UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Advanced Data Structures

Computer Science Department

Union-Find data structure: An empirical analysis

Author:
Alex Herrero

Professor:
Conrado Martínez

March 16, 2025

Contents

List of Figures

List of Tables

1 Introduction

Given a non-empty set $\mathcal{A}$, a partition Π of $\mathcal{A}$ is defined as a collection of subsets of $\mathcal{A}$, i.e., $\Pi = \{A_1, A_2, \ldots, A_k\}$, such that:

- $\bigcup_{i=1}^{k} A_i = \mathcal{A}$.
- $A_i \cap A_j = \emptyset$ for all $i \neq j$.

The elements of Π are called *classes*. Furthermore, we define an equivalence relation between two elements $a, b \in \mathcal{A}$ by stating that a and b are equivalent if and only if there exists some $A_i \in \Pi$ such that $a, b \in A_i$. In this case, we say that a and b are equivalent and denote this by $a \equiv b$.

One can trivially verify that this binary relation is reflexive, symmetric, and transitive. These properties allow us to define a *representative* for each class—an element that represents the entire class. By the reflexive, symmetric, and transitive properties, every other element in the same class is related to this *representative*.

This concept, previously established by mathematicians, is widely used in Computer Science to implement the Union-Find data structure. The Union-Find data structure is designed to efficiently store and manage a partition of a set $\mathcal{A}$. It provides two fundamental operations:

- `Find Operation:` Given an element $a \in \mathcal{A}$ find the representative of the class from which a belongs.
- `Union Operation:` Given two element $a \in A_i$ and $b \in A_j$ make a union of classes A_i and A_j. That is, if the Union-Find represents a partition Π, after doing the `Union` operation the data structure will represent a new partition Π' such that $\Pi' = (\Pi \setminus \{A_i, A_j\}) \cup (\{A_i \cup A_j\})$

2 Implementation

A C++ implementation has been made for the sake of efficiency of the Union-Find operations. A C++ class has been created for the Union-Find data structure (look at `UnionFind.hh` for a more detailed exploration of the class). It consists mainly of an array in which every element will either point to another element that belongs to the same class or it will be the representative. For the representative, depending on the union strategy they will be represented differently:

- With the Quick-Union strategy an element i is the representative of a class if $v[i] = i$.
- With the other strategies (union by rank/size) they will be represented by a negative number that indicates the size/rank multiplied by -1 (i.e. $v[i] = -size$ or $v[i] = -rank$).

The `main.cc` file requires, as input, the size of the data structure, a Union strategy (a natural number between 0 and 2, representing Quick-Union, Union by Weight, and Union by Rank, respectively), and a Path strategy (a natural number between 0 and 4, representing No Compression, Full Compression, Path Splitting, Path Halving and Type One Reversal (a new heuristic explained later)). After that, the program will repeatedly request two natural numbers (the elements to be merged) and perform the corresponding union operation. This process continues until the data structure consists of a single block, at which point the program terminates.

Every $\Delta = 250$ elements, the program will output information about the current TPL and TPU of the data structure. The TPU parameter is computed using a counter that increments by one each time a pointer switch occurs (see Appendix B), while the TPL is calculated by traversing the entire data structure (see Appendix C). Although this approach may not be the most efficient in terms of performance, the author chose this method for computing the TPL because the focus is on obtaining the value rather than optimizing execution time. Later, execution time will be measured separately, excluding this part of the computation.

To execute the program, one can compile it using the provided `Makefile` by running `make main` and then executing the program from the command line. For instance, suppose `file.txt` contains pairs of numbers (between 0 and 999) that, when processed, will lead to a single block in the Union-Find data structure. In that case, we can process these numbers using Union by Weight and Path Halving by executing the following command:

```
./main.exe 1000 1 3 < file.txt
```

For getting time results you just need to compile by running `make main_time`.

3 Experimental Setup

To generate random instances, a number n of elements and a seed s are requested as input. Then, this program generates a random number of different pairs (i, j) where $0 \leq i < j < n$ until the total number of pairs generated forms a single block in a Union-Find data structure.

The time complexity of this program is difficult to calculate, as there is no guarantee that the program will terminate. Since it generates pseudo-random different pairs, it might enter a loop where all the pairs generated are repetitions. However, this approach is better than the initial implementation, which involved generating all possible $\binom{n}{2}$ pairs—requiring $\Theta(n^2)$ time and space complexity—and then selecting random pairs until a single block is formed in the Union-Find structure. Using this new approach we use a `C++ set` to keep track of all different pairs, requesting only necessary memory.

For the following experiments, 20 instances of size $(n = 1000, 5000, 10000)$ have been created. Each instance was generated with a seed s that goes from 1964 (in honor of Union-Find data structure discover[1]!) to 1983. Each instance has been executed with the 15 possible different strategies and the data has been collected as follows:

1. For each size n, union strategy, and path strategy, execute all instances of size n.
2. We will average all values obtained for every $k \in \{\Delta, 2\Delta, 3\Delta, \ldots\}$ pairs processed of all instances with same size and heuristics, provided that at least five instances have reached the point of processing k pairs.

Although this process might be tedious, it can be automated by executing a single script! (See README).

Using these samples, we were able to process approximately 3,750 pairs of elements for $n = 1000$, 26,250 pairs for $n = 5000$, and 52,250 pairs for $n = 10,000$.

4 Results

4.1 Metric Comparisons

4.1.1 Path Strategy: No Compression

Using this technique, the minimization of paths between an element i and its representative r_i relies solely on the Union heuristic. Specifically, in the case of Quick-Union, Figure 1 shows that Quick-Union differs significantly from TPL compared to the other union techniques. This difference is expected, as no heuristic is used to minimize these paths.

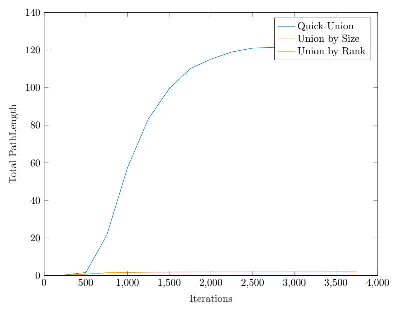
In fact, since the total path length is significantly larger when not using unions with heuristics, we can be certain that the asymptotic cost will be much higher. Indeed, although `union` operations are $\Theta(1)$, a `find` operation can theoretically take as much as $\Theta(n)$. However, we observe that the TPL stabilizes after approximately 50% of the pairs have been processed.

This is not surprising because, as more unions are performed, the probability that the selected pair is already connected increases, meaning the union operation has no effect and the TPL remains unchanged. Still, we can see that the TPL increases as n increases.

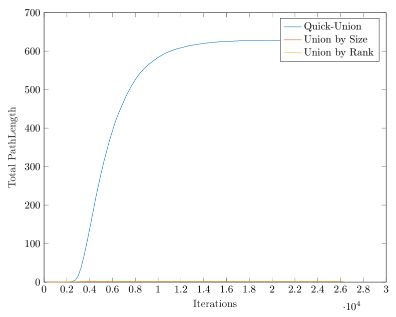
Taking a closer look at unions with heuristics, which are also shown in Figure 1, we observe two key differences compared to Quick-Union:

- TPL stabilizes sooner than in Quick-Union. Specifically, it reaches a *constant* value of approximately 2 after processing around 20%–30% of the pairs.
- Although TPL increases slightly as n grows, it consistently remains close to 2. In fact from a theoretical analysis we know that a block of size k has $O(\log k)$ height using these unions, which also helps to explain why we reach this stability *sooner*. This indicates that using heuristics in unions leads to overall stability in TPL and, consequently, in `find` operations.

