UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

ADVANCED DATA STRUCTURES

COMPUTER SCIENCE DEPARTMENT

Union-Find data structure: An empirical analysis

Author:
Alex Herrero

Professor:
Conrado Martínez

February 25, 2025

Contents

List of Figures

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, given a partition Π the result of the union operation will be 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 (check `UnionFind.hh` for a more detailed view of the different methods implemented and `UnionFind.cc` for such implementation). 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 for 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). 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 (i.e. *total path length*) and TPU (i.e. *total pointer updates*) of the data structure. TPU 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, I have decided to use this method for computing TPL because the focus is on obtaining the value rather than optimizing execution time. See README file for knowing how to execute this program.

3 Experimental Setup

To generate random instances, a number n of elements and a seed s are requested as input of `generator.cc`. 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 determine, as there is no guarantee that the program will terminate. Since it generates pseudo-random pairs, it might enter a loop where all the generated pairs are repetitions. However, this approach is better than my initial implementation, which involved generating all possible $\binom{n}{2}$ pairs—requiring $\Theta(n^2)$ time and space complexity—then randomly shuffling the pairs and introducing them to the Union-Find until a single block was formed.

With this new approach, we use a C++ `set` to keep track of all unique pairs, requesting only the 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[GF64]!) 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. For every instance with the same size, union strategy, and path strategy, we will average all values obtained for every $k \in \{\Delta, 2\Delta, 3\Delta, \ldots\}$ pairs processed, as long as 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).

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 makes sense, as we do not use any heuristic to help minimize those paths.

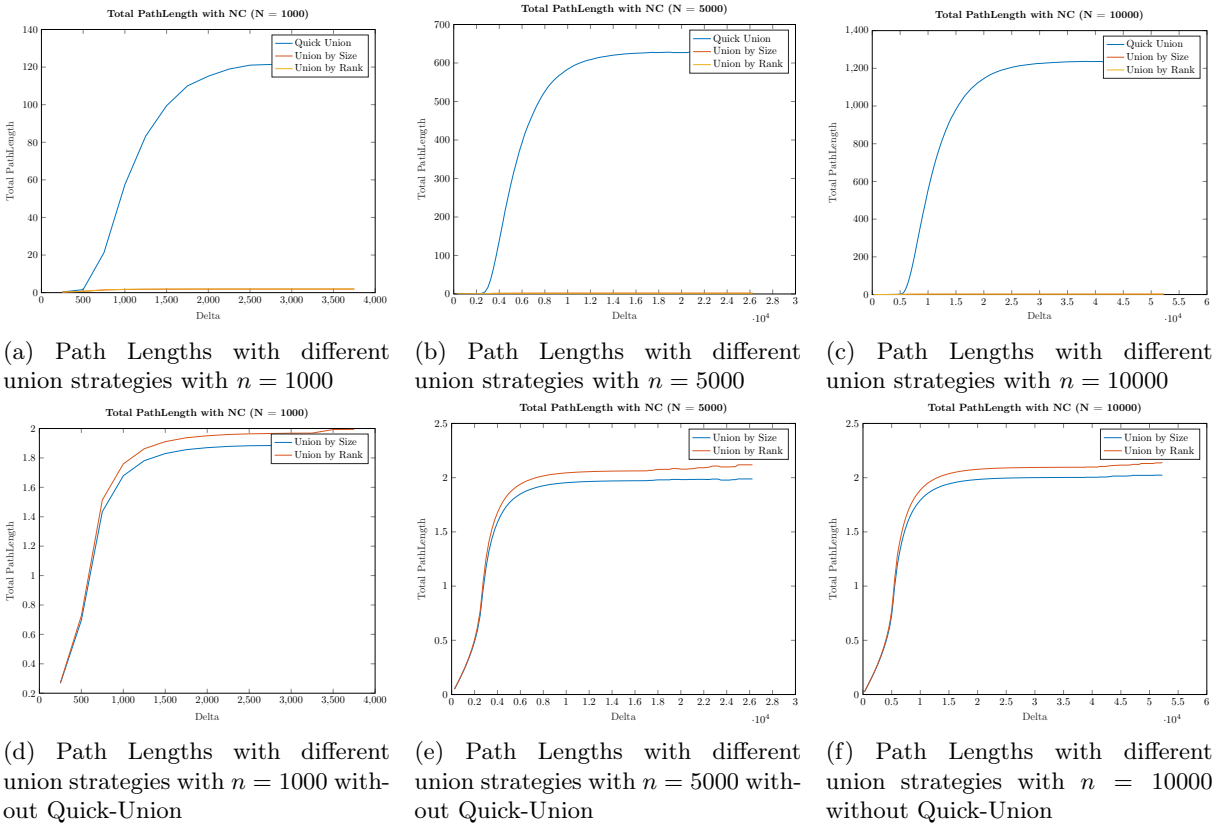
Still we need to take a look on the performance of Union by size and Union by Weight, Figure 1 also provides a look to such performance. As we see, there is not a big difference (although union by size performs slightly better) between these strategies in this context.

Both heuristics exhibit a logarithmic behavior, which is expected. When performing a union without compression, the length added is proportional to the expected length, $O(\log n)$, due to randomness. This explains why the curve starts flat and then increases quickly. However, as more unions are performed, the probability that the pair is already connected increases, so the union operation does nothing and the TPL does not change.

