Disjoin Set Union

Tuan Kiet Phan - Duy Vu Hoang

July 2024

23APCS2
HCMUS - VNUHCM

Contents

§1 Introduction

Disjoint Set Union (DSU), also known as Union-Find, is a fundamental data structure widely used in various fields of computer science, particularly in graph theory and network connectivity. It efficiently manages a collection of disjoint sets and supports two primary operations: union and find. The union operation merges two sets into a single set, while the find operation determines the representative or leader of the set containing a given element.

In graph-related problems, DSU is instrumental in handling scenarios involving connected components. A connected component in a graph is a subgraph where any two vertices are connected to each other by a path, and no vertex is connected to any vertex outside the subgraph. DSU is exceptionally efficient for dynamically connecting nodes and querying their connectivity, which makes it ideal for applications such as network connectivity, image processing, and clustering.

Consider the following problem: Given a graph with n vertices. There are two main operations on this graph:

- Add an edge between vertex x and vertex y.
- Print YES if x and y in the same connected component. Otherwise, print NO.

If we consider each vertex in the graph as an element and each connected component in the graph as a set, the first operation becomes merging the sets containing elements x and y into a single set. The second operation becomes querying whether the elements x and y are in the same set.

To achieve efficient performance, the DSU data structure employs two key optimization techniques: path compression and union by rank. Path compression flattens the structure of the tree whenever find is called, ensuring that all nodes directly point to the root. Union by rank ensures that the smaller tree is always added under the root of the larger tree during union operations, maintaining a balanced tree structure. These optimizations result in nearly constant time complexity for both union and find operations, making DSU highly efficient.

Real-world applications of DSU include Kruskal's algorithm for finding the Minimum Spanning Tree (MST) of a graph, network connectivity problems, and clustering analysis in data mining. The efficiency and simplicity of DSU make it an indispensable tool in these and many other applications.

In summary, the Disjoint Set Union data structure is a powerful tool for managing and querying connectivity in a set of elements. Its efficiency and simplicity make it an indispensable component in many algorithms and applications where dynamic connectivity is a concern.

§2 History

The concept of disjoint-set forests was first introduced by Bernard A. Galler and Michael J. Fischer in 1964. In 1973, their time complexity was improved to $O(\log(n))$ (the iterated logarithm of n) by Hopcroft and Ullman. In 1975, Robert Tarjan proved that the time complexity of the algorithm is bounded by $O(m\alpha(n))$ (where $\alpha(n)$ is the inverse Ackermann function), and he showed this bound to be tight. In 1979, Tarjan also established that this was the lower bound for a certain class of algorithms, including the Galler-Fischer structure.

In 1989, Fredman and Saks demonstrated that $\omega(\alpha(n))$ (amortized) words of $O(\log(n))$ bits must be accessed by any disjoint-set data structure per operation, proving the optimality of the data structure in this model.

In 1991, Galil and Italiano published a comprehensive survey on data structures for disjoint sets. This was followed in 1994 by Richard J. Anderson and Heather Woll, who described a parallelized version of Union-Find that never needs to block.

In 2007, Sylvain Conchon and Jean-Christophe Filliâtre developed a semi-persistent version of the disjoint-set forest data structure and formalized its correctness using the proof assistant Coq. A semi-persistent structure means that previous versions of the structure are efficiently retained, but accessing previous versions invalidates later ones. Their fastest implementation achieves performance nearly as efficient as the non-persistent algorithm, although they did not perform a complexity analysis.

Variants of disjoint-set data structures with better performance for restricted classes of problems have also been considered. Gabow and Tarjan demonstrated that if the possible unions are restricted in specific ways, then a truly linear time algorithm is achievable.

Figure 1: Bernard Galler

§3 Main Operations

The Disjoint Set Union (DSU) data structure provides the following capabilities. We are given several elements, each of which is a separate set. A DSU will have an operation to combine any two sets, and it will be able to tell in which set a specific element is. The classical version also introduces a third operation, allowing it to create a set from a new element.

Thus, the basic interface of this data structure consists of only three operations:

- `make_set(v)` - creates a new set consisting of the new element v.
- `union_sets(a, b)` - merges the two specified sets (the set in which the element a is located, and the set in which the element b is located).
- `find_set(v)` - returns the representative (also called leader) of the set that contains the element v. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after `union_sets` calls). This representative can be used to check if two elements are part of the same set or not. a and b are exactly in the same set if `find_set(a) == find_set(b)`. Otherwise, they are in different sets.