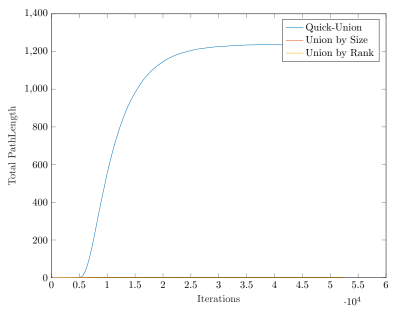
Between Union by Weight/Size, there is no significant difference between the two strategies, although Union by Size performs slightly better. However, both strategies significantly outperform Quick-Union without any path compression technique.



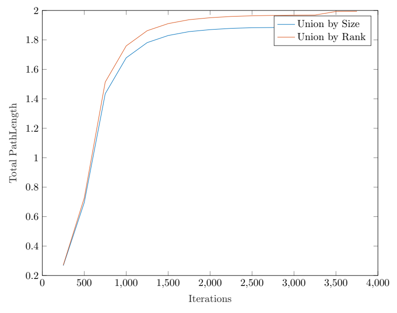
(a) Path Lengths with different union strategies with $n = 1000$



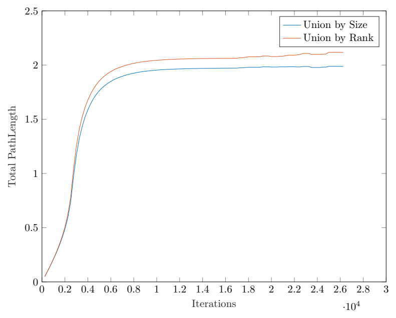
(b) Path Lengths with different union strategies with $n = 5000$



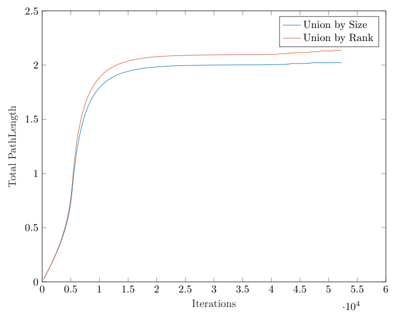
(c) Path Lengths with different union strategies with $n = 10000$



(d) Path Lengths with different union strategies with $n = 1000$ without Quick-Union



(e) Path Lengths with different union strategies with $n = 5000$ without Quick-Union



(f) Path Lengths with different union strategies with $n = 10000$ without Quick-Union

Figure 1: Total Path Length normalized with No Compression

4.1.2 Path Strategy: using Heuristics

Now let us introduce *Path Compression* in our analysis. This technique consists in, when a `find` operation is performed, rearranging pointers so that we expect that future `find` operations take less time. In this analysis we are going to consider the following path compression heuristics:

- The **Full Compression** heuristic ensures that every element in the path from i to the representative of the class r_i will point directly to r_i by traversing the path twice (first to identify r_i and second to rearrange the pointers). At the cost of traversing this path twice, we expect to speed up future find operations involving a class where this heuristic has been applied.
- Traversing a path twice might not always be the best option so, in order to speed up future find operations, we are going to decrement path lengths within the class by making that every node

points to its grandparent as long as this exists. That is what **Path Splitting** and **Path Halving** does. The only difference between Path Splitting and Halving is that in splitting we will always make every node point to its grandparent (as long as they exists) while in halving we are going to do that operation every other node (so we expect to end sooner a find operation).

- Similar to Full Compression we have **Type One Reversal** which, given a path $(i, n_1, n_2, \ldots, n_k, r_i)$ where i is an element and r_i is its representative, consists of the following: Every node from n_1 up to n_k will point to i and i will point to r_i. Although this might be strange, this heuristic has a similar idea to Full Compression but this time we are traverse a path only once instead of twice at the cost of not pointing directly to the representative. In this analysis we will only consider **Type One Reversal** but in [2] is defined a type k reversal for $k \geq 1$.

As expected, we see in Figure 4 that Quick-Union, once again, performs the worst. However, this time, full compression, path splitting, and path halving exhibit similar behavior in terms of TPL: it initially grows quickly but soon tends to converge to the TPL achieved using unions with heuristics.

Again, this should not be surprising, as increasing the number of iterations raises the probability that two pairs belong to the same class. For knowing if two elements belong to the same class we need to perform two `find` operations, and when these operations are executed with compression heuristics, they tend to *fix* paths within the trees, thereby reducing the IPL corresponding to the tree of such class.

Although full compression, splitting, and halving perform very similarly, we observe slight differences in the maximum peak of the TPL. In fact, the more compression applied in the heuristic, the lower the peak values become, though the difference remains unnoticeable in practice.

Still, it is interesting to examine Type One Reversal. Unlike previous heuristics, it does not converge to the TPL obtained with unions with heuristics; instead, it remains closer to a value approximately twice the peak values observed with other compression heuristics.

Type One Reversal is a tricky compression heuristic because performing a `find` operation in a class may result in a new tree representation with a higher internal path length than before the operation. For instance see Figure 2, where we decreased by one the path lengths of all subtrees rooted at node 4, but simultaneously increased by one the path lengths of all subtrees rooted at node 3.

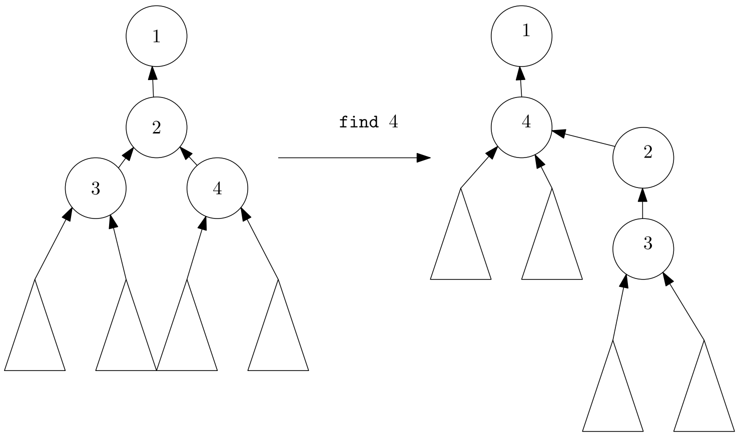


Figure 2: Example of Type One Reversal which may lead to a worst IPL

In fact, when Quick-Union with Type One Reversal reaches a *stable* point in total path length, it continuously improves and deteriorates in an oscillatory manner.

A more general view of Figure 2 can be found in Figure 3, where we start with a generic tree configuration using this path compression technique. This configuration consists of a representative, a single node pointing to the representative, and several subtrees pointing to this intermediate node.

In Figure 3, we illustrate a `find` operation that is always performed at the root of a subtree. If this was not initially the case, the node would become the root of the subtree after applying compression, ensuring that our representation remains valid. Additionally, we depict nodes 3 and 4 as having only two subtrees pointing to them. This simplification is solely for clarity in the figure; in reality, each node may have a different number of subtrees pointing to it.

