Disjoint-Set

Author: Hoai Dinh
ID: 1001704815
Course: CSE5311

Introduction

The disjoint-set data structure, also known as the union-find or merge-find data structure, is a data structure that stores a collection of sets. Each set in the disjoint-set data structure will have zero overlapping elements, which makes all the sets disjoint from each other.

Each set within a disjoint-set is identified by one of its elements, known as a representative element [1]. Since each set needs to have a representative element to identify it, then there cannot be a set in the disjoint-set data structure with zero elements. There are no requirements as to which element from the set will be the representative, but there may be some situations where a specific element (such as largest or smallest) is preferred [1].

Functionality

There are three key fundamental functions required for the disjoint-set data structure [1]:

- MakeSet(x): creates a new set with x as its sole element in the disjoint-set structure. The element x cannot already exist in another set of the disjoint-set.
- Union(x, y): performs the union operation between the set that x is an element of and the set that y is an element of. The result is that one set is merged with the other, and one of the representatives will become the representative of the merged set.
- FindSet(x): return the representative of the set that contains x

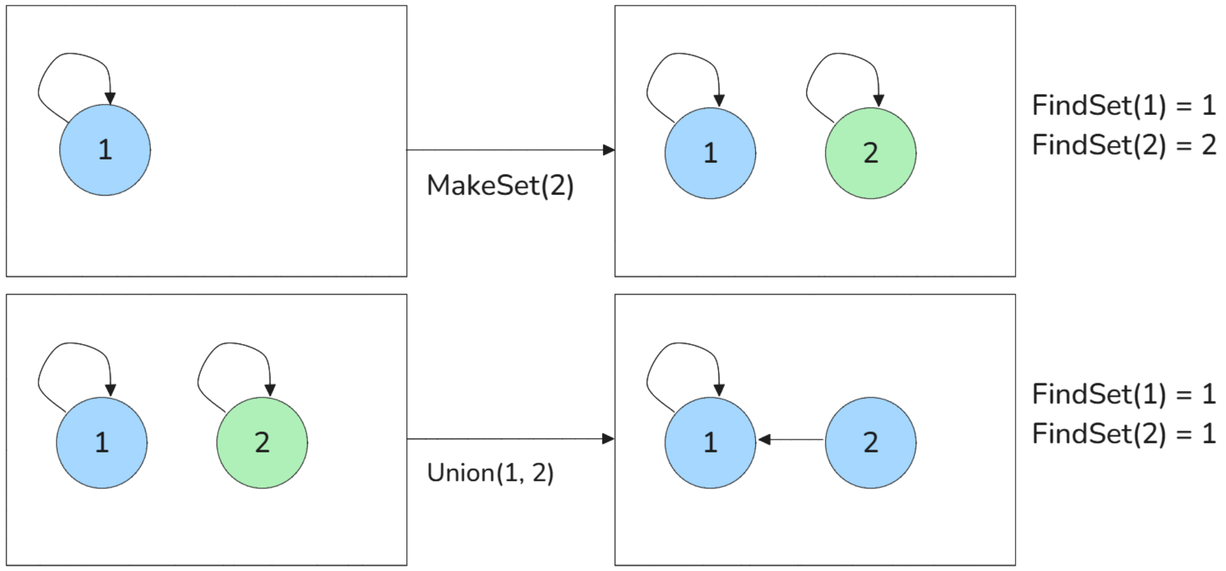


Fig. 1. Diagram showcasing operations done on a disjoint-set where MakeSet(1) is already called [2]

The figure above is a visualization of the disjoint-set data structure. Representative elements of each set are denoted with a loop to itself, and each set is color coded. The disjoint-set initially starts off empty, and calling MakeSet(1) will create a set with one element colored as blue. Since this is the only element in this set, then the representative element is 1. MakeSet(2) creates a new set, colored by green, with the representative element of that set being 2. Using FindSet(2) will return the representative element of the set that 2 is in, and the same happens with FindSet(1). Union(1, 2) takes the set that 1 is in and the set that 2 is in and performs the union operation. The result is a set where 2 is now part of the blue set. Here, FindSet(1) is equal to FindSet(2), as both elements are in the same set, with the representative element being 1.

There are different ways to implement disjoint-sets, either with arrays or with tree structures. This report will explore more with the tree structure implementation to better fit with the visual representation of disjoint-sets in Fig. 1.

Tree Structure Implementation

For the tree structure, each item will be represented by a node with a parent pointer and the data description, similar to Fig. 1. The parent pointer will eventually lead to the representative element of the set, and if the node is the representative element, then its parent pointer will point to itself.

