Union-Find

Arnar Bjarni Arnarson

September 22, 2025

School of Computer Science
Reykjavík University

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element
- Supports two operations efficiently: `find(x)` and `union(x,y)`.

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element
- Supports two operations efficiently: `find(x)` and `union(x,y)`.
- Operation `find(x)` finds the representative of the set x is in

Union-Find

- We have n items
- Maintains a collection of disjoint sets (or equivalently, an equivalence relation)
- Each of the n items is in exactly one set
- We represent each set with one of its members, a representative element
- Supports two operations efficiently: `find(x)` and `union(x,y)`.
- Operation `find(x)` finds the representative of the set x is in
- Operation `union(x, y)` unions the sets of which x and y are members.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `union(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `union(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.
- `union(2, 4)` then results in $\{\{1, 3\}, \{2, 4, 5\}\}$.
- `union(1, 4)` finally results in $\{\{1, 2, 3, 4, 5\}\}$.

Union-Find

- It is generally initialized with all items being in their own set.
- So for $n = 5$ we start out with $\{\{1\}, \{2\}, \{3\}, \{4\}, \{5\}\}$.
- `union(1, 3)` then changes this to $\{\{1, 3\}, \{2\}, \{4\}, \{5\}\}$.
- `union(2, 5)` then results in $\{\{1, 3\}, \{2, 5\}, \{4\}\}$.
- `union(2, 4)` then results in $\{\{1, 3\}, \{2, 4, 5\}\}$.
- `union(1, 4)` finally results in $\{\{1, 2, 3, 4, 5\}\}$.
- At any given point `find(x)` returns some value in the same set as x.
- The important bit is that `find(x)` returns the same value for all elements of the same set, the representative.

Union-Find

- We can do this by maintaining an array of parents, letting the i-th value be the index of the parent of the i-th item.
- If a value has no parent, we can denote this somehow, make it its own parent, give it the value -1, exactly what we do is not important.

Union-Find

- We can do this by maintaining an array of parents, letting the i-th value be the index of the parent of the i-th item.
- If a value has no parent, we can denote this somehow, make it its own parent, give it the value -1, exactly what we do is not important.
- To get the representative of x we go to the parent of our current item (starting at x) until the item has no parent.
- Then to unite x, y we simply make the representative of x the parent of the representative of y.

Quick-Find

```
struct union_find {
    vector<int> parent;
    union_find(int n) : parent(n) {
        for(int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
    int find(int x) {
        return parent[x];
    }
    void unite(int x, int y) {
        for (int i = 0; i < n; i++) {
            if (find(i) == find(x)) {
                parent[i] = find(y);
            }
        }
    }
};
```

- The time complexity of the find operation is $\mathcal{O}(1)$.
- The time complexity of the unite operation is $\mathcal{O}(n)$.
- This is not good enough, so lets try a different approach.

Quick-Union

```
struct union_find {
    vector<int> parent;
    union_find(int n) : parent(n) {
        for(int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
    int find(int x) {
        return parent[x] == x ? x : find(parent[x]);
    }
    void unite(int x, int y) {
        parent[find(x)] = find(y);
    }
};
```

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.
- One method is to use what is known as small-to-large merging, where the smaller group's leader is made to point to the larger group's leader.
- This ensures the height increases by 1 as a group's size doubles, resulting in $\mathcal{O}(\log n)$ complexity.

Union-Find

- The problem here is that for specific queries, the parent chains may end up being of length $\mathcal{O}(n)$, making each query linear.
- The key to making this more efficient is making those chains shorter.
- One method is to use what is known as small-to-large merging, where the smaller group's leader is made to point to the larger group's leader.
- This ensures the height increases by 1 as a group's size doubles, resulting in $\mathcal{O}(\log n)$ complexity.
- We can also do this by flattening the chain each time we query `find`, so the amortized complexity becomes good.
- Here the worst case is still $\mathcal{O}(n)$ but the amortized complexity is $\mathcal{O}(\alpha(n))$ which may as well be a constant, as it is < 5 for n equal to the number of atoms in the observable universe.

Weighted Quick-Union

```
struct union_find {
    vector<int> parent, sizes;
    union_find(int n) : parent(n), sizes(n) {
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            sizes[i] = 1;
        }
    }
    int find(int x) {
        if(parent[x] == x) return x;
        return find(parent[x]);
    }
    void unite(int x, int y) {
        int xp = find(x);
        int yp = find(y);
        if (xp == yp) return;
        if (sizes[xp] > sizes[yp]) swap(xp, yp);
        parent[xp] = parent[yp];
        sizes[yp] += sizes[xp];
    }
};
```

Quick-Union with path compression

```
struct union_find {
    vector<int> parent;
    union_find(int n) : parent(n) {
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
    int find(int x) {
        if(parent[x] == x) return x;
        return parent[x] = find(parent[x]);
    }
    void unite(int x, int y) {
        parent[find(x)] = find(y);
    }
};
```

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data
  - Number of different sets currently
  - Current size of the set containing x
  - An iterable list of all elements of the set containing x

Union-Find applications

- Union-Find maintains a collection of disjoint sets
- When are we dealing with such collections?
- Usually when we want to work with equivalence relations like graph connectivity
- By modifying the data structure it can also contain more queryable data
  - Number of different sets currently
  - Current size of the set containing x
  - An iterable list of all elements of the set containing x
- When tracking size you can use it to always perform small-to-large merges for $\mathcal{O}(\log n)$ time complexity.

Example problem: Skolavslutningen

- https://open.kattis.com/problems/skolavslutningen