4.1.2 Path Strategy: using Heuristics

Here we are going to consider the following heuristics for path compression:

- **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.



(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

- 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 points to i and i points to r_i. Although this might be strange this heuristic has a similar idea to Full Compression (but we traverse a path only once insted of twice!). Although in this analysis we will only consider **Type One Reversal**, in [TVL84] they define a type k reversal for $k \geq 1$.

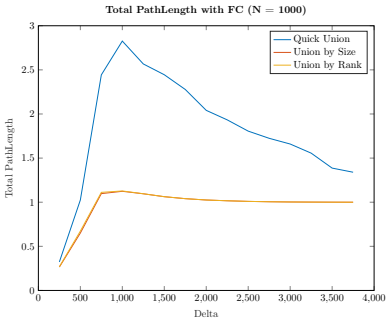
As expected, Figure 2 clearly shows that Quick-Union begins increasing its TPL faster than the other methods. There is a considerable initial increment, but it quickly starts decreasing (except in the case of type-one reversal, which reaches stability in a logarithmic number of steps—at twice the maximum peak of the other heuristics—since the minimum allowed path length is 2). This aligns with expectations: the more find operations we perform, the more balanced each tree becomes.

However, we do not observe a *smooth* decrease in the shape, unlike with other union heuristics (see Figure 3). This suggests that, although TPL improves as more find operations are executed, the choice of heuristic for the merge strategy also impacts performance. Indeed, there is a drastic improvement in TPL once it starts decreasing, as Quick-Union initially leaves more room for optimization.

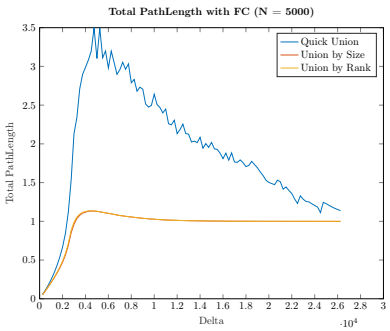
Both heuristics, Union by Weight and Union by Rank, exhibit the same behavior. Since they aim to maintain balanced trees, there is no noticeable improvement after a certain point, and stability is reached *quickly*.

A similar argument can be used to analyze TPUs in Figures 4 and 5: because the average TPL in Quick-Union is greater than in Weighted Unions, we traverse more nodes from one element to its representative, resulting in more changes within the path.

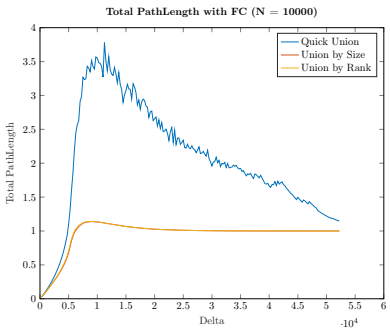
A particularly interesting result in Figures 3 and 5 is that every path compression heuristic (except for Type One Reversal) with Weighted Unions behaves in exactly the same way. This is an experimental view of the following claim: every combination of m Union and Find operations on n disjoint sets, using Union by Size/Rank and any of the path compression heuristics (full, splitting, or halving), has a cost of $O(m \cdot \alpha(m, n))$, where α is the inverse Ackermann function[TVL84]. This function is well known for growing extremely slowly—slower even than the iterative logarithm. Type One Reversal performs in $O(m \log n)$.



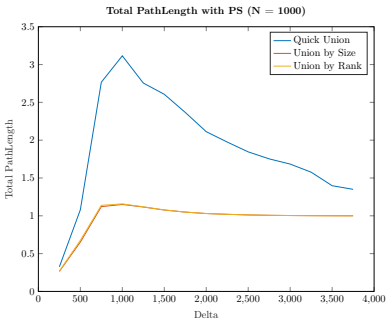
(a) Path Lengths with different union strategies with $n = 1000$ using Full Compression



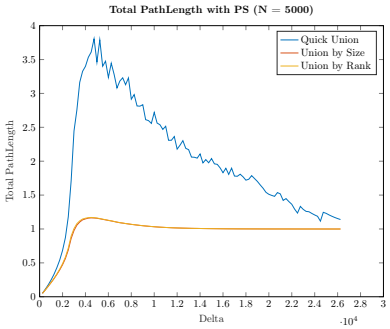
(b) Path Lengths with different union strategies with $n = 5000$ using Full Compression



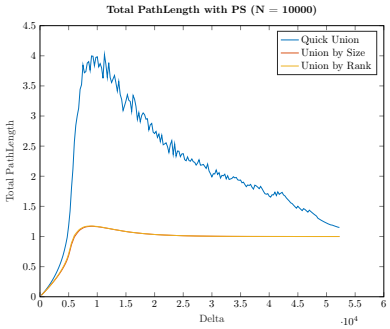
(c) Path Lengths with different union strategies with $n = 10000$ using Full Compression



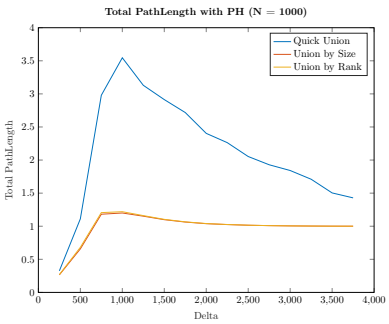
(d) Path Lengths with different union strategies with $n = 1000$ using Path Splitting



