CPCFI

Lecture: DISJOINT SET UNION

Unit: 2 - NON LINEAR DATA STRUCTURES

Instructor: NÉSTOR

DISJOINT SET UNION (UNION FIND) - DSU

- GIVEN SEVERAL SETS, A DSU WILL BE ABLE TO COMBINE ANY TWO SETS AND TELL EFFICIENTLY IN WHICH SET A SPECIFIC ELEMENT IS

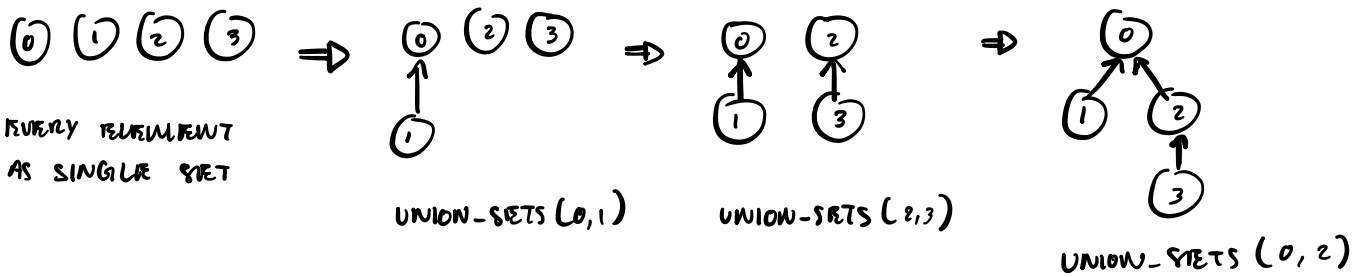
- MAIN OPERATIONS:

1. MAKE_SET(v): CREATES A NEW SET
2. UNION_SETS(a, b): MERGES TWO SETS → WE WANT TO OPTIMIZE THIS FUNCTION
3. FIND_SET(v): RETURNS THE REPRESENTATIVE OF THE SET THAT CONTAINS v
   - CAN BE USED TO CHECK IF TWO ELEMENTS BELONG IN THE SAME SET

ALMOST ALL IN O(1)

[BUILDING DSU]

- STORE SETS AS A TREE, WITH THE ROOT BEING THE LEADER (REPRESENTATIVE)
  - EACH TREE CORRESPONDS TO ONE SET

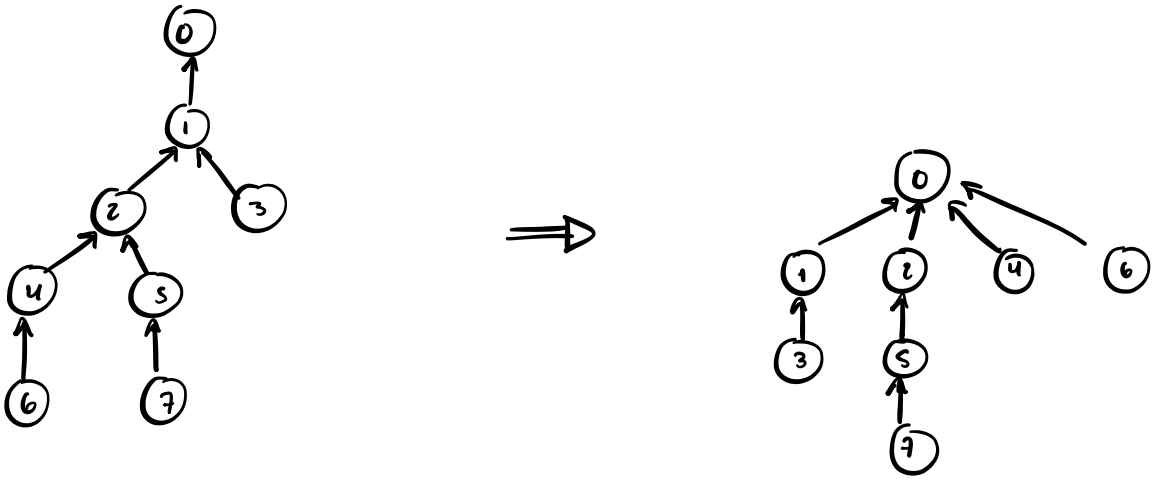


[NAIVE IMPLEMENTATION] - C++ CODE

- ARRAY PARENT STORES ALL SETS RELATIONSHIPS
- INEFFICIENT → FIND_SET(v) TAKES O(n) TIME (OUR GOAL IS O(1))
- GENERATES LONG CHAINS WHICH EACH FIND_SET CAN TAKE O(n) TIME

