Union Find

Josh Wilcox (jw14g24@soton.ac.uk)

February 24, 2025

Contents

1 Dynamic Connectivity 2

2 Union-Find data structure 2

2.1 Quick-Find . . . 2

2.1.1 Java Implementation . . . 2

2.1.2 Time Complexity . . . 3

2.2 Quick-Union . . . 3

2.3 Java Implementation . . . 4

3 Improving Union-Find 5

3.1 Use Weighting in Union . . . 5

3.2 Path Compression . . . 5

3.2.1 Path Compression Example . . . 6

3.3 Complexity using both Quick Union and Path Compression . . . 7

1 Dynamic Connectivity

- Union Command: Connects two objects
- `isConnected()` : Queries whether two objects are connected by a **path**
- Connectivity can change over time - hence the term *dynamic*

2 Union-Find data structure

- **Union**: Merge two equivalence classes into one
- **Find**: Check which class a given element belongs to

2.1 Quick-Find

- For an Integer array `id[]` of size N
- Elements p and q are equivalent $\iff$ thir IDs are the same
- `isConnected(p,q)` would check whether p and q have the same ID in the `id[]` array
- `find(p)` would simply return p's id
- `union(p,q)` would merge the components containing p and q by changing the id of *all elements* with `id[p]` to `id[q]`

2.1.1 Java Implementation

```java
// QuickFindUF implements the Quick-Find algorithm for the Union-Find (disjoint-set) data structure.
// It provides basic operations like union and find, and it can quickly check if two elements are in the same set.
public class QuickFindUF {
    // The 'id' array holds the component identifier for each element.
    // Two elements are connected if they have the same id.
    private int[] id;

    // Constructor: Creates a new QuickFindUF object with N elements (0 through N-1)
    // Initially, each element is in its own component (i.e., id[i] == i).
    public QuickFindUF(int N) {
        id = new int[N]; // Allocate memory for the id array
        // Initialize each element to be its own root.
        for (int i = 0; i < N; i++) {
            id[i] = i; // Each element starts disconnected from all other elements.
        }
    }

    // isConnected: Determines whether the two elements 'p' and 'q' are in the same component.
    // It returns true if both elements have the same component identifier.
    public boolean isConnected(int p, int q) {
        // Compare the identifiers of p and q; if they are equal, p and q are connected.
```

```
22          return id[p] == id[q];
23      }
24
25      // find: Returns the component id for the element 'p'.
26      // This is a simple lookup in the id array.
27      public int find(int p) {
28          // In the quick-find algorithm, the value in id[p] is directly the
            ↪ component id.
29          return id[p];
30      }
31
32      // union: Merges the components containing elements 'p' and 'q'.
33      // It does so by changing all entries with the component id of 'p' to the
        ↪ component id of 'q'.
34      public void union(int p, int q) {
35          // Get the component identifiers for p and q.
36          int pid = id[p];
37          int qid = id[q];
38
39          // If p and q are already in the same component, no changes are necessary.
40          if (pid == qid) return;
41
42          // Iterate over the entire id array.
43          // Every time we find an element with an id equal to pid (the id for p),
44          // we change it to qid (the id for q), effectively merging the two
            ↪ components.
45          for (int i = 0; i < id.length; i++) {
46              // Check if the element i has the same component id as p.
47              if (id[i] == pid) {
48                  id[i] = qid; // Update the id to the identifier of q's component.
49              }
50          }
51      }
52  }
```

2.1.2 Time Complexity

- Initialisation - $\mathcal{O}(N)$
- Find: - $\mathcal{O}(1)$
- Union - $\mathcal{O}(N)$

2.2 Quick-Union

- Uses another integer array of size N
- Components are stored as *trees*
- The whole structure is a **forest**
- The id of an element:
  - Is its parent - if not a root of the relevant tree in the forest

