Advanced Data Structures

Computer Science Department

Union-Find data structure: An empirical analysis

Author:
Alex Herrero

Professor:
Conrado Martínez

February 22, 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, 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. 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 3, representing No Compression, Full Compression, Path Splitting, and Path Halving). 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
```

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!) to 1983. Each instance has been executed with the 12 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).

Using these samples we gained samples for, approximately: 3750 pairs of elements for $n = 1000$, 26250 pairs of elements for $n = 5000$ and 52250 pairs of elements for $n = 10000$.

4 Results

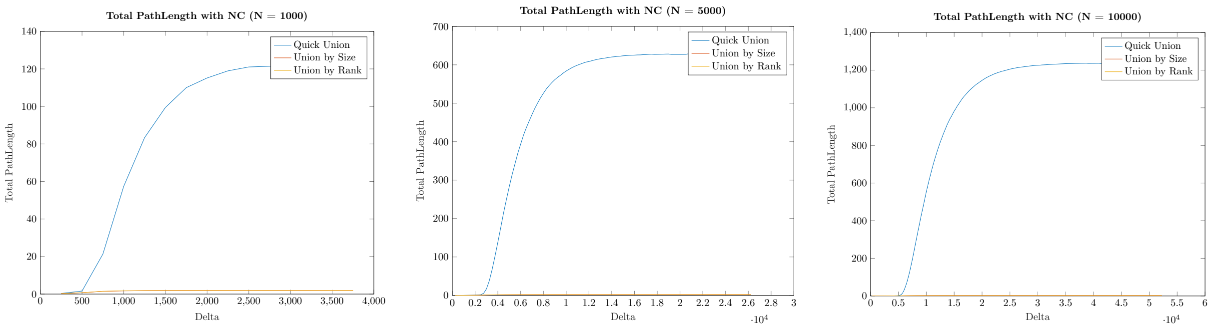
4.1 Metric Comparisons

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 2 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.

Taking a close look to Figures 1 and 2, when you perform a union without compression, the length you add is proportional to the expected length, $O(\log n)$ because of randomness, which explains why it starts flat and increases quickly. However, as you continue performing more unions, the probability that the pair is already connected increases, so the union does nothing and the TPL does not change. That is why the shape of both figures seems logarithmic.