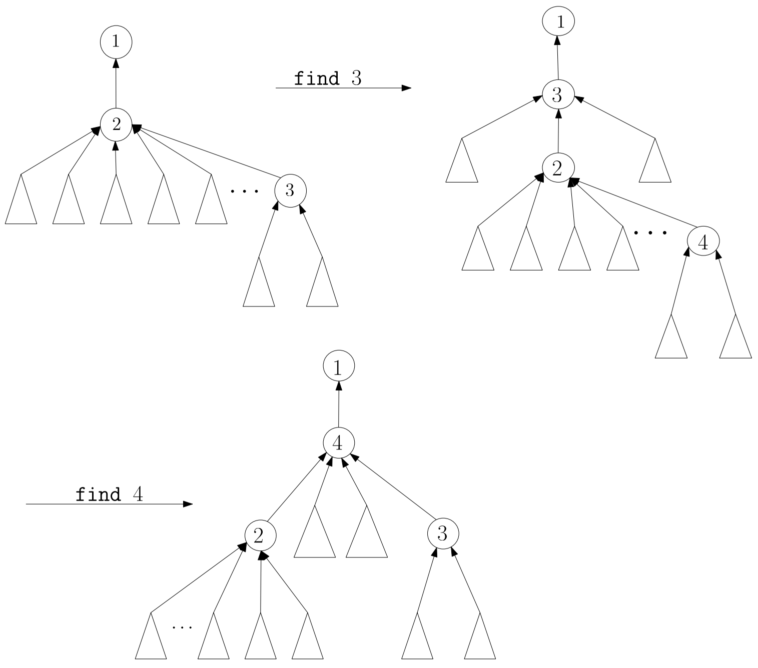


Figure 3: More accurate example of the representation of a tree after several `find` operations

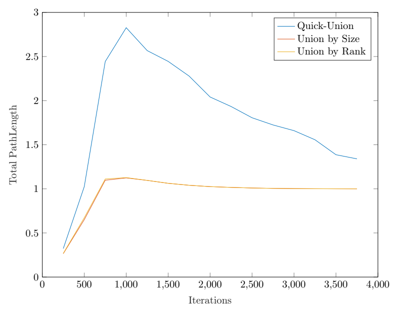
Now we see that performing a `find` operation will increase by one the distance of all nodes in a subtree to the representative while decreasing by one the distance in another subtree.

If we apply union heuristics, due to this adjustment and the randomness introduced in our analysis, each subtree will have approximately the same number of nodes. Consequently, when performing another `find` operation, it is more likely to occur in a node within the subtree rooted at 2 rather than any other node. As a result, we will then increase by one the distance of each node in the subtree rooted at 3 to the representative while decreasing by one the distance in the subtree rooted at 4.

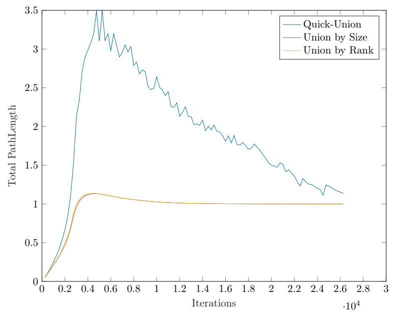
In fact, this behavior is observed in Figure 7, where the TPL remains between 2 and 3. If we do not attempt to balance each tree in the union heuristics, we may end up increasing the distance for more nodes than we decrease, potentially by one or even more.

Regarding pointer updates (Figures 6 and 7), we highlight the following observations:

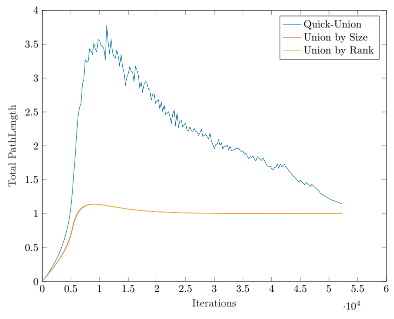
- Full Compression and Path Splitting perform a similar number of pointer updates, which is expected since all nodes along the same path will have their pointers updated.
- Path Halving performs slightly fewer pointer updates, which makes sense as it updates pointers for every other node.
- Type One Reversal always increases the number of pointer updates, as it consistently modifies pointers.
- The longer a path is, the more pointer updates are required. This explains why Quick-Union consistently performs the worst.



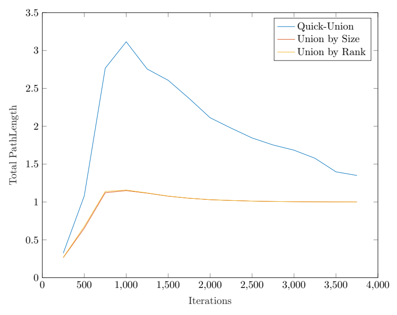
(a) Path Lengths with different union strategies with $n = 1000$ using Full Compression



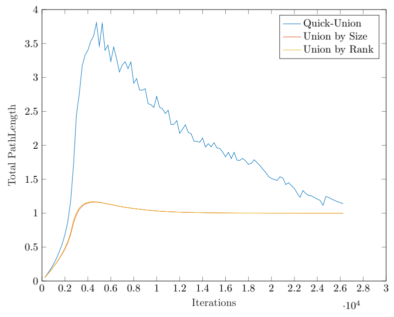
(b) Path Lengths with different union strategies with $n = 5000$ using Full Compression



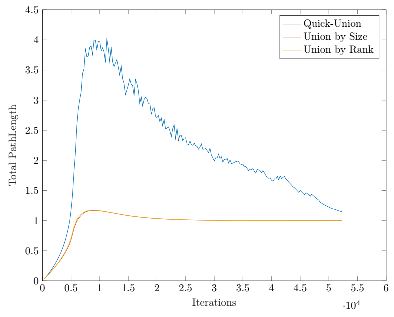
(c) Path Lengths with different union strategies with $n = 10000$ using Full Compression



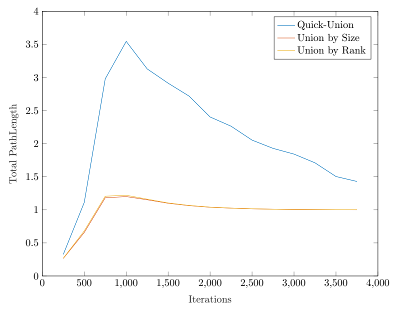
(d) Path Lengths with different union strategies with $n = 1000$ using Path Splitting



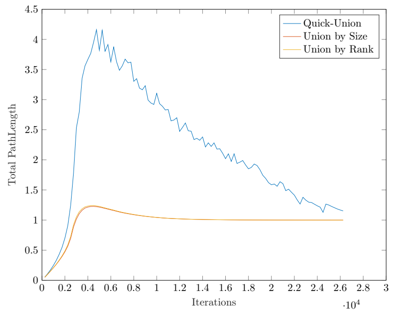
(e) Path Lengths with different union strategies with $n = 5000$ using Path Splitting



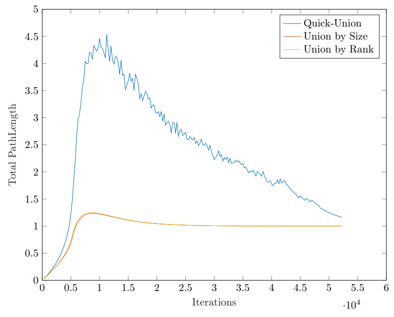
(f) Path Lengths with different union strategies with $n = 10000$ using Path Splitting



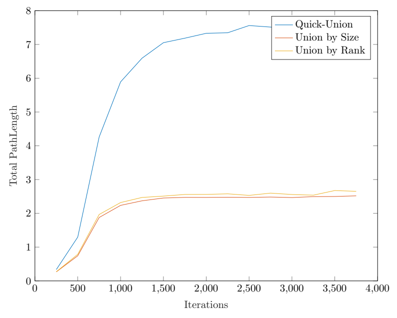
(g) Path Lengths with different union strategies with $n = 1000$ using Path Halving