As described in more detail later, the data structure allows you to do each of these operations in almost $O(1)$ time on average.

§3.1 Alternative Structure of DSU

Also in one of the subsections, an alternative structure of a DSU is explained, which achieves a slower average complexity of $O(\log n)$, but can be more powerful than the regular DSU structure.

§3.2 Detailed Explanation of Operations

The `make_set(v)` operation initializes a new set with the element v as its only member. This operation is usually performed at the beginning when each element is its own set.

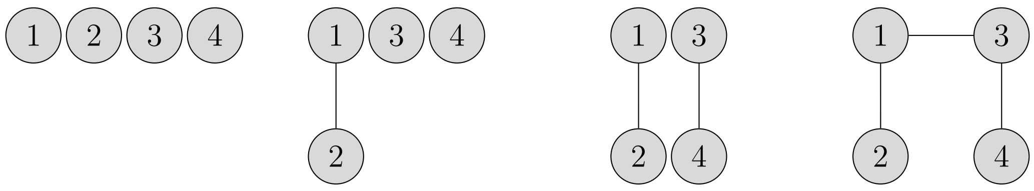
The `union_sets(a, b)` operation merges the sets containing elements a and b. This is done by connecting the representatives (leaders) of these sets. If the sets are already connected, this operation does nothing. Typically, this operation includes optimizations such as union by rank or size, where the smaller set is attached to the larger set to keep the tree shallow.

The `find_set(v)` operation finds the representative of the set containing v. This operation includes path compression as an optimization, which flattens the structure of the tree whenever `find_set` is called, ensuring that future operations are faster.

§4 Efficient Data Representation

To build an efficient data structure, we will store the sets in the form of **trees**. Each tree will correspond to one set, and the root of the tree will be the representative or leader of the set.

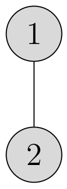
In the following figure, we can see the representation of such trees at different stages of the union operations.



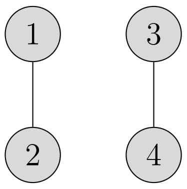
Initially, every element starts as a single set, so each vertex is its own tree. For example, we begin with the elements $\{1, 2, 3, 4\}$ each in separate sets.

Next, we perform a series of union operations:

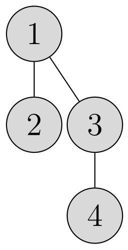
- **Step 1**: Combine the set containing element 1 and the set containing element 2. The tree now looks like this:



- **Step 2**: Combine the set containing element 3 and the set containing element 4. The trees now look like this:



- **Step 3**: Finally, combine the set containing element 1 and the set containing element 3. The tree now looks like this:



For the implementation, we maintain an array `parent` that stores a reference to the immediate ancestor of each element in the tree. If an element is the root of a tree, its parent reference points to itself. For example:

$$\text{parent} = \begin{bmatrix} 1 & 1 & 1 & 3 \end{bmatrix}$$

This array helps efficiently perform the `find_set` operation using path compression, and the `union_sets` operation can be optimized using union by rank or size.

§5 Naive Implementation

We can start by implementing the basic version of the Disjoint Set Union (DSU) data structure. Initially, this implementation will be quite inefficient, but with subsequent optimizations, we will be able to achieve nearly constant time complexity for each operation.

In our initial implementation, all the information about the sets of elements is maintained in an array called `parent`. This array will store the parent or representative of each element in the set.

To create a new set, which is denoted by the operation `make_set(v)`, we need to initialize a new tree with the vertex v as its root. This means that v will be its own ancestor, indicating that v is the only member of its set at this point.

When it comes to combining two sets, we use the `union_sets(a,b)` operation. The process begins by finding the representative (or root) of the set containing element a and the representative of the set containing element b. If both elements are already in the same set, that is, if their representatives are the same, then no further action is needed as the sets are already merged. However, if the representatives differ, we merge the two sets by making one representative the parent of the other. This effectively combines the two trees into a single tree.