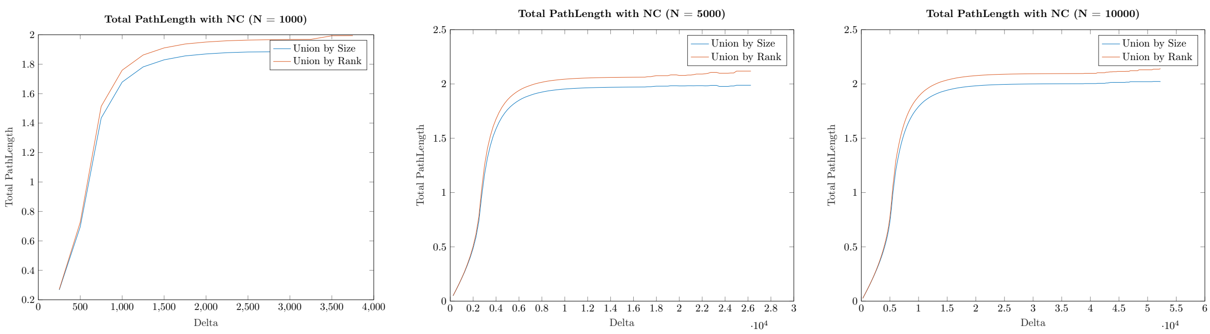


(a) Path Lengths with different union strategies with $n = 1000$

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

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

Figure 1: Total Path Length normalized with No Compression



(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 $n = 10000$

Figure 2: Total Path Length normalized with No Compression without Quick-Union

4.1.2 Path Strategy: Full Compression

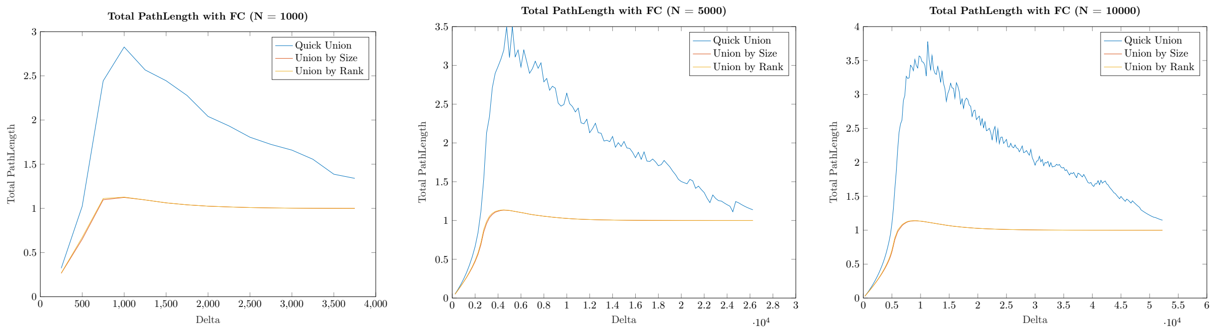
The Full Compression heuristic ensures that every element in the path from i to the representative of the class r_i will point directly to r_i by traversing the path twice (first to identify r_i and second to rearrange the pointers). At the cost of traversing this path twice, we expect to speed up future find operations involving a class where this heuristic has been applied.

As expected, Figure 3 clearly shows that Quick-Union starts increasing its TPL faster than the other methods. There is a considerable initial increment, but it quickly starts decreasing, which 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, rather than other heuristics in Figure 4. This suggests that, although there is an improvement in TPL as more find operations are executed, the