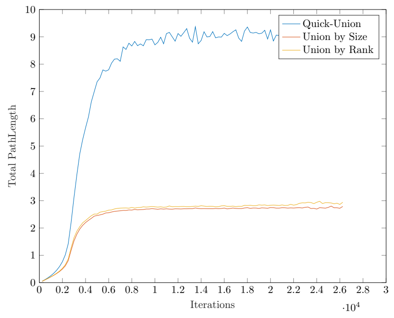
(h) Path Lengths with different union strategies with $n = 5000$ using Path Halving



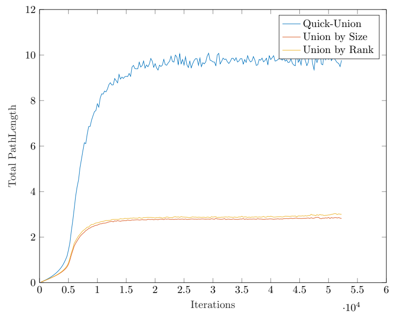
(i) Path Lengths with different union strategies with $n = 10000$ using Path Halving



(j) Path Lengths with different union strategies with $n = 1000$ using Type One Reversal

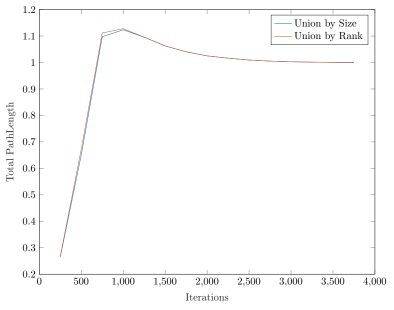


(k) Path Lengths with different union strategies with $n = 5000$ using Type One Reversal

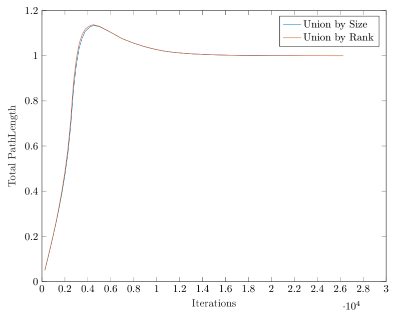


(l) Path Lengths with different union strategies with $n = 10000$ using Type One Reversal

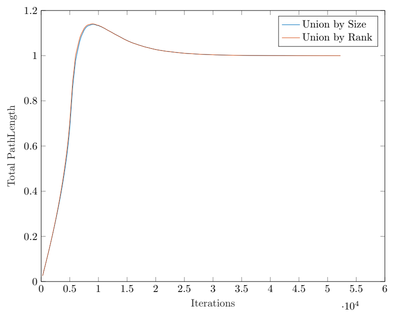
Figure 4: Total Path Length normalized for different heuristics



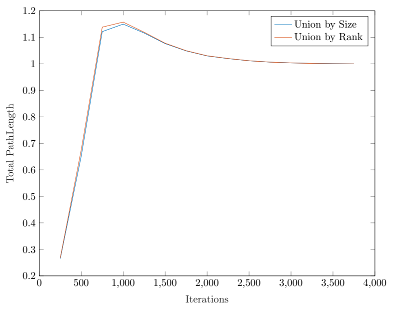
(a) Path Lengths with different union strategies with $n = 1000$ using Full Compression



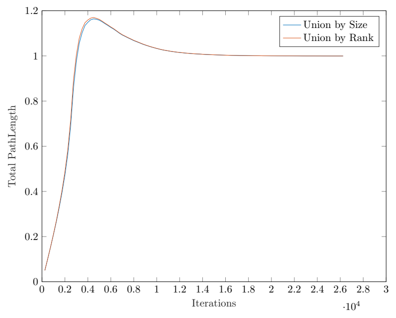
(b) Path Lengths with different union strategies with $n = 5000$ using Full Compression



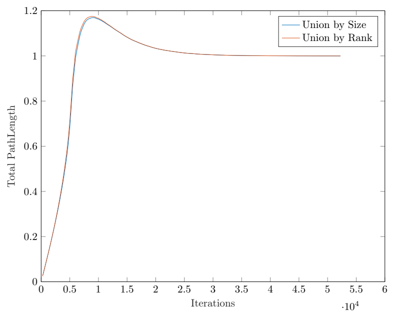
(c) Path Lengths with different union strategies with $n = 10000$ using Full Compression



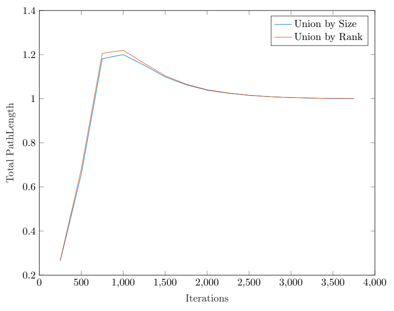
(d) Path Lengths with different union strategies with $n = 1000$ using Path Splitting



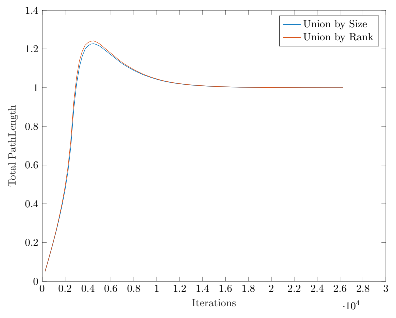
(e) Path Lengths with different union strategies with $n = 5000$ using Path Splitting



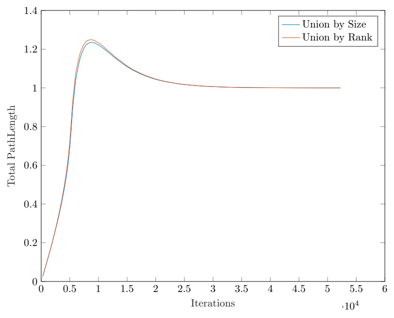
(f) Path Lengths with different union strategies with $n = 10000$ using Path Splitting



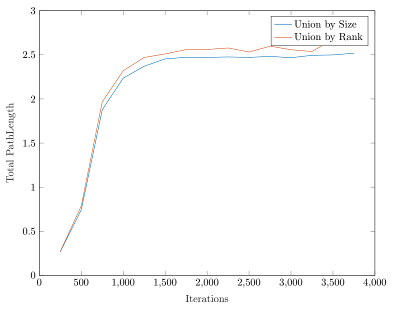
(g) Path Lengths with different union strategies with $n = 1000$ using Path Halving



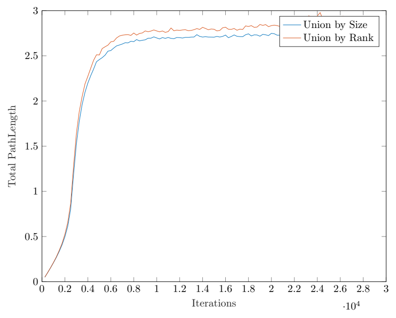
(h) Path Lengths with different union strategies with $n = 5000$ using Path Halving



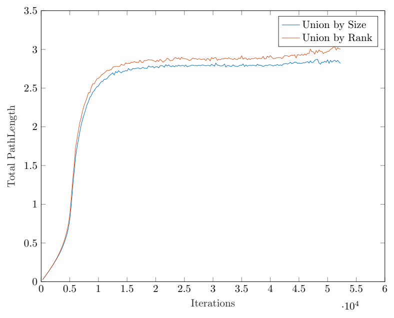
(i) Path Lengths with different union strategies with $n = 10000$ using Path Halving



(j) Path Lengths with different union strategies with $n = 1000$ using Type One Reversal