[PATH COMPRESSION] - 1ST OPTIMIZATION

- SPEEDS UP FIND_SET



NAIVE DSU

- LONGER PATHS $O(n)$
- FIND_SET(6) = 0

  $6 \to 4 \to 2 \to 1 \to \underline{0}$

PATH COMPRESSION

- MAKE PATHS SHORTER $O(\log n)$
- FIND_SET(6) = 0

  $6 \to \underline{0}$
- VISITED NODES ARE ATTACHED DIRECTLY TO THE REPRESENTATIVE

[UNION BY SIZE / DEPTH] - 2ND OPTIMIZATION

- OPTIMIZES UNIONSETS(a, b)
- CHOOSE WHICH TREE GETS ATTACHED (AS OPPOSED TO ALWAYS ATTACHING THE SECOND TO THE FIRST)
- IDEA: ATTACH TREE WITH LOWER RANK TO THE ONE WITH HIGHER RANK
  - ① USING SIZE OF THE TREE
  - ② USING DEPTH OF THE TREE

  } USE ANY OF THEM

APPLICATIONS

① CONNECTED COMPONENTS → FINDSET(a) == FINDSET(b) ?

(2) COMPRESS JUMPS ALONG SEGMENTS / PAINTING SUBARRAYS OFFLINE

One common application of the DSU is the following: There is a set of vertices, and each vertex has an outgoing edge to another vertex. With DSU you can find the end point, to which we get after following all edges from a given starting point, in almost constant time.

A good example of this application is the **problem of painting subarrays**. We have a segment of length L, each element initially has the color 0. We have to repaint the subarray $[l, r]$ with the color c for each query (l, r, c). At the end we want to find the final color of each cell. We assume that we know all the queries in advance, i.e. the task is offline.

For the solution we can make a DSU, which for each cell stores a link to the next unpainted cell. Thus initially each cell points to itself. After painting one requested repaint of a segment, all cells from that segment will point to the cell after the segment.

Now to solve this problem, we consider the queries **in the reverse order**: from last to first. This way when we execute a query, we only have to paint exactly the unpainted cells in the subarray $[l, r]$. All other cells already contain their final color. To quickly iterate over all unpainted cells, we use the DSU. We find the left-most unpainted cell inside of a segment, repaint it, and with the pointer we move to the next empty cell to the right.

Here we can use the DSU with path compression, but we cannot use union by rank / size (because it is important who becomes the leader after the merge). Therefore the complexity will be $O(\log n)$ per union (which is also quite fast).

Implementation:

```
for (int i = 0; i <= L; i++) {
    make_set(i);
}

for (int i = m-1; i >= 0; i--) {
    int l = query[i].l;
    int r = query[i].r;
    int c = query[i].c;
    for (int v = find_set(l); v <= r; v = find_set(v)) {
        answer[v] = c;
        parent[v] = v + 1;
    }
}
```

There is one optimization: We can use **union by rank**, if we store the next unpainted cell in an additional array `end[]`. Then we can merge two sets into one ranked according to their heuristics, and we obtain the solution in $O(\alpha(n))$.

(3) SUPPORT DISTANCES UP TO THE REPRESENTATIVE

Sometimes in specific applications of the DSU you need to maintain the distance between a vertex and the representative of its set (i.e. the path length in the tree from the current node to the root of the tree).

If we don't use path compression, the distance is just the number of recursive calls. But this will be inefficient.

However it is possible to do path compression, if we store the **distance to the parent** as additional information for each node.

In the implementation it is convenient to use an array of pairs for `parent[]` and the function `find_set` now returns two numbers: the representative of the set, and the distance to it.

```
void make_set(int v) {
    parent[v] = make_pair(v, 0);
    rank[v] = 0;
}

pair<int, int> find_set(int v) {
    if (v != parent[v].first) {
        int len = parent[v].second;
        parent[v] = find_set(parent[v].first);
        parent[v].second += len;
    }
    return parent[v];
}

void union_sets(int a, int b) {
    a = find_set(a).first;
    b = find_set(b).first;
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = make_pair(a, 1);
        if (rank[a] == rank[b])
            rank[a]++;
    }
}
```

(4) SUPPORT THE PARITY OF THE PATH LENGTH / CHECKING BIPARTITENESS ONLINE

In the same way as computing the path length to the leader, it is possible to maintain the parity of the length of the path before him. Why is this application in a separate paragraph?

The unusual requirement of storing the parity of the path comes up in the following task: initially we are given an empty graph, it can be added edges, and we have to answer queries of the form "is the connected component containing this vertex **bipartite**?".