choice of heuristic for the merge strategy also impacts performance (indeed, there is a drastically improvement on the TPL when it starts decreasing because with a Quick-Union strategy there is much more improvement left to do). Both heuristics Union by Weight and Rank behave in the same way (as we try to not unbalance the trees there is not a noticeable improvement).

A similar argument can be used to argument TPUs in Figures 5 and 6: Because average TPL in Quick-Union is greater than with weighted unions we will traverse more nodes from one element to its representative, resulting in more changes within the path.

4.1.3 Path Strategy: Path Splitting and Path Halving

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

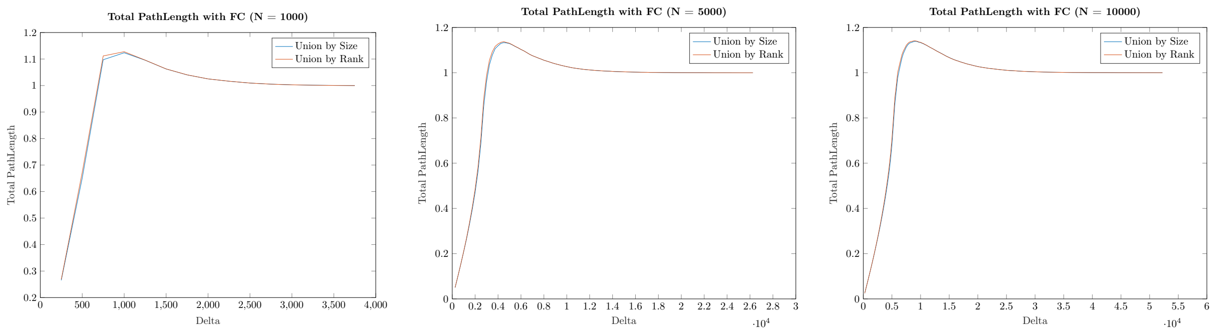


(a) Path Lengths with different union strategies with $n = 1000$

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

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

Figure 3: Total Path Length normalized with Full Compression



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

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

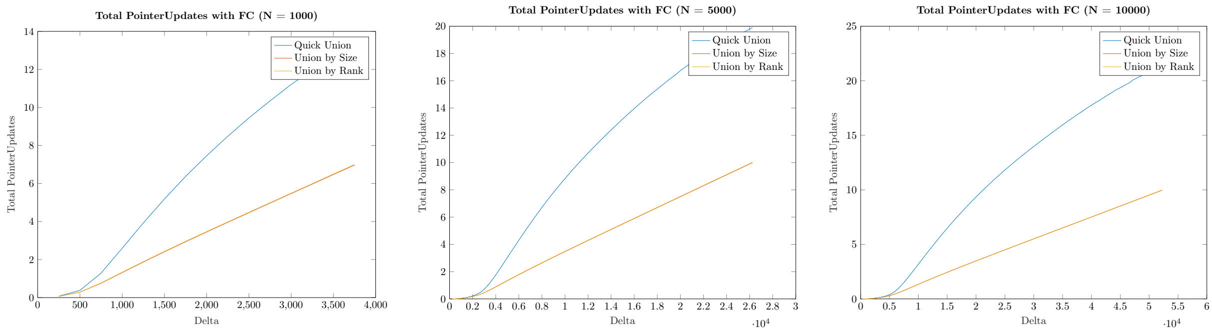
(c) Path Lengths with different union strategies with $n = 10000$

Figure 4: Total Path Length normalized with Full Compression without Quick-Union

grandparent as long as this exists. 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).

