Advanced Data Structures

Computer Science Department

Union Find Data Structure: An empirical analysis

Author:
Alex Herrero

Professor:
Conrado Marínez

February 18, 2025

Contents

List of Figures

List of Tables

1 Introduction

Given a binary relation R such that is:

- Reflexive: aRa
- Symmetric: $aRb \Rightarrow bRa$
- Transitive: $aRb \wedge bRc \Rightarrow aRc$

we say that R provides a partition Π of A into disjoint equivalence classes. That is, $\Pi = \{A_1, \dots, A_k\}$ is defined as follows:

- $\forall A_i, A_j \in \Pi, A_i \cap A_j = \emptyset \iff A_i \neq A_j$
- $A = \bigcup\limits_{i=1}^{k} A_i$
- $a \equiv b \iff a, b \in A_i$ for some $A_i \in \Pi$.

Such idea was used in Computer Science by Galler and Fisher[GF64] as the **union-find data structure** which is a data structure that stores partition of a set into disjoint sets. In particular **union find** consist of two main operation:

- Find Operation: Given two elements $a, b \in A$ determine if $\exists A_i \in \Pi$ s.t $a, b \in A_i$.
- Union Operation: Given two elements $a \in A_i$ and $b \in A_j$, *merge* A_i and A_j. That is, the result of this operation to the partition Π will be a new partition Π' such that $\Pi' = (\Pi \setminus \{A_i, A_j\}) \cup (A_i \cup A_j)$.

2 Implementation

From now on we are going to assume that our set A is defined as $\{0, \dots, n-1\}$ (and if it is not the case we can use a dictionary to map elements from A to that range).

2.1 Representation of the partition

Since every subset defines an equivalence class, it is equivalent to represent each $A_i \subseteq A$ with a single element $r_i \in A_i$, called the *representative* of A_i. Every other element of A_i is related to r_i by the properties of the previously defined binary relation. That is why a union-find data structure consists of an array $v[0 : n-1]$, where each element $i \in A$ either points to another element $j \in A$ belonging to the same class (i.e., $v[i] = j$) or is the representative of a class and is *marked specially* (in a few lines we are going to define what *marked specially* is).

For instance, Figure 1 provides an example of a union-find structure representing the partition $\Pi = \{\{0, 1, 2, 3\}, \{4, 5, 6\}\}$. In this example, the class $\{0, 1, 2, 3\}$ has 1 as its representative (hence, its position is marked with a special symbol). Elements 0 and 3 point to the representative (indicated by the fact that both positions in the array contain 1), while element 2 points to element 0, even though 0 is not the representative.

Although our implementation will consist of an array, the tree representation in the figure will help in understanding the union-find structure. Since we are representing a directed tree, we know that there is always a unique path from any element to the representative. As we move along this path, we will eventually reach the representative.

When defining how to represent the representative of a class, there are three natural ways to do so:

- A representative $i \in A$ of some class can be represented by its own number (i.e., $v[i] = i$). We can identify such an element as the representative of a class because representatives are marked by their own position in the array. For instance, in the example of Figure 1, we can mark element 1 with $v[1] = 1$ and element 5 with $v[5] = 5$.

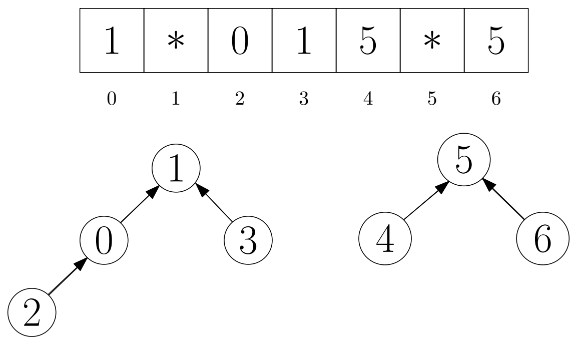

Figure 1: An example of a Union-Find with partition $\Pi = \{\{0,1,2,3\},\{4,5,6\}\}$

- A representative $i \in A$ of some class can be represented by the size of the tree it holds. Although the size of a tree is a natural number, we can use negative numbers to indicate that position i is the representative of a class, with the negative value representing the size (i.e., $v[i] = -\text{size}$). For instance, in the example of Figure 1, we can mark element 1 with $v[1] = -4$ and element 5 with $v[5] = -3$.