The following are pseudo-code implementations of the core functions of a disjoint-set:

```
MakeSet(x)
    x.parent = x

Union(x, y)
    s1 = FindSet(x)
    s2 = FindSet(y)
    if s1 != s2
        s2.parent = s1

FindSet(x)
    if x.parent != x
        return FindSet(x.parent)
    return x
```

The time complexity of MakeSet(x) can be trivially analyzed as $\Theta(1)$. There are no additional traversals or function calls that would modify the time complexity. However, Union(x, y) and FindSet(x) are not as easy to analyze. Due to the simplicity of union, it is possible to create a set that is basically a linked list of N elements and subtrees. As a result, the FindSet(x) operation may end up becoming costly depending on what element x is.

Take for instance this figure below,

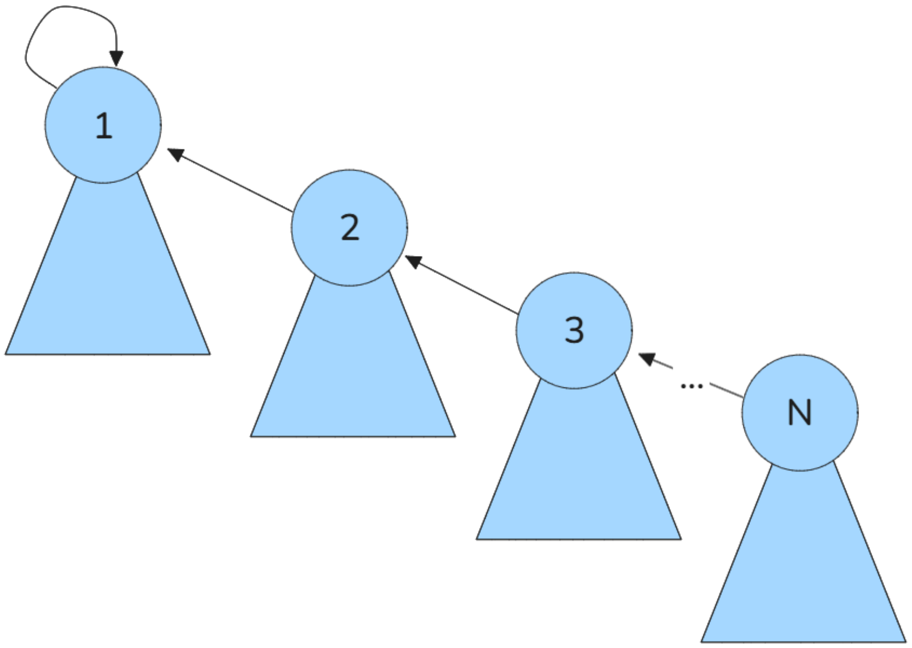


Fig. 2. A possible set configuration in the disjoint set with the pseudo code above [2]

Here in Fig. 2, it is possible that FindSet(x) can have a worst-case complexity of O(N), where N is the number of elements in the disjoint set. As a result, the performance of Union(x, y) will also suffer due to the fact that Union(x, y) uses FindSet. This scenario can be remedied with path compression and union by rank.

Performance Improvements - Path Compression

Here is FindSet(x) with path compression added:

```
FindSet(x)
    if x.parent != x
        x.parent = FindSet(x.parent)
    return x.parent
```

The change here is that x's parent is updated to whatever FindSet of its parent is. Adding path compression will prevent these long chains of subtrees from forming, as every call to FindSet(x) will update x and anyone else along the chain from x to the representative parent values to that representative node. This allows each call to FindSet to reduce the recursion chain of future FindSets by changing the parent pointer to be directly to the representative.

Overall, there will be an increase in performance with multiple FindSet calls with path compression.

Performance Improvements - Union by Rank

Union by rank is another optimization that can be added to help improve the time complexity of the Union(x, y) function. Here is the modified functions to allow union by rank:

```
MakeSet(x)
    x.parent = x
    x.rank = 0
Union(x, y)
    s1 = FindSet(x)
    s2 = FindSet(y)
    if s1 != s2
        if s1.rank < s2.rank
            s1.parent = s2
        else
            s2.parent = s1
            if s1.rank == s2.rank
                s1.rank = s1.rank + 1
```