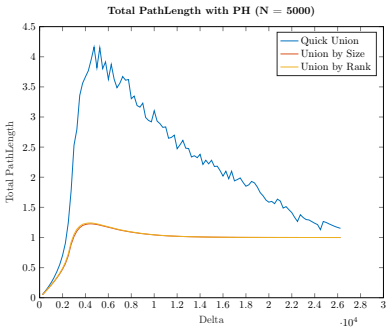
(e) Path Lengths with different union strategies with $n = 5000$ using Path Splitting



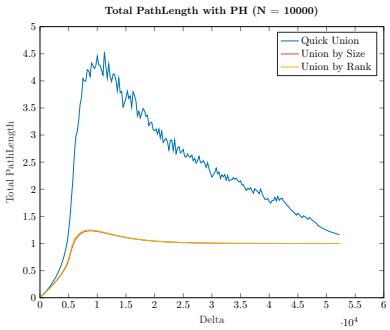
(f) Path Lengths with different union strategies with $n = 10000$ using Path Splitting



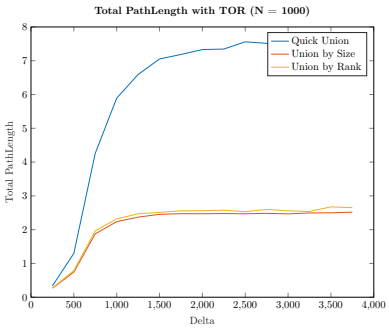
(g) Path Lengths with different union strategies with $n = 1000$ using Path Halving



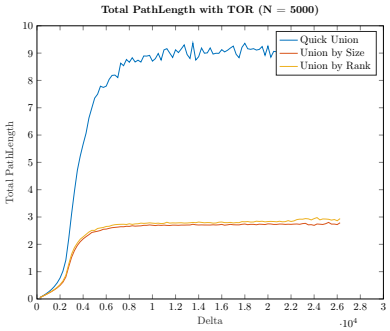
(h) Path Lengths with different union strategies with $n = 5000$ using Path Halving



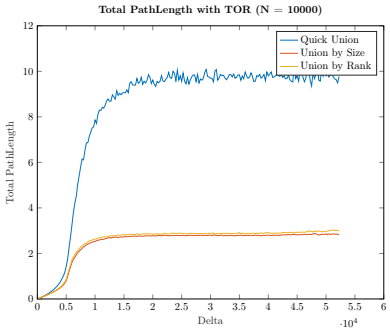
(i) Path Lengths with different union strategies with $n = 10000$ using Path Halving



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

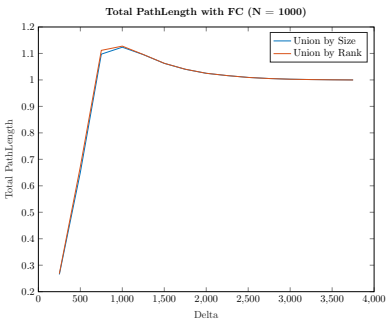


(k) Path Lengths with different union strategies with $n = 5000$ using Type One Reversal

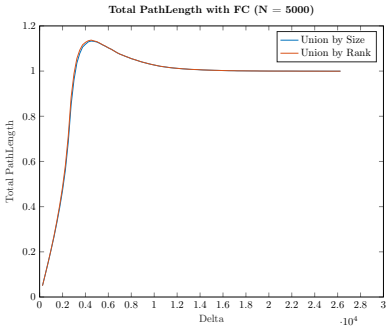


(l) Path Lengths with different union strategies with $n = 10000$ using Type One Reversal

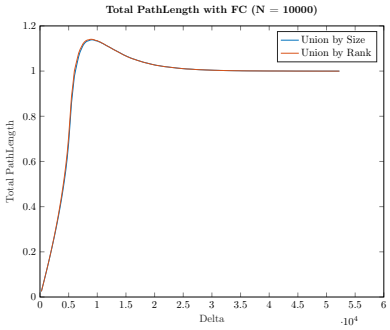
Figure 2: Total Path Length normalized for different heuristics



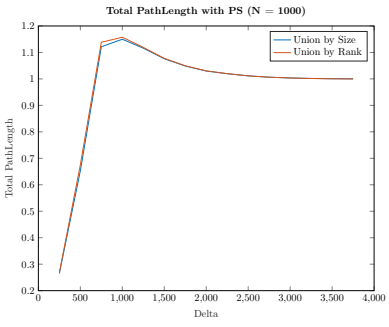
(a) Path Lengths with different union strategies with $n = 1000$ using Full Compression



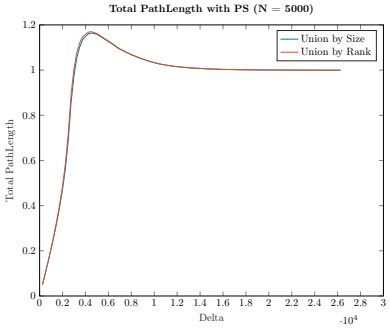
(b) Path Lengths with different union strategies with $n = 5000$ using Full Compression