– Is itself - if a root of the relevant tree in the forest

2.3 Java Implementation

```java
public class QuickUnionUF {
    // 'id' stores the parent of each element.
    // The union-find structure represents disjoint sets as trees.
    private int[] id;

    // Constructor: initializes N elements,
    // where each element is initially its own root.
    public QuickUnionUF(int N) {
        id = new int[N]; // allocate memory for the 'id' array
        // Each element starts in its own set: id[i] = i.
        for (int i = 0; i < N; i++) {
            id[i] = i;
        }
    }

    // Helper method: Finds the root of element 'i'.
    // The root is the element that is its own parent.
    // This method follows parent pointers until a root is reached.
    private int root(int i) {
        // Continue traversing the tree until the root is found.
        while (i != id[i]) {
            // In this basic implementation, no path compression is performed.
            i = id[i];
        }
        return i;
    }

    // isConnected: Determines if elements 'p' and 'q' are in the same set.
    // Two elements are connected if they share the same root.
    public boolean isConnected(int p, int q) {
        return root(p) == root(q);
    }

    // find: Returns the root (or representative) of the set containing 'p'.
    // Essentially, this is just an alias for the root method.
    public int find(int p) {
        return root(p);
    }

    // union: Merges the set containing element 'p' with the set containing element
    ↪ 'q'.
    // It does so by linking the root of 'p' to the root of 'q'.
    public void union(int p, int q) {
        // Find the root of each element.
        int i = root(p);
```

```
45          int j = root(q);
46
47          // If both elements have the same root, they are already connected.
48          if (i == j) {
49              return;
50          }
51
52          // Merge: Make the root of p point to the root of q,
53          // thus combining the two trees into one.
54          id[i] = j;
55      }
56  }
```

3 Improving Union-Find

- Union-Find is even worse
- Initialise, Find, Union are all $\mathcal{O}(N)$
- However it can be improved using some techniques

3.1 Use Weighting in Union

- Create a `size[]` array that measures the total weight of each tree
- Modify union to:
  - Merge smaller tree into the larger tree - **instead of just first to second**
  - Keep the `size[]` array updated
- This makes the trees balanced and the time complexity of find and union become $\mathcal{O}(log_2(N))$

```
1  if (size[i] < size[j]){
2      id[i] = j;
3      size[j] += size[i]
4  } else{
5      id[j] = i;
6      size[i] += size[j]
7  }
```

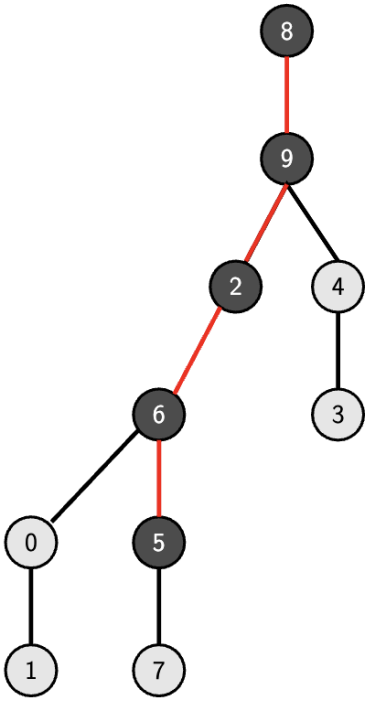
3.2 Path Compression

- When performing `find(p)` in quick union, we are actually just looking for a *path* from element p to its root
- Therefore, it is better to keep the tree as **flat** as possible
- Method:
  - Set the `id` of the items in the path to be the root each time a root is computed

3.2.1 Path Compression Example

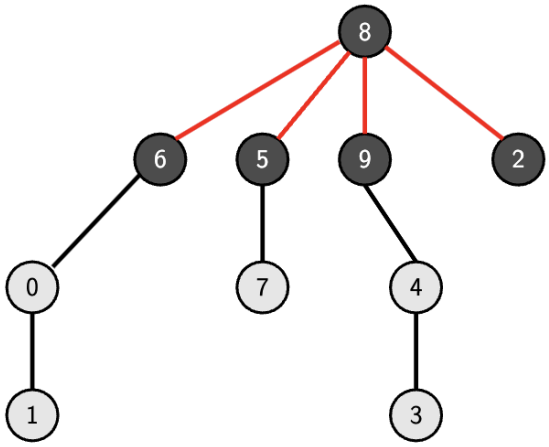
Improvement 2: Path compression

`find(5)`



Improvement 2: Path compression

`find(5)`



3.3 Complexity using both Quick Union and Path Compression

- Initialisation: $\mathcal{O}(N)$
- Find: $\mathcal{O}(log*(N))$
- Union: $\mathcal{O}(log*(N))$