4.1.4 Path Strategy: Path Halving

An expected faster version of Path Splitting can be implemented by traversing every other node in the path, expecting that we will end sooner.

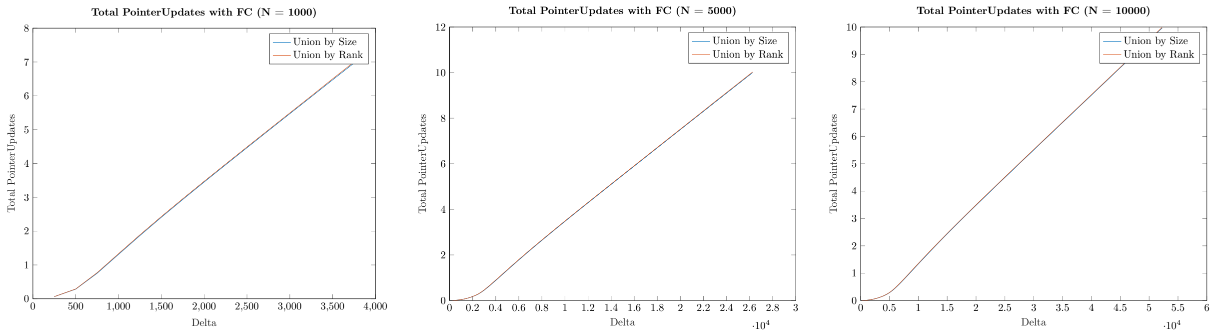


(a) Pointer Updates with different union strategies with $n = 1000$

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

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

Figure 5: Total Pointer Update normalized with Full Compression

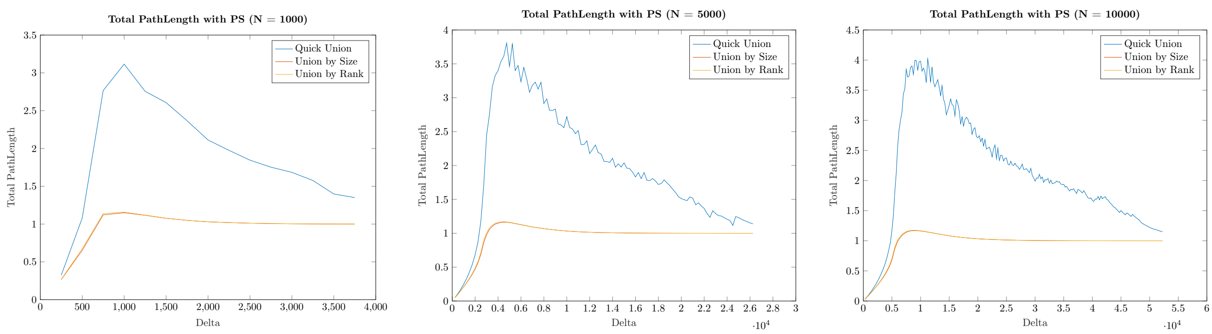


(a) Pointer Updates with different union strategies with $n = 1000$

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

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

Figure 6: Total Pointer Update normalized with Full Compression without Quick-Union

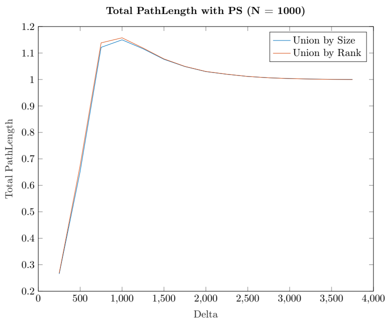


(a) Path Lengths with different union strategies with $n = 1000$

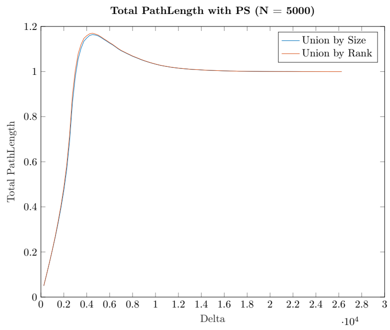
(b) Path Lengths with different union strategies with $n = 5000$

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

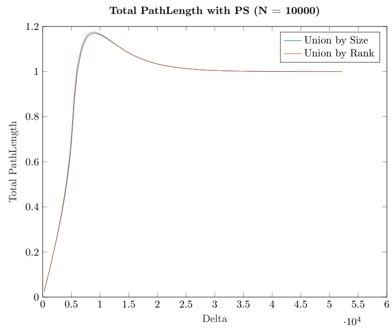
Figure 7: Total Path Length normalized with Path Splitting



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

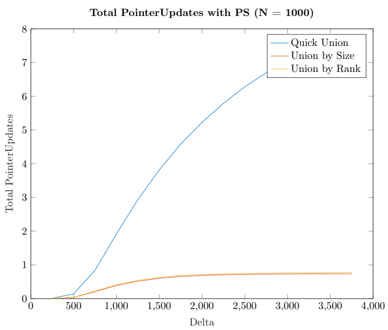


(b) Path Lengths with different union strategies with $n = 5000$

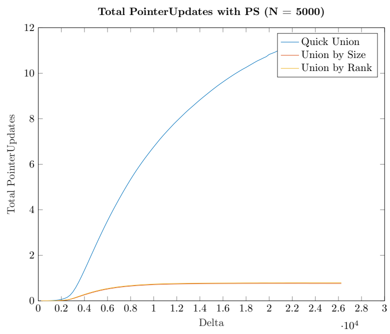