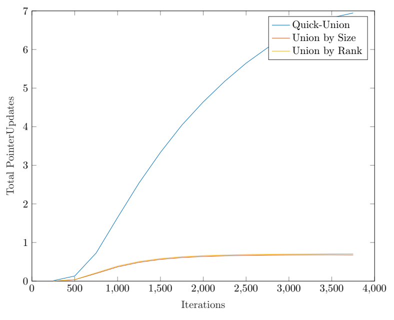


(k) Path Lengths with different union strategies with $n = 5000$ using Type One Reversal

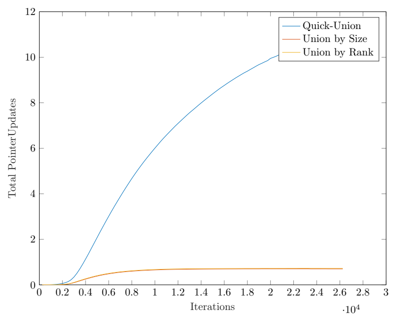


(l) Path Lengths with different union strategies with $n = 10000$ using Type One Reversal

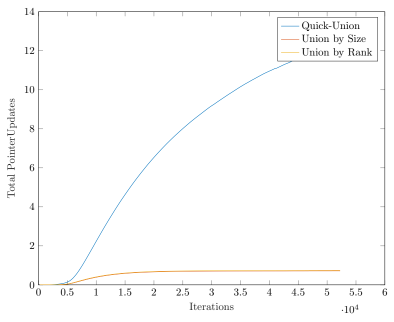
Figure 5: Total Path Length normalized for different heuristics without Quick-Union



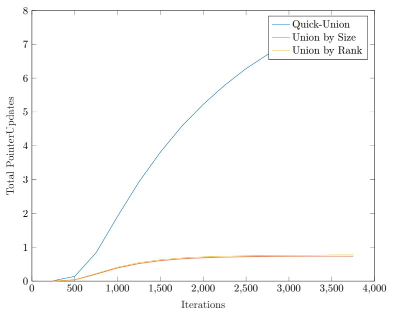
(a) Pointer Updates with different union strategies with $n = 1000$ using Full Compression



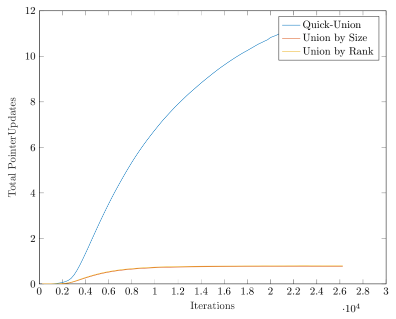
(b) Pointer Updates with different union strategies with $n = 5000$ using Full Compression



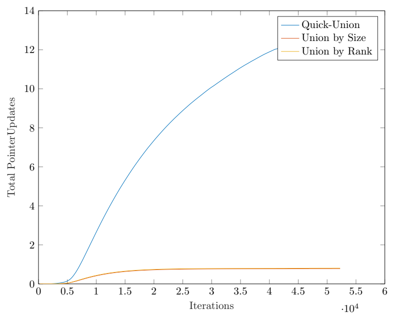
(c) Pointer Updates with different union strategies with $n = 10000$ using Full Compression



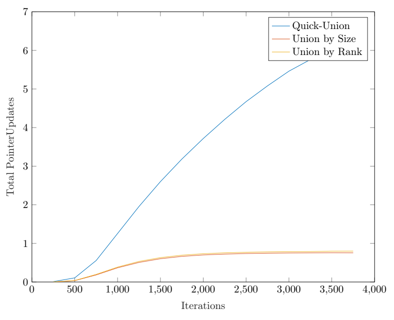
(d) Pointer Updates with different union strategies with $n = 1000$ using Path Splitting



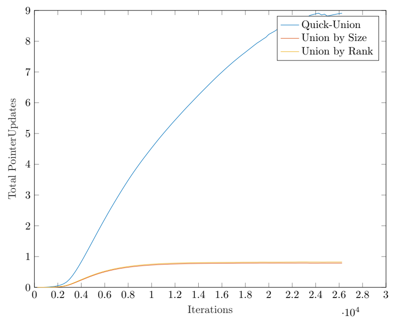
(e) Pointer Updates with different union strategies with $n = 5000$ using Path Splitting



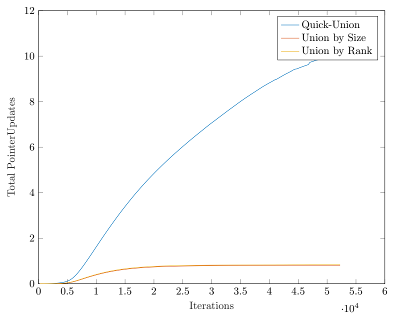
(f) Pointer Updates with different union strategies with $n = 10000$ using Path Splitting



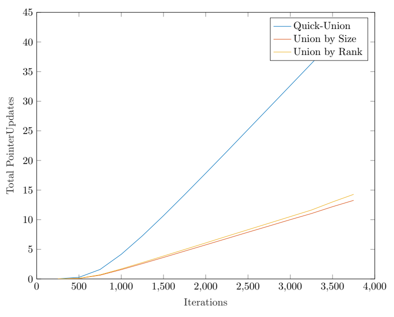
(g) Pointer Updates with different union strategies with $n = 1000$ using Path Halving



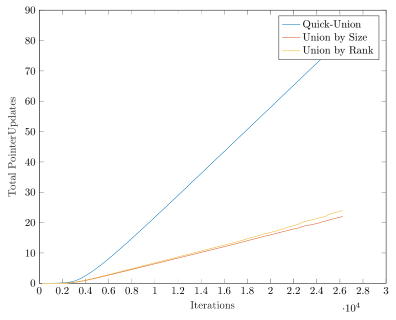
(h) Pointer Updates with different union strategies with $n = 5000$ using Path Halving



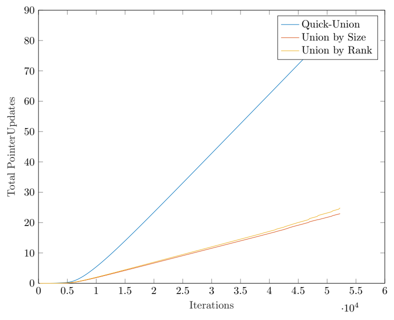
(i) Pointer Updates with different union strategies with $n = 10000$ using Path Halving



(j) Pointer Updates with different union strategies with $n = 1000$ using Type One Reversal

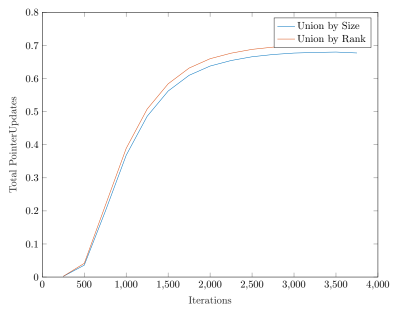


(k) Pointer Updates with different union strategies with $n = 5000$ using Type One Reversal

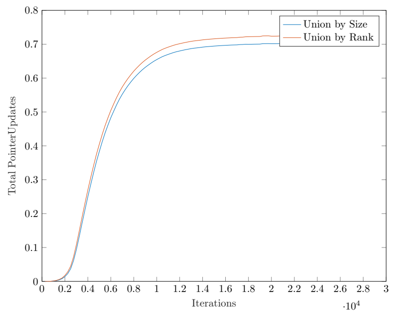


(l) Pointer Updates with different union strategies with $n = 10000$ using Type One Reversal