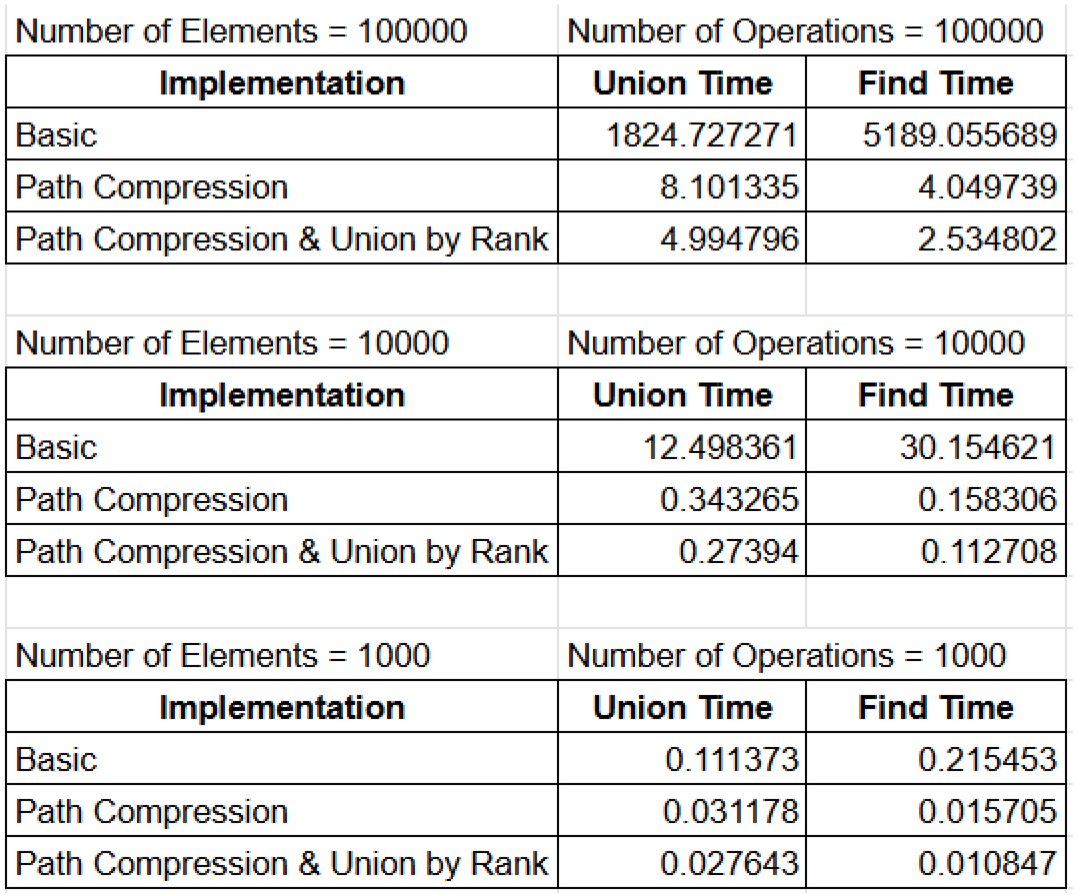
The idea behind union by rank is to minimize the height gain when merging two sets with varying lengths. Whenever two sets of the same rank are merged, one set will absorb the other, and increase its own rank. This rank is a heuristic that signifies the height of the tree. To minimize tree heights, and in turn improve performance of both FindSet and Union, a lower ranking set should become the child of the higher ranking set. This is important as it helps reduce the amount of times FindSet will be recursively called if an element deep in the tree is selected.

The rank value will not always represent the set's tree height, as when this method is used with FindSet path compression, the rank value does not decrease. In general both union by rank and path compression are used in tandem to reduce the time it takes to execute other Unions and FindSets by modifying the tree structure to be optimal for each set. The general goal is to have FindSet run in near constant time most of the time.

Time Complexity and Benchmarking Results

The space complexity of the disjoint-set data structure is simply $\Theta(N)$, where N is the number of elements in the collection of disjoint sets. The time complexity of the disjoint-set data structure varies based on implementation. As discussed previously, the worst case ended up being $O(N)$. However, taking into account both, path compression and union by rank, we can see that the amortized cost analysis of both Union and FindSet will see significant improvements. The complexity with path compression and union by rank ends up being $O(\alpha(N))$, where α is the inverse Ackerman function, which grows at an extremely slow rate [1].

Table 1 below showcases a benchmark between the different disjoint-set (or union-find) implementations. It serves to demonstrate the performances between a basic implementation, one with path compression and one with both path compression and union by rank. It also helps illustrate the time complexity of the operations and how they change when introducing the performance improvements discussed previously.



Number of Elements = 100000	Number of Operations = 100000	
Implementation	**Union Time**	**Find Time**
Basic	1824.727271	5189.055689
Path Compression	8.101335	4.049739
Path Compression & Union by Rank	4.994796	2.534802

Number of Elements = 10000	Number of Operations = 10000	
Implementation	**Union Time**	**Find Time**
Basic	12.498361	30.154621
Path Compression	0.343265	0.158306
Path Compression & Union by Rank	0.27394	0.112708

Number of Elements = 1000	Number of Operations = 1000	
Implementation	**Union Time**	**Find Time**
Basic	0.111373	0.215453
Path Compression	0.031178	0.015705
Path Compression & Union by Rank	0.027643	0.010847

Table 1. Benchmark between the different implementations of the disjoint-set (union-find) data structure where the Time is in milliseconds (ms) [2]

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, third edition*. 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1614191

[2] H. Dinh, *CSE5311-Disjoint-Sets*, GitHub repository, 2025. [Online]. Available: https://github.com/hoaihdinh/CSE5311-Disjoint-Sets