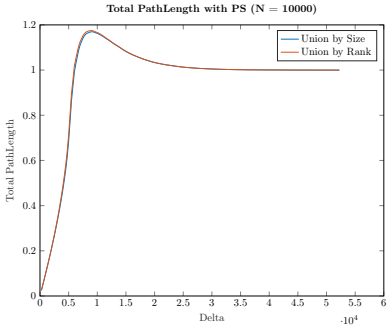
(c) Path Lengths with different union strategies with $n = 10000$ using Full Compression



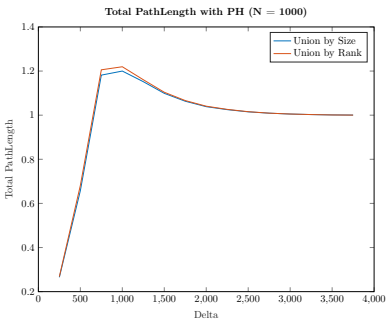
(d) Path Lengths with different union strategies with $n = 1000$ using Path Splitting



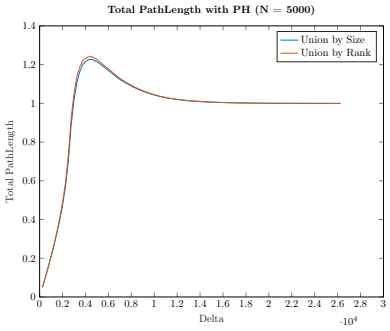
(e) Path Lengths with different union strategies with $n = 5000$ using Path Splitting



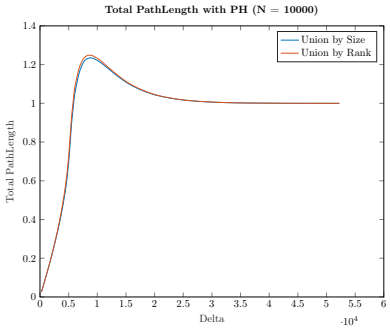
(f) Path Lengths with different union strategies with $n = 10000$ using Path Splitting



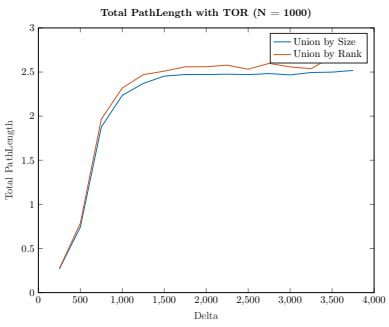
(g) Path Lengths with different union strategies with $n = 1000$ using Path Halving



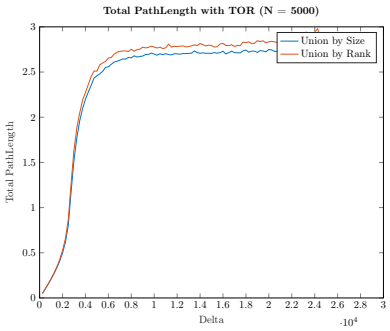
(h) Path Lengths with different union strategies with $n = 5000$ using Path Halving



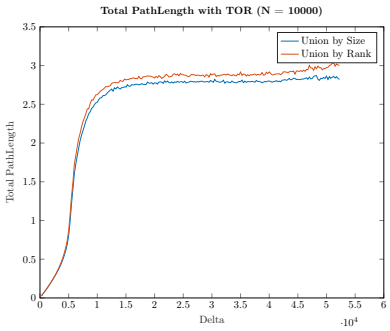
(i) Path Lengths with different union strategies with $n = 10000$ using Path Halving



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

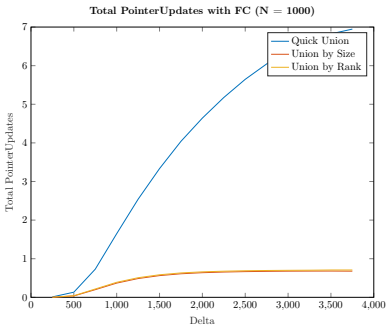


(k) Path Lengths with different union strategies with $n = 5000$ using Type One Reversal

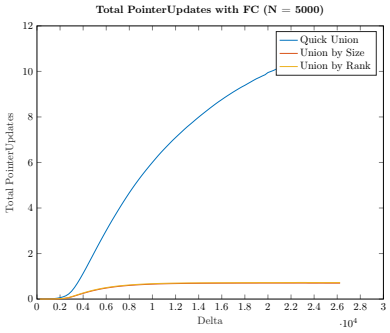


(l) Path Lengths with different union strategies with $n = 10000$ using Type One Reversal

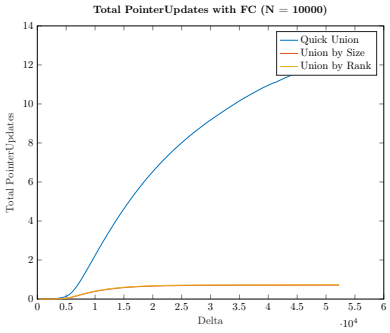
Figure 3: Total Path Length normalized for different heuristics without Quick-Union