The `find_set(v)` operation is used to locate the representative of the set containing the element v. This is achieved by traversing the ancestry of v until we reach the root of the tree. The root is the vertex such that the reference to its ancestor points to itself. This operation can be implemented recursively, where we repeatedly call `find_set` on the parent of the current vertex until we find the root.

```cpp
#include <iostream>

using namespace std;

const int N = 2e5;
int parent[N];

void make_set(int v) {
    parent[v] = v;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    return find_set(parent[v]);
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b)
        parent[b] = a;
    else
        return;
}
```

§6 Optimization

The naive implementation is inefficient. It is easy to construct an example, so that the trees degenerate into long chains. In that case each call `find_set(v)` can take $O(n)$ time.

This is far away from the complexity that we want to have (nearly constant time). Therefore we will consider two optimizations that will allow to significantly accelerate the work.

§6.1 Path Compression optimization

This optimization is designed for speeding up the `find_set(v)`.

If we call `find_set(v)` for some vertex v, we actually find the representative p for all vertices that we visit on the path between v and the actual representative p. The trick is to make the paths for all those nodes shorter, by setting the parent of each visited vertex directly to p in each recursive call when finding the representation of the node v. In this way, we only take $O(n)$ in the first iteration, for the later querying, we don't have to recursively call for a long chain any more. At the end, for m queries of finding the representative element of a set, the total time complexity will be $O(m)$.

`Visualization of path compression:`

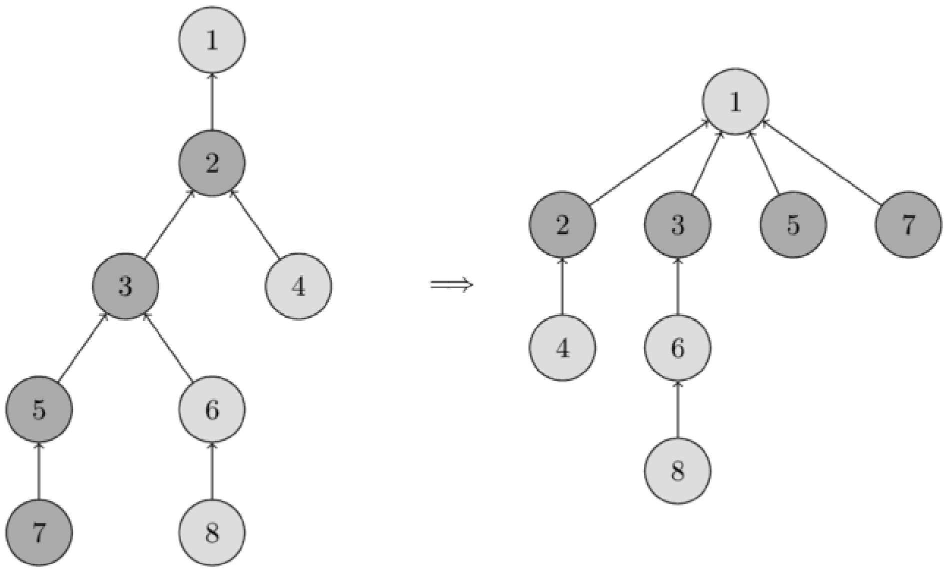


Figure 2: Path Compression

In the following image, you can observe the operation of the `find_set` function in action. The left side of the image depicts a tree structure before the operation is performed. This tree consists of several nodes, with node 7 being the node of interest. On the right side of the image, you can see the resulting tree after the `find_set(7)` function has been executed. This operation involves path compression, which effectively shortens the paths to the root for all visited nodes. Specifically, the paths for nodes 7, 5, 3, and 2 are optimized during this process, reducing the depth of the tree and improving future access times for these nodes. This visualization clearly illustrates the impact of the `find_set` function on the tree structure by showing the state before and after path compression.

The new implementation of `find_set` is as follow:

```
int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}
```

Another shorter way of implementation:

```
int find_set(int v) {
    return v == parent[v] ? v : parent[v] = find_set(parent[v]);
}
```

The simple implementation does what was intended: first find the representative of the set (root vertex), and then in the process of stack unwinding the visited nodes are attached directly to the representative.

This simple modification of the operation already achieves the time complexity $O(\log(n)$ per call on average. There is another optimization, which improve our algorithm even faster, which is call union from small to large.