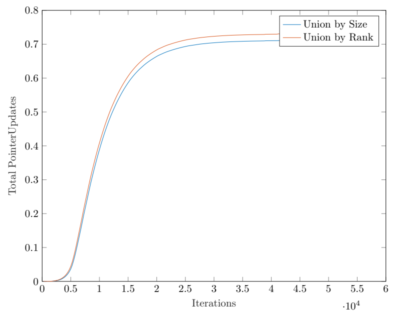
Figure 6: Total Pointer Update normalized using different heuristics



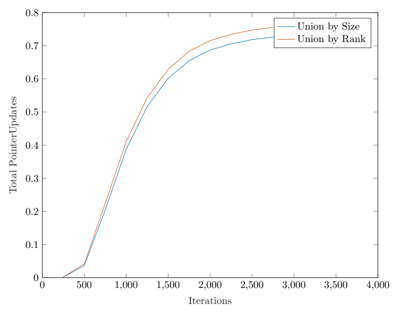
(a) Pointer Updates with different union strategies with $n = 1000$ using Full Compression



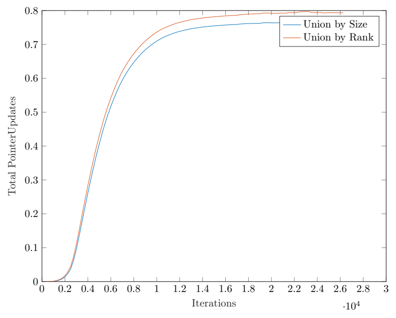
(b) Pointer Updates with different union strategies with $n = 5000$ using Full Compression



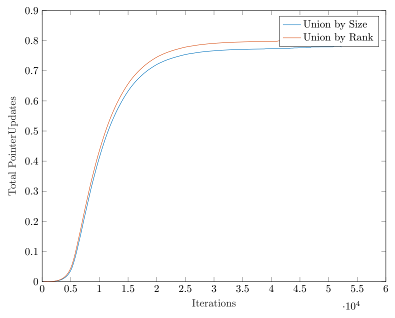
(c) Pointer Updates with different union strategies with $n = 10000$ using Full Compression



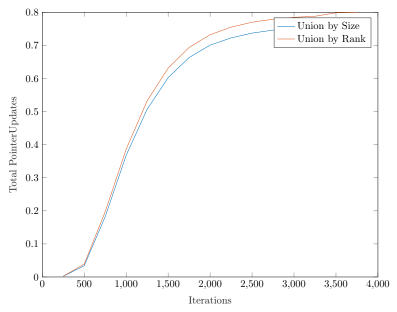
(d) Pointer Updates with different union strategies with $n = 1000$ using Path Splitting



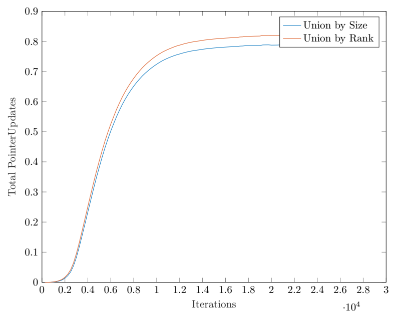
(e) Pointer Updates with different union strategies with $n = 5000$ using Path Splitting



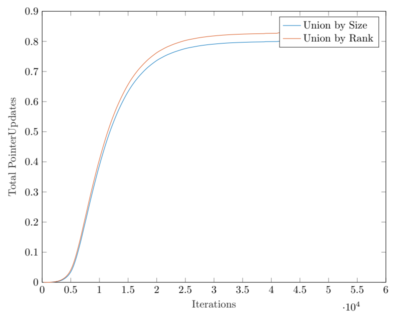
(f) Pointer Updates with different union strategies with $n = 10000$ using Path Splitting



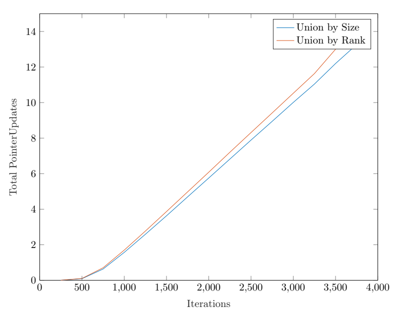
(g) Pointer Updates with different union strategies with $n = 1000$ using Path Halving



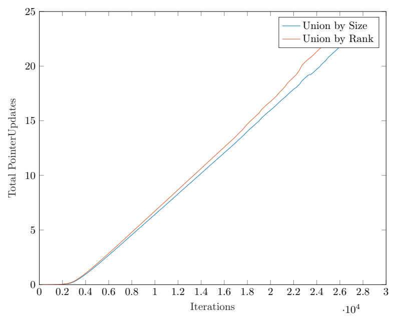
(h) Pointer Updates with different union strategies with $n = 5000$ using Path Halving



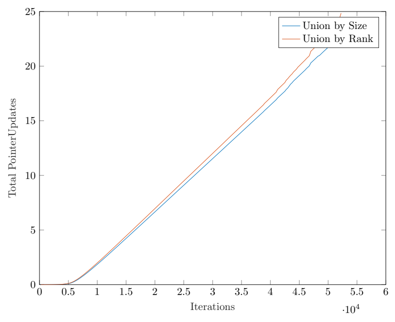
(i) Pointer Updates with different union strategies with $n = 10000$ using Path Halving



(j) Pointer Updates with different union strategies with $n = 1000$ using Type One Reversal



(k) Pointer Updates with different union strategies with $n = 5000$ using Type One Reversal

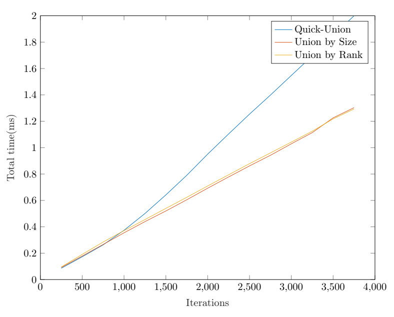


(l) Pointer Updates with different union strategies with $n = 10000$ using Type One Reversal

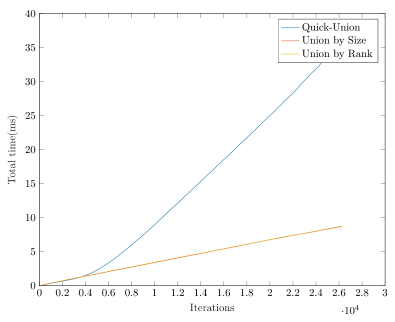
Figure 7: Total Pointer Update normalized without Quick-Union

5 Time Comparison

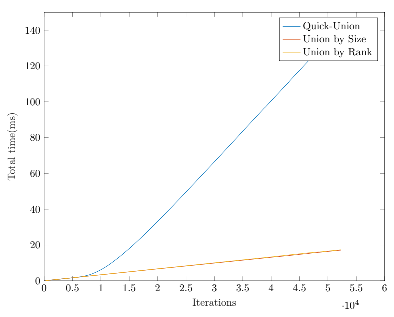
With the same approach explained in Section 3, times have been calculated. Figure 8 and 9 provides plots for the execution time counting only the time needed for doing merge operations (not the whole execution of the program!). As expected, Quick-Union without path compression performs the worst when we compare with weighted unions (the difference is even more clear as we increase the size of the data structure). Still, with weighted unions we see that time is getting more close to each other as we increase the size (but this is not a surprise, as time execution is related to average TPL and TPU which has been calculated in the previous section, where we saw that both metrics are similar using weighted unions).



