Report on

Generic Disjoint-Set Data Structure

Submitted in partial fulfillment of the requirements for ***Sem VI***

UE18CS331 Generic Programming

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Neel Kamath	**PES2201800467**
Tushar Goankar	**PES2201800352**
Naveen K Murthy	**PES2201800051**

Under the guidance of
Prof. N S Kumar
Visiting Professor
PES University, Bengaluru

January – May 2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

1. Introduction

The disjoint-set data structure, a.k.a union-find, merge-find or DSU data structure, maintains and operates on a collection of sets. These sets are disjoint, i.e. no element belongs to more than one set.

The DSU data structure supports two core operations:

1. `find()`: Given an element of a set, this function returns a particular element belonging to its subset, often referred to as the *component representative*.
2. `unite()`: This method joins together two disjoint sets.

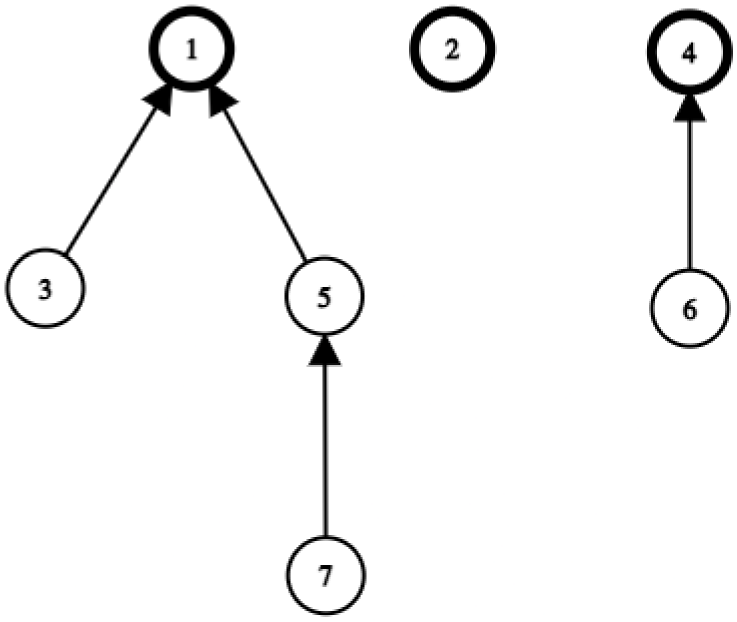
The main goal of this project is to demonstrate the usage of a generically-written DSU data structure in the context of solving commonly encountered Graph Theory problems such as detecting cycles, finding connected components, building minimum spanning trees, etc.

2. Theoretical Overview

The DSU data structure stores each component/set as a directed, rooted tree.
Consider the following disjoint sets of elements:

`{1,3,5,7}, {2}, {4,6}`

One way that these sets can be represented graphically is as follows:

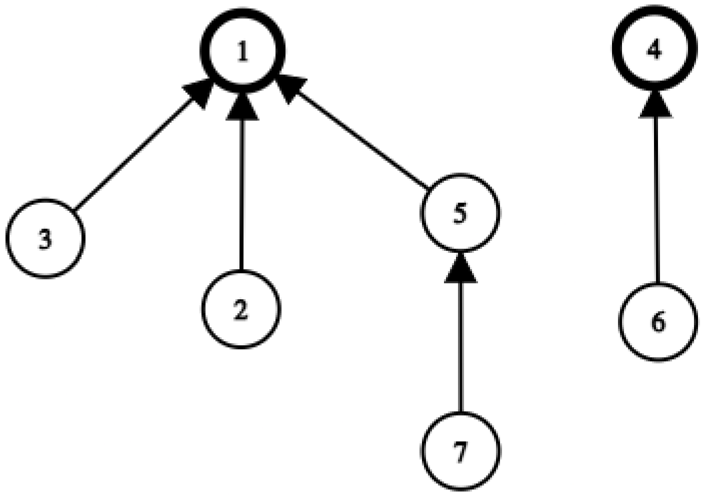


Each set allocates one element to be representative of all elements belonging to that set. Note that for single element sets, the lone element is itself the representative.

In the above example, these component representatives are 1, 2 and 4 respectively (shown in bold).

We can find the component representative of any particular element by tracing our way from that element to its tree's root via its parents.
Eg. The parent of 7 is 1 (by following the path 7→5→1).

