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

February 21, 2025

Contents

1 Introduction **3**

2 Implementation **3**

A Union Operation: Code **5**
A.1 Quick-Union . . . 5
A.2 Union by Weight . . . 5
A.3 Union by Rank . . . 6

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:

- $\bigcup\limits_{i=1}^{k} A_i = \mathcal{A}$.
- $A_i \cap A_j = \emptyset$ 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. 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, while the TPL is calculated by traversing the entire data structure. 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 example, 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
```

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—more precisely, in $\Theta(1)$—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;
    }
}
```