835	0.366000	0.404070	0.437400	0.526535
Union By Size	0.413235	0.370250	0.423965	0.399155	0.476770
No Union Criterion	0.518260	0.458165	0.561325	0.475670	4.042575

Table 2: Execution times in ms, N=5000

	Full Compression	Path Inversion	Path Splitting	Path Halving	No Compression
Union By Rank	2.496115	2.747400	2.657960	2.609720	3.195910
Union By Size	2.667675	2.531870	2.512790	2.644965	3.262595
No Union Criterion	3.522175	3.251660	4.199750	3.314440	137.719850

Table 3: Execution times in ms, N=10000

	Full Compression	Path Inversion	Path Splitting	Path Halving	No Compression
Union By Rank	5.567585	5.260790	6.032025	5.887235	7.149945
Union By Size	6.113980	5.476620	5.416980	5.481810	6.627275
No Union Criterion	7.335860	6.977460	8.935445	6.938330	566.535100

5 Conclusions

In terms of metrics, the full compression algorithm for compression seems to outperform the rest, and it's difficult to compare whether union by rank or by size is better. One might think union by size to be better since it uses less pointer updates and has on average shorter total path lengths. However, when measuring time, union by rank seems to perform the best, and full compression is not always the optimal strategy. Instead, it seems that path inversion - the close second in terms of metrics, outperforms full compression in time, probably because it's a more cache friendly algorithm.

This paper does not provide absolute evidence for the fact that this is the case. Instead, it just provides an argument in favor of this conclusion. Further research needs to be conducted to be able to say anything conclusive.

6 Source Code

The source code for this project, together with the scripts and even the exact benchmarks used, can be found in the following link:
https://github.com/hectobreak/UnionFindMetrics