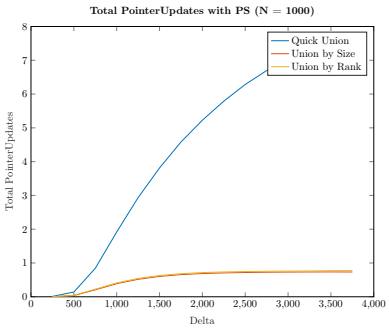
(a) Pointer Updates with different union strategies with $n = 1000$ using Full Compression



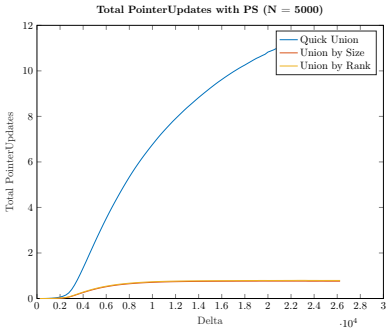
(b) Pointer Updates with different union strategies with $n = 5000$ using Full Compression



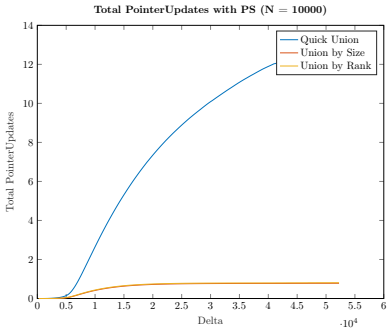
(c) Pointer Updates with different union strategies with $n = 10000$ using Full Compression



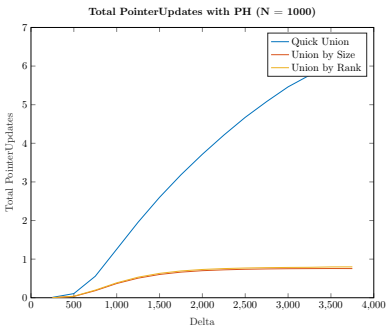
(d) Pointer Updates with different union strategies with $n = 1000$ using Path Splitting



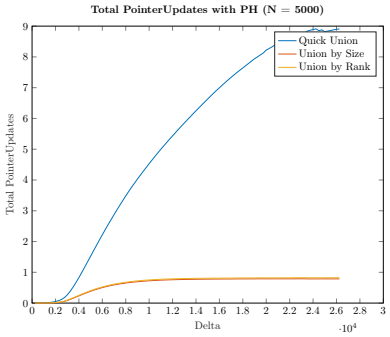
(e) Pointer Updates with different union strategies with $n = 5000$ using Path Splitting



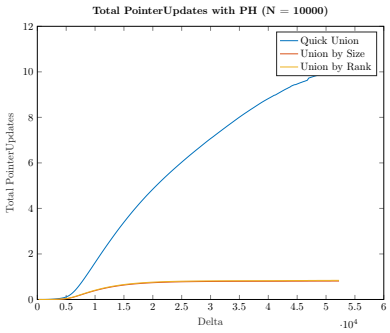
(f) Pointer Updates with different union strategies with $n = 10000$ using Path Splitting



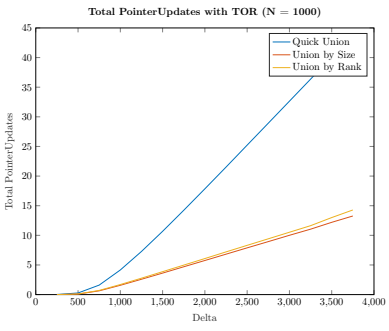
(g) Pointer Updates with different union strategies with $n = 1000$ using Path Halving



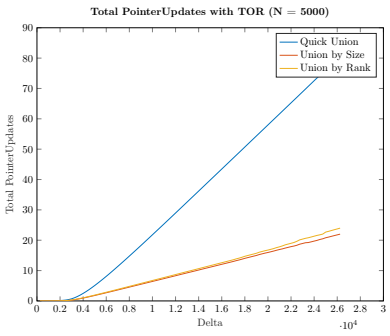
(h) Pointer Updates with different union strategies with $n = 5000$ using Path Halving



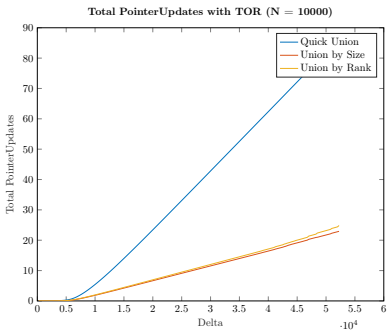
(i) Pointer Updates with different union strategies with $n = 10000$ using Path Halving



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

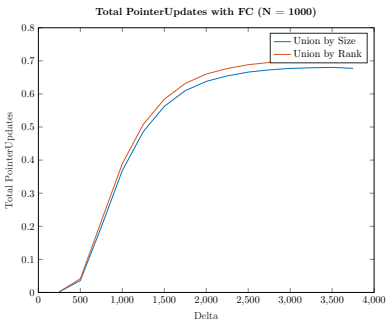


(k) Pointer Updates with different union strategies with $n = 5000$ using Type One Reversal