- A representative $i \in A$ of some class can be represented by the rank of the tree it holds (again, using negative numbers). The rank of a tree is related to its height (later, we will see that the rank does not always correspond to the exact height, but we will use it as an upper bound). For instance, in the example of Figure 1, we can mark element 1 with $v[1] = -2$ and element 5 with $v[5] = -1$.

2.2 Find Operation

Let us first implement the *find* operation, which consists of finding, for a given element $i \in A$, the representative r_i of the class to which i belongs.

From what we know about union-find, every element in a certain class will either point to the representative of the class or to another element that lies on the path between the representative and itself. Since we are representing a directed tree, we will eventually reach the representative.

Thus, our algorithm will consist of traversing this path until we reach the representative. Algorithm 1 gives a simple description of how we can do that.

Algorithm 1 Find Operation

```
function FIND($v$, $i$)
    $r_i \leftarrow i$
    while $\neg isRepresentative(v[r_i])$ do
        $r_i \leftarrow v[r_i]$
    end while
    return $r_i$
end function
```

One can check if an element is the representative of a class or not in $\Theta(1)$ time, but the implementation will depend on how the representative is defined. Still, it requires only one comparison.

The cost of the find operation depends on the height of the tree that represents the class. One might assume that, in the worst case, we are going to have a single unbalanced tree, and in such a case, the cost of the find operation is $O(n)$, which holds true if no heuristics are used in order to rearrange the tree. So let us apply heuristics during the find execution (switching some pointers) in order to speed up future calls of that operation (which is called *path compression*).

2.2.1 Full path compresion

The idea behind this heuristic is to traverse the path from i to r_i twice: once to determine r_i and once more to make every node within the path point to r_i. Although this doubles the cost of the find operation, we expect that a future find operation on the same class will be faster than usual. Algorithm 2 provides an implementation of such heuristic.

Algorithm 2 Find operation with path compression

```
function FIND($v$, $i$)
    if $isRepresentative(v[i])$ then
        return $i$
    else
        $v[i] = Find(v, v[i])$
        return $v[i]$
    end if
end function
```

2.2.2 Path Splitting

In order to not traverse twice the path we can make that every node points to its grandparent (except if they do not have). Although the tree will not be as balanced as with the previous heuristic we are not traversing twice the path which might impact in the performance of the operation. Algorithm 3 provides an implementation of such heuristic.

Algorithm 3 Find operation with path splitting

```
function FIND($v$, $i$)
    if $isRepresentative(v[i])$ then
        return $i$
    end if
    $i_1 \leftarrow i$
    $i_2 \leftarrow v[i_1]$
    while $\neg isRepresentative(v[i_2])$ do
        $v[i_1] = v[i_2]$
        $i_1 = i_2$
        $i_2 = v[i_2]$
    end while
    return $i_2$
end function
```

2.2.3 Path Halving

We can also make that every other node points to its grand-parent. Algorithm 4 provides an implementation of such heuristic.

Algorithm 4 Find operation with path splitting

```
function FIND($v$, $i$)
    if $isRepresentative(v[i])$ then
        return $i$
    end if
    $i_1 \leftarrow i$
    $i_2 \leftarrow v[i_1]$
    $setParent \leftarrow true$
    while $\neg isRepresentative(v[i_2])$ do
        if $setParent$ then
            $v[i_1] = v[i_2]$
        end if
        $setParent \leftarrow \neg setParent$
        $i_1 = i_2$
        $i_2 = v[i_2]$
    end while
    return $i_2$
end function
```

References

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