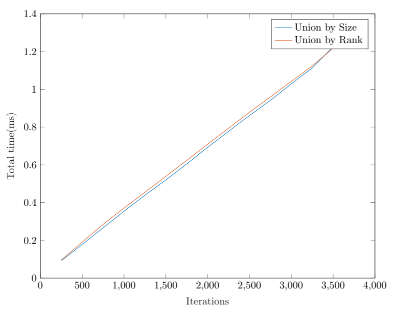
(a) Total time with different union strategies with $n = 1000$



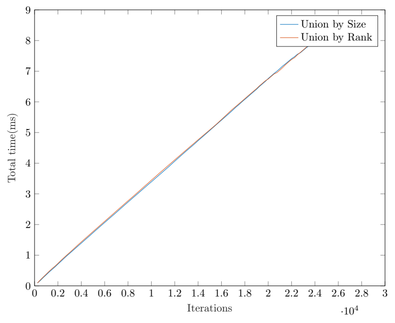
(b) Total Time with different union strategies with $n = 5000$



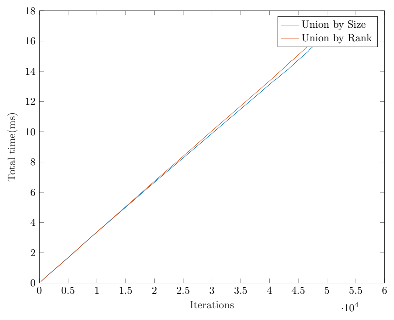
(c) Total Time with different union strategies with $n = 10000$



(d) Total Time with different union strategies with $n = 1000$ without Quick-Union



(e) Total Time with different union strategies with $n = 5000$ without Quick-Union

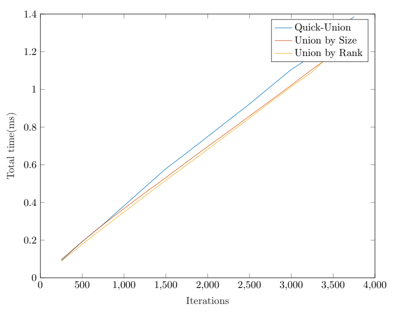


(f) Total Time with different union strategies with $n = 10000$ without Quick-Union

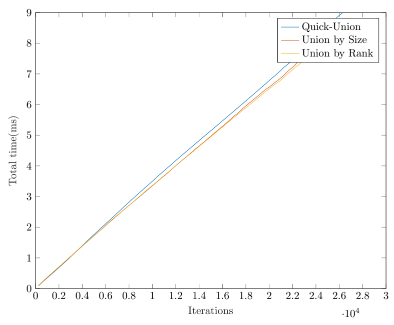
Figure 8: Total time No Compression

What is really interesting is that when we add path compression techniques, although Quick-Union performed the worst in both TPL and TPU metrics using any path heuristic, its execution time does not notably differ from weighted unions (see Figure 9), as TPL and TPU do. We might need a larger sample size to spot a notable difference in the trade-off of performing *cheap* unions in exchange for letting the find operation balance the trees. In fact there is not a noticable difference in execution time rather than calculating metrics.

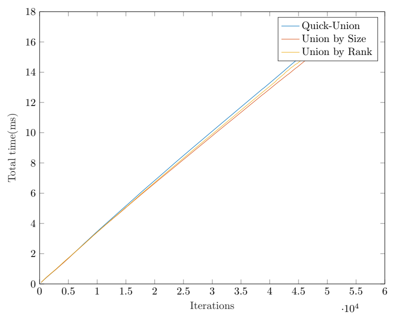
From a theoretical analysis[2], we know that Union-Find can have an amortized cost (using heuristics on both compression (full compression/splitting/halving) and unions) of $O(m\,\alpha(n, m))$, where α is the inverse Ackermann function, m is the number of operations we want to perform and n the number of disjoint blocks. Without compression (or using type one reversal), we can achieve $O(m \log n)$. We may need a larger input size in order to see a noticeable difference, only with those sizes we cannot see any time difference, although we really saw it calculating metrics.



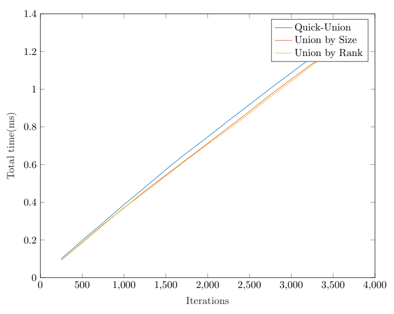
(a) Total Time with different union strategies with $n = 1000$ using Full Compression



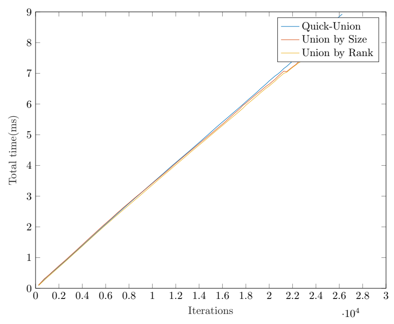
(b) Total Time with different union strategies with $n = 5000$ using Full Compression



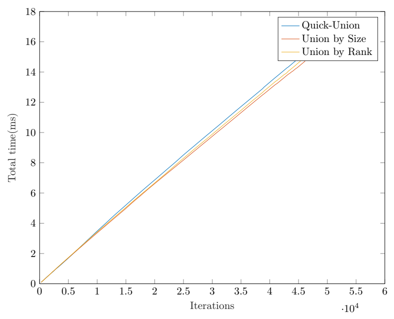
(c) Total Time with different union strategies with $n = 10000$ using Full Compression



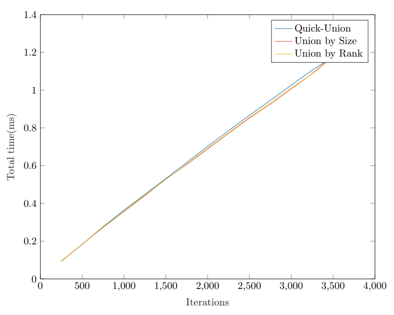
(d) Total Time with different union strategies with $n = 1000$ using Path Splitting



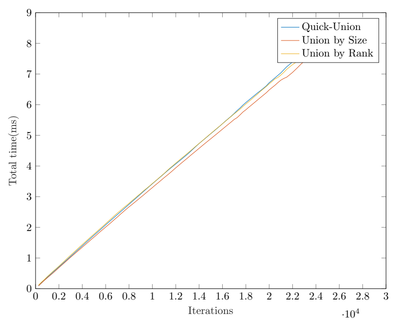
(e) Total Time with different union strategies with $n = 5000$ using Path Splitting



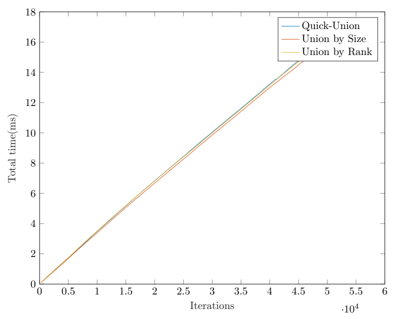
(f) Total Time with different union strategies with $n = 10000$ using Path Splitting



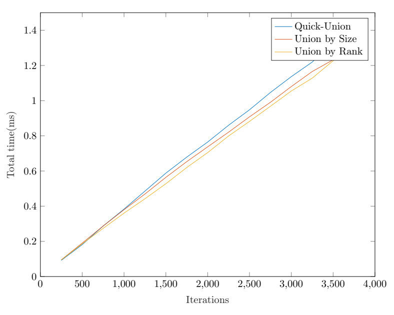
(g) Total Time with different union strategies with $n = 1000$ using Path Halving