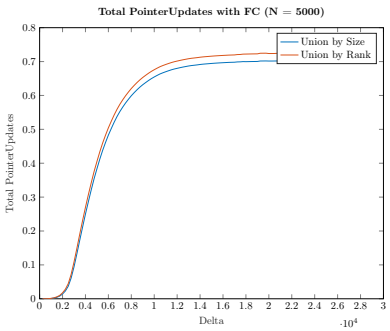


(l) Pointer Updates with different union strategies with $n = 10000$ using Type One Reversal

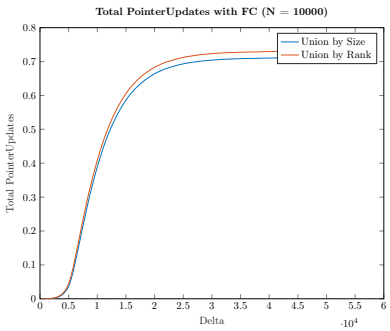
Figure 4: Total Pointer Update normalized using different heuristics



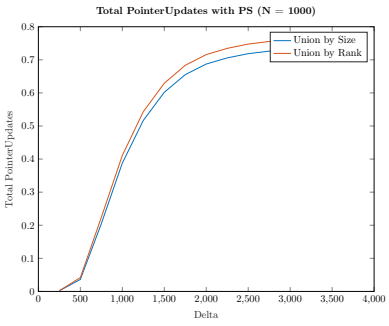
(a) Pointer Updates with different union strategies with $n = 1000$ using Full Compression



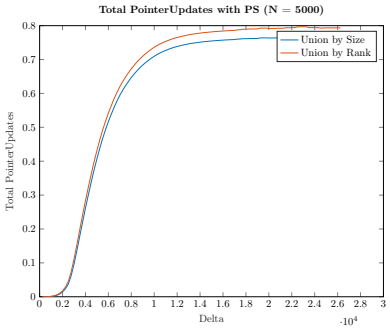
(b) Pointer Updates with different union strategies with $n = 5000$ using Full Compression



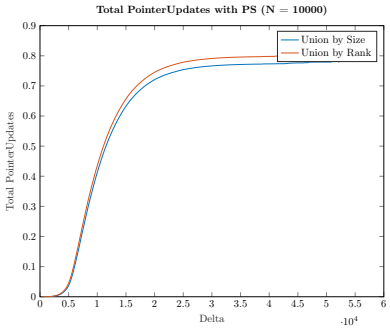
(c) Pointer Updates with different union strategies with $n = 10000$ using Full Compression



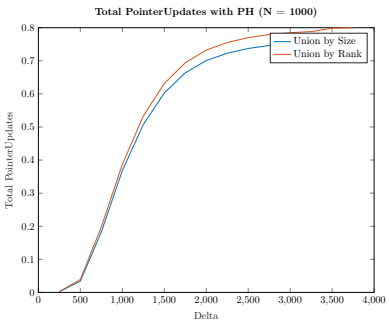
(d) Pointer Updates with different union strategies with $n = 1000$ using Path Splitting



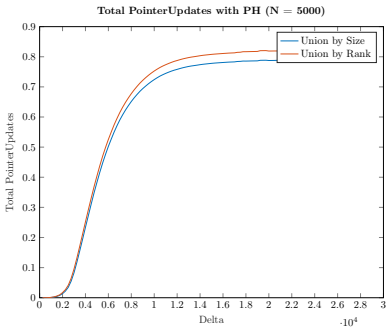
(e) Pointer Updates with different union strategies with $n = 5000$ using Path Splitting



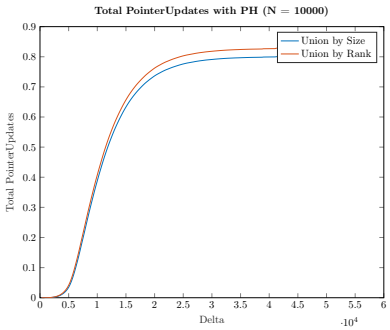
(f) Pointer Updates with different union strategies with $n = 10000$ using Path Splitting



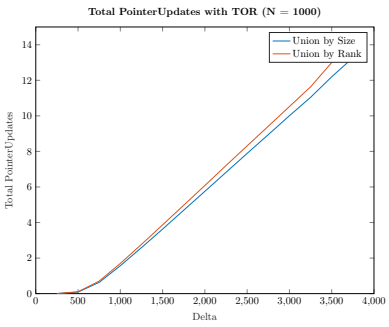
(g) Pointer Updates with different union strategies with $n = 1000$ using Path Halving



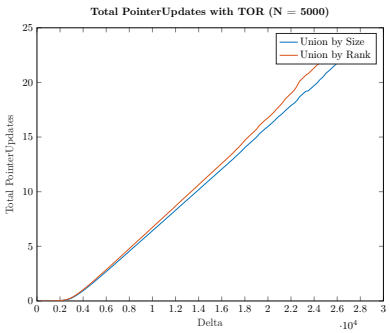
(h) Pointer Updates with different union strategies with $n = 5000$ using Path Halving



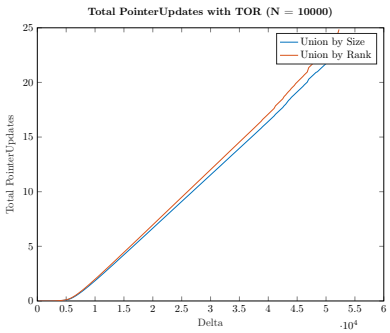
(i) Pointer Updates with different union strategies with $n = 10000$ using Path Halving