To solve this problem, we make a DSU for storing of the components and store the parity of the path up to the representative for each vertex. Thus we can quickly check if adding an edge leads to a violation of the bipartiteness or not: namely if the ends of the edge lie in the same connected component and have the same parity length to the leader, then adding this edge will produce a cycle of odd length, and the component will lose the bipartiteness property.

The only difficulty that we face is to compute the parity in the `union_find` method.

If we add an edge (a, b) that connects two connected components into one, then when you attach one tree to another we need to adjust the parity.

Let's derive a formula, which computes the parity issued to the leader of the set that will get attached to another set. Let x be the parity of the path length from vertex a up to its leader A, and y as the parity of the path length from vertex b up to its leader B, and t the desired parity that we have to assign to B after the merge. The path contains the of the three parts: from B to b, from b to a, which is connected by one edge and therefore has parity 1, and from a to A. Therefore we receive the formula ($\oplus$ denotes the XOR operation):

$$t = x \oplus y \oplus 1$$

Thus regardless of how many joins we perform, the parity of the edges is carried from on leader to another.

We give the implementation of the DSU that supports parity. As in the previous section we use a pair to store the ancestor and the parity. In addition for each set we store in the array `bipartite[]` whether it is still bipartite or not.

```
void make_set(int v) {
    parent[v] = make_pair(v, 0);
    rank[v] = 0;
    bipartite[v] = true;
}

pair<int, int> find_set(int v) {
    if (v != parent[v].first) {
        int parity = parent[v].second;
        parent[v] = find_set(parent[v].first);
        parent[v].second ^= parity;
    }
    return parent[v];
}
```

```
void add_edge(int a, int b) {
    pair<int, int> pa = find_set(a);
    a = pa.first;
    int x = pa.second;

    pair<int, int> pb = find_set(b);
    b = pb.first;
    int y = pb.second;

    if (a == b) {
        if (x == y)
            bipartite[a] = false;
    } else {
        if (rank[a] < rank[b])
            swap (a, b);
        parent[b] = make_pair(a, x^y^1);
        bipartite[a] &= bipartite[b];
        if (rank[a] == rank[b])
            ++rank[a];
    }
}

bool is_bipartite(int v) {
    return bipartite[find_set(v).first];
}
```

(5) OFFLINE RMQ (RANGE MINIMUM QUERY) IN O(α(n)) ON AVG / ARPA'S TRICK

We are given an array `a[]` and we have to compute some minima in given segments of the array.

The idea to solve this problem with DSU is the following: We will iterate over the array and when we are at the `i`th element we will answer all queries `(L, R)` with `R == i`. To do this efficiently we will keep a DSU using the first `i` elements with the following structure: the parent of an element is the next smaller element to the right of it. Then using this structure the answer to a query will be the `a[find_set(L)]`, the smallest number to the right of `L`.

This approach obviously only works offline, i.e. if we know all queries beforehand.

It is easy to see that we can apply path compression. And we can also use Union by rank, if we store the actual leader in an separate array.

```
struct Query {
    int L, R, idx;
};

vector<int> answer;
vector<vector<Query>> container;
```

`container[i]` contains all queries with `R == i`.

```
stack<int> s;
for (int i = 0; i < n; i++) {
    while (!s.empty() && a[s.top()] > a[i]) {
        parent[s.top()] = i;
        s.pop();
    }
    s.push(i);
    for (Query q : container[i]) {
        answer[q.idx] = a[find_set(q.L)];
    }
}
```

(6) OFFLINE LCA IN A TREE IN $O(\alpha(n))$ ON AVG

↳ TARJAN'S OFFLINE ALGORITHM

(7) FINDING BRIDGES ONLINE IN $O(\alpha(n))$ ON AVG

One of the most powerful applications of DSU is that it allows you to store both as **compressed and uncompressed trees**. The compressed form can be used for merging of trees and for the verification if two vertices are in the same tree, and the uncompressed form can be used - for example - to search for paths between two given vertices, or other traversals of the tree structure.

In the implementation this means that in addition to the compressed ancestor array `parent[]` we will need to keep the array of uncompressed ancestors `real_parent[]`. It is trivial that maintaining this additional array will not worsen the complexity: changes in it only occur when we merge two trees, and only in one element.

On the other hand when applied in practice, we often need to connect trees using a specified edge other that using the two root nodes. This means that we have no other choice but to re-root one of the trees (make the ends of the edge the new root of the tree).

At first glance it seems that this re-rooting is very costly and will greatly worsen the time complexity. Indeed, for rooting a tree at vertex v we must go from the vertex to the old root and change directions in `parent[]` and `real_parent[]` for all nodes on that path.

However in reality it isn't so bad, we can just re-root the smaller of the two trees similar to the ideas in the previous sections, and get $O(\log n)$ on average.