Joining two elements (belonging to different sets), or equivalently, merging two sets is thus straight-forward: we simply make the component representative of the first element point to that of the second element (or vice-versa).
Eg. Consider merging the subsets `{1,3,5,7}` and `{2}`. This results in the following representation:



The original elements are now grouped into the following sets and the underlined elements are the component representatives.

`{1,3,5,7,2}, {4,6}`

The complexities of the two core operations are as follows:

Functionality	**Time Complexity (Amortised)**	**Space Complexity**
`find()`	O(α(n))*	O(1)
`merge()`	O(α(n))*	O(1)

*α() is the inverse Ackermann function which is approximately equal to O(1).

3. Implementation Details

Template Class

This project utilizes C++ templates to implement the operations mentioned above. The actual DSU data structure itself is a template class with member functions that accomplish various functionality.

```cpp
template <typename T>
class disjoint_set {
private:
    unordered_map<T, size_t> set_size;

public:
    unordered_map<T, T> link;

    disjoint_set(){};
    disjoint_set(disjoint_set<T> const&) = default;
    disjoint_set<T>& operator=(const disjoint_set<T>&) = default;

    /** Core functionality **/
    void insert(T a);
    void insert(disjoint_set<T>);
    void unite(T a, T b);
    void merge(T a, T b);
    T find(T a);
    bool same(T a, T b);

    /** Overloaded operators **/
    T& operator[](T);
    disjoint_set<T>& operator+=(const disjoint_set<T>& rhs);
    disjoint_set<T> operator+(const disjoint_set<T>& rhs);
    bool operator==(const disjoint_set<T>& rhs) const;
    bool operator!=(const disjoint_set<T>& rhs) const;

    /** Helper Functions for Debugging **/
    void disp_link() const;
    void disp_set_size() const;
    friend std::ostream& operator<<(std::ostream&, const disjoint_set&);
};
```

This class can be utilized by instantiating it for the required type of element:

```
disjoint_set<double> ds;
ds.insert(3.14);
ds.insert(2.71);
ds.insert(1.414);
ds.insert(6.28);

ds.join(3.14, 2.71);
ds.join(6.28, 3.14);

cout << boolalpha;
cout << ds.same(3.14, 6.28) << '\n'; // true
```

Path Compression

The `find()` function implements the idea of *path compression*: each time we traverse up the tree to find the component representative, the parent pointer of each encountered node is reassigned to point to the root/representative element. This essentially *flattens* the tree and results in quite a large performance improvement.

```
template <typename T>
T disjoint_set<T>::find(T a) {
    if (a == link[a]) return a;
    return link[a] = find(link[a]);  // we set the direct parent to the root
of the set to reduce path length.
}
```

Generic Types

The example discussed in Section 2 described a DSU data structure in which the elements were all plain integers (1-7).
The code developed for this project allows more complex sets such as:

`{ “Alice”,“Bob” }`
`{ “Charlie”,“Daniel” }`
`{ “Erica” }`
	Each element is a string.

`{ (1.0,1.1),(1.3,1.4),(1.5,1.7) }`
`{ (2.0,2.1) }`
`{ (3.0,3.1),(4.0,4.1) }`
	Each element is a pair of floating-point numbers.

`{ {1,2,3},{4,5},{9} }`
`{ {6,8},{7},{1,2,4},{4,5,7} }`
	Each element is a vector of integers.

`{ X(‘z’) }`	Each element is a user-defined

`{ X('y'),X('w'),X('q') }`
`{ X('a'),X('c') }`
	type, i.e. an instance of a class 'X'.

This is achieved by providing custom hash functions for the various C++ containers.

4. Applications

This project demonstrates the following use cases of a generic DSU data structure:

- Detecting a cycle in a graph.
- Generating the minimum spanning tree of a weighted graph.
- Identifying connected components in a graph.
- Solving the Job Sequencing problem

5. Scope for Improvement

- Provide stronger hash functions with collision minimization as the goal.
- Implement a way to visualize the underlying component trees in the console.
- Provide ways for this data structure to interact with other containers provided in the C++ Standard Library.

6. References

- MIT Lecture Notes: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec16.pdf
- Laaksonen, A. 2017. *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests*. Cham, Switzerland: Springer.
- An Introduction to the USA Computing Olympiad by Darren Yao: https://darrenyao.com/usacobook/cpp.pdf
- https://csacademy.com/lesson/disjoint_data_sets/
- https://usaco.guide/gold/dsu/