(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

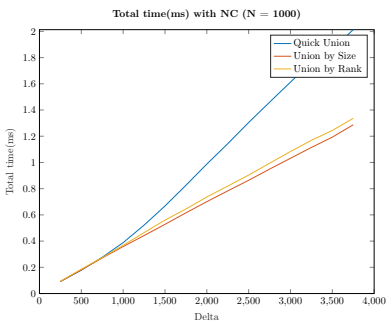


(l) Pointer Updates with different union strategies with $n = 10000$ using Type One Reversal

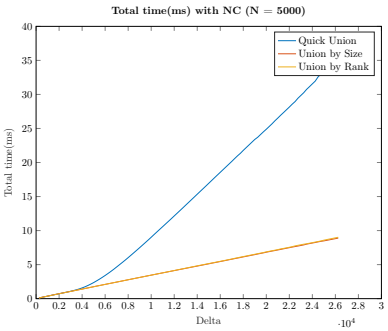
Figure 5: Total Pointer Update normalized using different heuristics without Quick-Union

5 Time Comparison

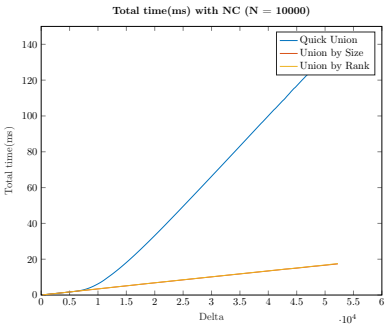
With the same approach explained in Section 3, time has been calculated. Figure 6 provides plots for the execution time counting only the time needed for doing merge operations. As expected, Quick-Union performs the worst when we compare with weighted unions (the difference is even more clear as we increase 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).



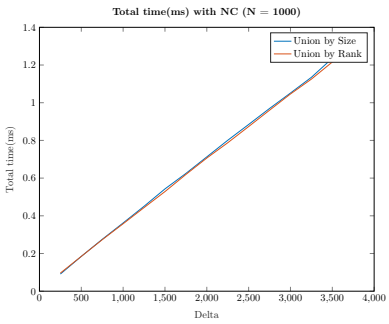
(a) Total time with different union strategies with $n = 1000$



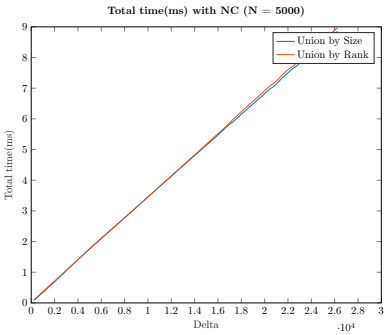
(b) Total Time with different union strategies with $n = 5000$



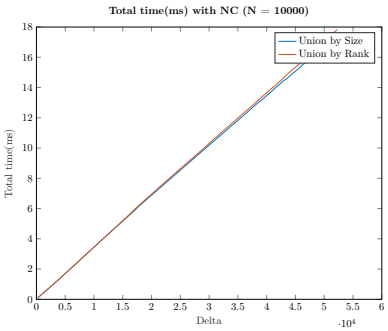
(c) Total Time with different union strategies with $n = 10000$



(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 6: Total time No Compression

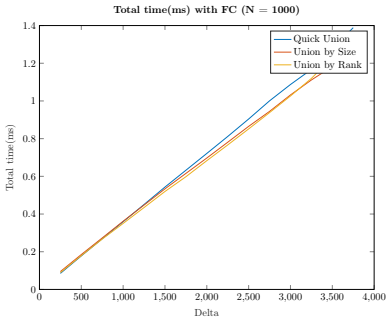
What is really interesting is that, although Quick-Union performed the worst in both TPL and TPU metrics using any path heuristic, its execution time does not notably differ from weighted unions (see Figure 7), as TPL and TPU do. We might need a larger sample size to spot a notable difference in the trade-off of performing *cheap* unions in exchange for letting the find operation balance the trees. There is a similar behavior with Type One Reversal, which performed the worst on both TPL and TPU metrics of path heuristics.

6 Conclusions

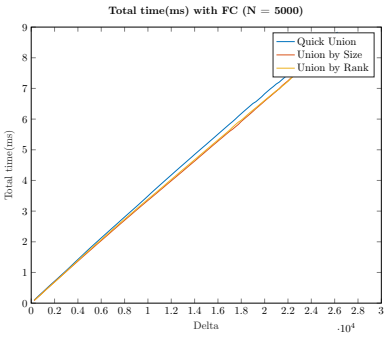
We still need larger sizes to accurately identify a time difference when using any heuristics, as, for our current size, their execution times are similar. This might lead to confusion between these empirical results and the theoretical ones, from which we know that some combinations perform better asymptotically.

Nonetheless, by examining TPL and TPU, we observe that these metrics performed the worst for combinations of heuristics that were non-optimal, which is entirely expected.