(c) Path Lengths with different union strategies with $n = 10000$

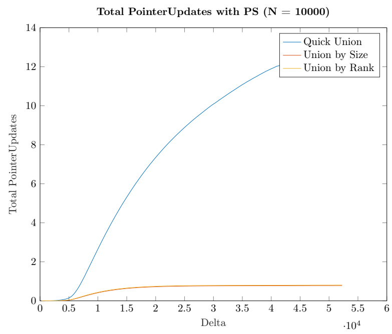
Figure 8: Total Path Length normalized with Path Splitting without Quick-Union



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

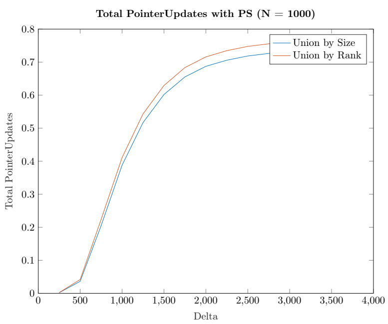


(b) Pointer Updates with different union strategies with $n = 5000$

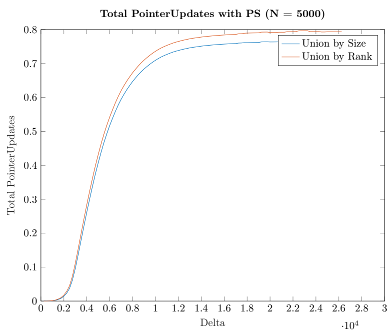


(c) Pointer Updates with different union strategies with $n = 10000$

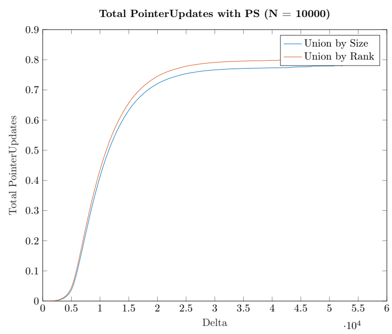
Figure 9: Total Pointer Update normalized with Path Splitting



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

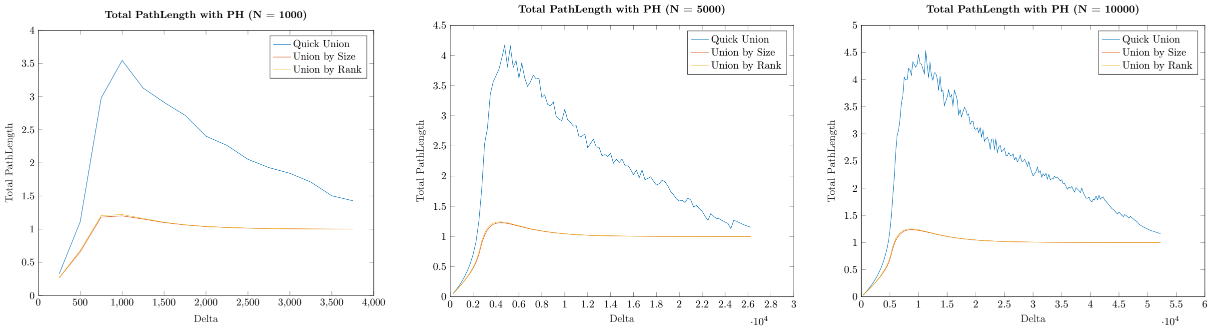


(b) Pointer Updates with different union strategies with $n = 5000$



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

Figure 10: Total Pointer Update normalized with Path Splitting without Quick-Union

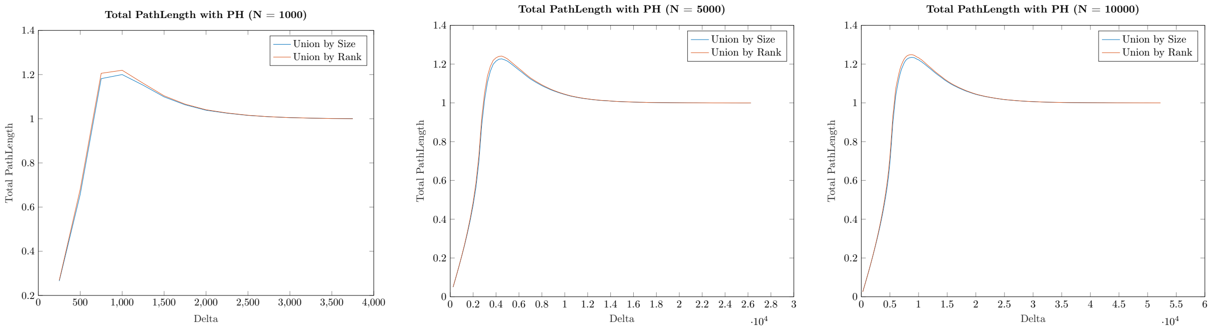


(a) Path Lengths with different union strategies with $n = 1000$

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

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

Figure 11: Total Path Length normalized with Path Halving

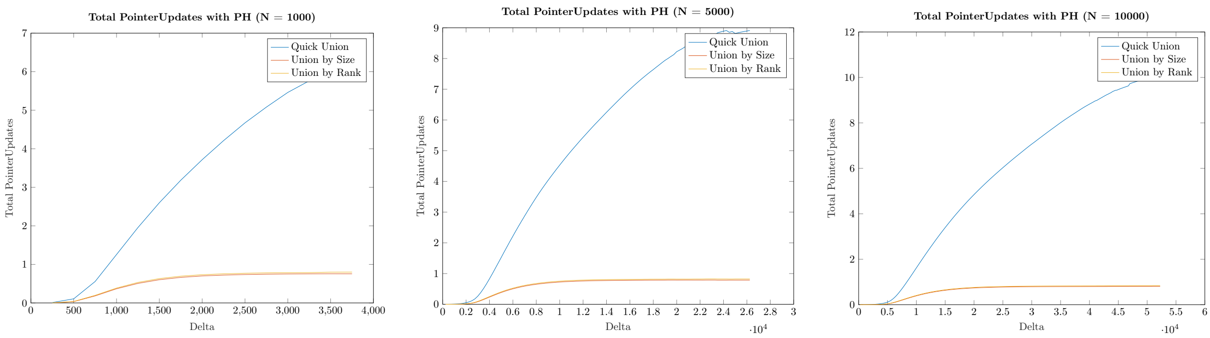


(a) Path Lengths with different union strategies with $n = 1000$

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

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