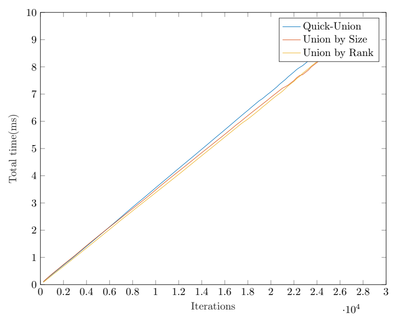
(h) Total Time with different union strategies with $n = 5000$ using Path Halving



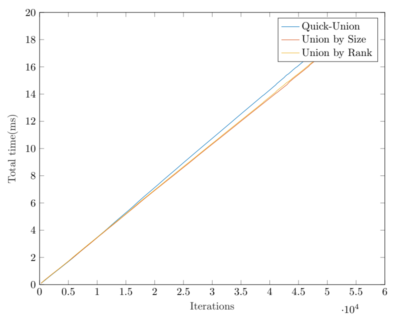
(i) Total Time with different union strategies with $n = 10000$ using Path Halving



(j) Total Time with different union strategies with $n = 1000$ using Type One Reversal



(k) Total Time with different union strategies with $n = 5000$ using Type One Reversal



(l) Total Time with different union strategies with $n = 10000$ using Type One Reversal

Figure 9: Total time using different heuristics

6 Conclusions

In this empirical analysis, we observed that Union-Find is a powerful data structure with efficient operations and a strong amortized cost. We also saw that time performance and metric gathering may not always be correlated (or perhaps they are, but we need a larger input size to confirm this).

Additionally, we studied a new path compression technique that is not as popular as Full Compression, Path Splitting, or Path Halving. At first glance, it appeared promising, but the metrics did not show any significant improvement.

Future work with larger input sizes is also promising in order to determine when the $\log n$ factor becomes noticeable compared to the inverse Ackermann function.

References

[1] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Communications of the ACM*, vol. 7, no. 5, pp. 301–303, 1964.

[2] R. E. Tarjan and J. Van Leeuwen, "Worst-case analysis of set union algorithms," *Journal of the ACM (JACM)*, vol. 31, no. 2, pp. 245–281, 1984.

A Union Operation: Code

From `main.cc` there is a single call to the Union operation. Such call is executed by the following merge function:

```
void UnionFind::merge(unsigned int i, unsigned int j) {
    unsigned int ri = find(i);
    unsigned int rj = find(j);
    if (ri == rj) return;
    --numBlocks;
    switch(strat) {
        case UnionStrategy::QU:
            mergeQU(ri, rj);
            break;
        case UnionStrategy::UW:
            mergeUW(ri, rj);
            break;
        case UnionStrategy::UR:
            mergeUR(ri, rj);
            break;
        default:
            break;
    }
}
```

Firstly one can see that there are two calls to the `find` operation which, depending on the strategy choosen for the path compression they will behave differently. For now let us just consider how the union operation is implemented

A.1 Quick-Union

Anyone who chooses to use this strategy as their union strategy will perform the union operation in an extremely efficient time—as there is only one single operation—at the cost of increased time complexity in the find operation (we do not keep track of how paths will change when we perform the union). The following code provides an implementation of this approach:

```
void UnionFind::mergeQU(unsigned int ri, unsigned int rj) {
    P[ri] = rj;
}
```

A.2 Union by Weight

An straightforward heuristic in order to choose how we can merge two different trees is to just merge the smaller one into the larger one. We will expect to not increase as much the path that we will create as if we do it the other way around. The following code provides an implementation of this approach:

```
void UnionFind::mergeUW(unsigned int ri, unsigned int rj) {
    //Recall that representatives here are negative numbers
    if (P[ri] >= P[rj]) {
        P[rj] += P[ri];
        P[ri] = rj;
    }

    else {
        P[ri] += P[rj];
        P[rj] = ri;
    }
}
```

A.3 Union by Rank

A similar idea from the Union by Weight heuristic can be applied here, but this time, we aim to control the rank of a tree, which we will use as an upper bound for its height. The following code provides an implementation of this approach:

```
void UnionFind::mergeUR(unsigned int ri, unsigned int rj) {

    //Recall that representatives here are negative numbers
    if (P[ri] >= P[rj]) {
        P[rj] = min(P[rj], P[ri] - 1);
        P[ri] = rj;
    }

    else {
        P[ri] = min(P[ri], P[rj] - 1);
        P[rj] = ri;
    }
}
```

B Find Operation: Code

Let us tackle the find operation, which is the first call during the union function. The following code correspond to such call:

```
unsigned int UnionFind::find(unsigned int i) {

    switch(path) {
        case PathStrategy::FC:
            return pathFC(i);
        case PathStrategy::PS:
            return pathPS(i);
        case PathStrategy::PH:
            return pathPH(i);
        case PathStrategy::TOR:
            return pathR(i);
        default:
            while (parent(i) != i) i = P[i];
            return i;
    }
}
```

As you can see, depending on the user's strategy for path compression, the code will choose the corresponding approach. It is worth noting that if the user decides not to use any path compression, the find operation will simply look for the representative of the class in a straightforward manner within the while loop (where `weighted` is just a boolean that indicates whether the representation of representatives uses negative numbers or not).

B.1 Full Compression

The following code provides an implementation of this approach:

```
unsigned int UnionFind::pathFC(unsigned int i) {

    //parent(i) is defined as P[i] < 0 ? i : P[i]
    if (parent(i) == parent(parent(i))) return parent(i);

    else {
        P[i] = pathFC(P[i]);
        #ifndef TIME
        ++tpu;
        #endif
        return P[i];
    }
}
```

B.2 Path Splitting

The following code provides an implementation of this approach:

```
unsigned int UnionFind::pathPS(unsigned int i) {

    //parent(i) is defined as P[i] < 0 ? i : P[i]
    while (parent(i) != parent(parent(i))) {
        unsigned int aux = P[i];
        P[i] = P[P[i]];
        i = aux;
        //Increase by one the counter which tracks
        //the number of pointer switches that one makes
        #ifndef TIME
        ++tpu;
        #endif
    }

    //This function will stop either at the root or
    //at a children of the node, that is why we
    //must make another call to parent(i)
    return parent(i);
}
```

B.3 Path Halving

The following code provides an implementation of this approach:

```
unsigned int UnionFind::pathPH(unsigned int i) {

    while (parent(i) != parent(parent(i))) {
        P[i] = P[P[i]];
        i = P[i];
        //Increase by one the counter which tracks
        //the number of pointer switches that one makes
        #ifndef TIME
        ++tpu;
        #endif
    }

    //This function will stop either at the root or
    //at a children of the node, that is why we
    //must make another call to parent(i)
    return parent(i);
}
```

B.4 Type One Reversal

The following code provides an implementation of this approach:

```
unsigned int UnionFind::pathR(unsigned int i) {

    unsigned int finalNode = i;
    i = parent(i);

    while (parent(i) != parent(parent(i))) {
        unsigned int aux = P[i];
        P[i] = finalNode;
        i = aux;
#ifndef TIME
        ++tpu;
#endif
    }

    if (parent(finalNode) != parent(i)) {
        P[finalNode] = parent(i);
        P[i] = finalNode;
#ifndef TIME
        tpu += 2;
#endif
    }
    return parent(finalNode);
}
```

C TPL Calculation

As stated before, we have decided to traverse the entire data structure to calculate the TPL. Again, it is important to remember that the goal of this metric is its value rather than the efficiency of its computation. The following code provides an implementation of this approach:

```
unsigned int UnionFind::getTPL() const {

    unsigned int tpl = 0;

    for (unsigned int i = 0; i < size; ++i) {
        unsigned int j = i;

        while (parent(j) != j) {
            j = P[j];
            ++tpl;
        }

    }

    return tpl;
}
```