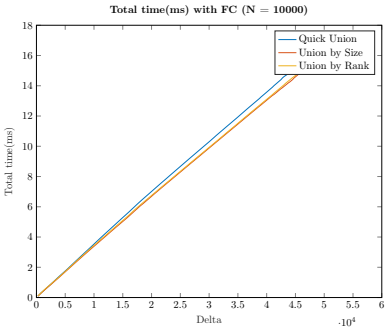
A more detailed experiment with a larger sample size (which must be significantly large, as asymptotic functions grow slowly) would be valuable for further understanding this data structure.



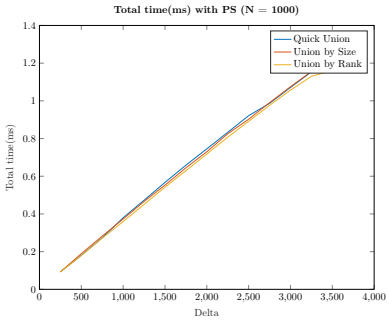
(a) Total Time with different union strategies with $n = 1000$ using Full Compression



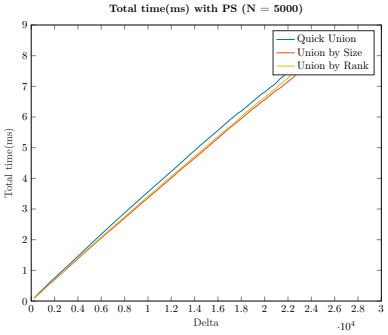
(b) Total Time with different union strategies with $n = 5000$ using Full Compression



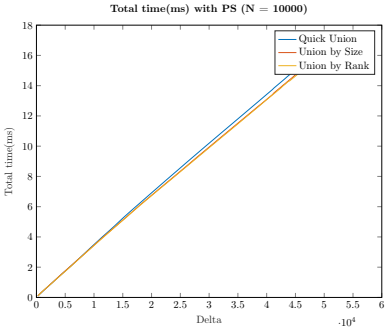
(c) Total Time with different union strategies with $n = 10000$ using Full Compression



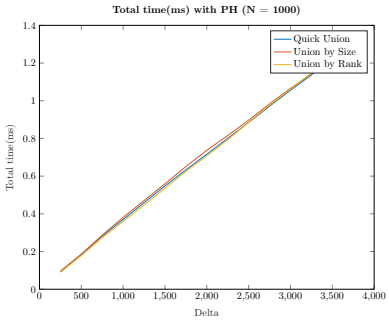
(d) Total Time with different union strategies with $n = 1000$ using Path Splitting



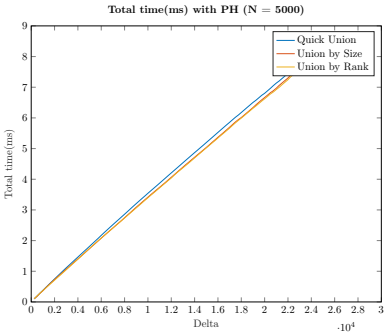
(e) Total Time with different union strategies with $n = 5000$ using Path Splitting



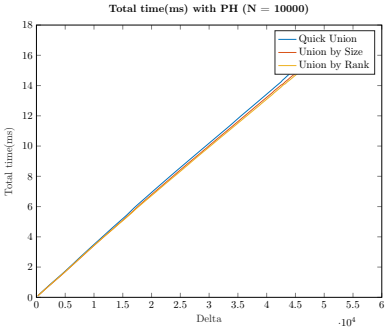
(f) Total Time with different union strategies with $n = 10000$ using Path Splitting



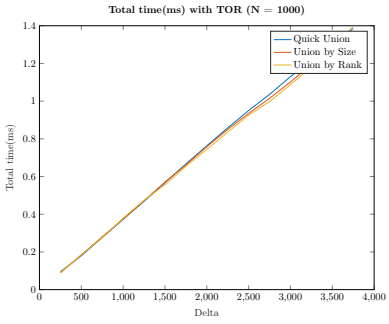
(g) Total Time with different union strategies with $n = 1000$ using Path Halving



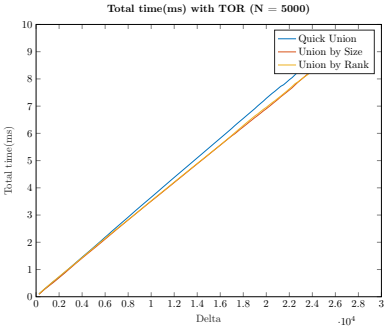
(h) Total Time with different union strategies with $n = 5000$ using Path Halving



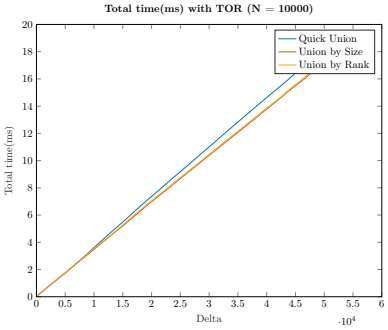
(i) Total Time with different union strategies with $n = 10000$ using Path Halving



(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 7: Total time using different heuristics

References

[GF64] Bernard A Galler and Michael J Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[TVL84] Robert E Tarjan and Jan Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):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. 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 (weighted ? P[i] > 0 : P[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 Path 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, the author has 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 (weighted ? P[j] > 0 : P[j] != j) {
            j = P[j];
            ++tpl;
        }

    }

    return tpl;
}
```