Figure 12: Total Path Length normalized with Path Halving without Quick-Union

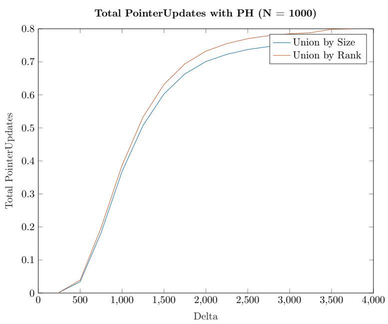


(a) Pointer Updates with different union strategies with $n = 1000$

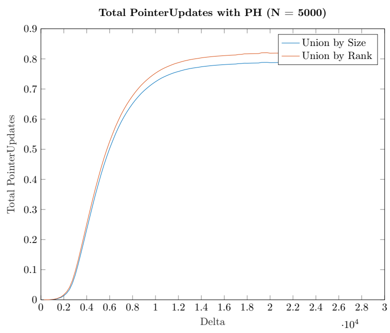
(b) Pointer Updates with different union strategies with $n = 5000$

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

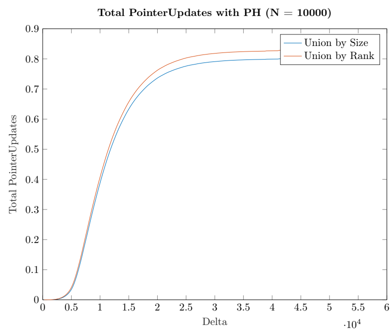
Figure 13: Total Pointer Update normalized with Path Halving



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



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



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

Figure 14: Total Pointer Update normalized with Path Halving without Quick-Union

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);
        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) {

    if (weighted ? P[i] < 0 : P[i] == i) return i;

    else {
        P[i] = pathFC(P[i]);
        //Increase by one the counter which tracks
        //the number of pointer switches that one makes
        ++tpu;
        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
        ++tpu;
    }

    //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
        ++tpu;
    }
```

```
    //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);
}
```

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;
}
```