§6.2 Union by size / rank

To enhance the union operation in the Disjoint Set Union (DSU) data structure, we will revise the strategy for determining which tree is attached to the other during the union process. In the naive implementation, the second tree is always attached to the first tree, potentially resulting in chains of length $O(n)$. This optimization aims to eliminate such inefficiencies by judiciously selecting the tree to be attached.
There are various heuristics that can be employed to optimize this operation. The two most prevalent approaches are:

- `Size-Based Ranking:` This approach uses the size of the trees as their rank. When performing a union, the tree with the smaller size is attached to the tree with the larger size. This helps in maintaining a balanced tree structure, thereby reducing the overall height of the tree.

- `Depth-Based Ranking:` In this approach, the depth of the trees (more precisely, the upper bound on the tree depth, which decreases due to path compression) is used as the rank. The tree with the smaller depth is attached to the tree with the larger depth. This technique further reduces the depth of the trees, ensuring that the DSU operations remain efficient.

Both methods share a common principle: the tree with the lower rank is attached to the tree with the higher rank. This strategy ensures that the resulting tree remains balanced, thereby optimizing the union operation and enhancing the overall performance of the DSU data structure.

By employing these ranking strategies, we can significantly enhance the efficiency of the DSU structure, particularly in scenarios involving large datasets with numerous union and find operations. These optimizations not only reduce the time complexity from a potentially linear scale to nearly constant time for each operation but also contribute to a more balanced and scalable system. This makes the DSU structure exceptionally valuable in a wide range of applications, from network connectivity to image processing and beyond.

Implementation of union by size:

```
#include <iostream>
using namespace std;

int find_set(int v) {
    return v == parent[v] ? v : parent[v] = find_set(parent[v]);
}

void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
    return;
}
```

Implementation of union by rank:

```
#include <iostream>
using namespace std;

int find_set(int v) {
    return v == parent[v] ? v : parent[v] = find_set(parent[v]);
}

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

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

§7 Time complexity

§7.1 Time complexity

- As mentioned before, if we combine both optimizations - path compression with union by size/rank - we will reach nearly constant time queries. It turns out, that the final amortized time complexity is $O(\alpha(n))$, where $\alpha(n)$ is the inverse **Ackermann function**, which grows very slowly. In fact, it grows so slowly, that it doesn't exceed 4 for all reasonable n (approximately $n < 10^{600}$).
- Amortized complexity is the total time per operation, evaluated over a sequence of multiple operations. The idea is to guarantee the total time of the entire sequence, while allowing single operations to be much slower than the amortized time. E.g., in our case a single call might take $O(\log n)$ in the worst case, but if we do m such calls back to back, we will end up with an average time of $O(\alpha(n))$.
- We will also not present a proof for this time complexity, since it is quite long and complicated.
- Also, it's worth mentioning that DSU with union by size/rank, but without path compression, works in $O(\log n)$ time per query.

§7.2 Inverse Ackermann Function

- The inverse Ackermann function is the inverse of the Ackermann function, which is a mathematical function defined as follows:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0, \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0, \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

- The inverse Ackermann function is the function that returns the pair of integers (m, n) that satisfies the equation $A(m, n) = x$, where x is a given value. This function is not defined for all values of x, as there are some values of x that do not correspond to any pair of integers (m, n).

Example of Inverse Ackermann Function:

- Set $m = 0$ and $n = 5$.
- Evaluate the function $A(0, 5)$ and check if it equals 5. If it does, then the pair of integers $(0, 5)$ is the solution to the equation. If it does not, then go to the next step.
- Set $m = 1$ and $n = 4$.
- Evaluate the function $A(1, 4)$ and check if it equals 5. If it does, then the pair of integers $(1, 4)$ is the solution to the equation. If it does not, then go to the next step.
- Repeat this process until you find a pair of integers (m, n) that satisfies the equation $A(m, n) = 5$, or until you determine that there is no solution to the equation.

- This process can be used to find the inverse Ackermann function for any given value of x. The inverse Ackermann function is not a well-defined function, as it is not defined for all values of x, and it is not unique, as there may be multiple pairs of integers (m, n) that satisfy the equation $A(m, n) = x$.

How does it grow?

- The Ackermann function grows very rapidly, which means that it increases in value very quickly as the input values m and n increase. This rapid growth is due to the recursive nature of the function, which causes it to call itself multiple times for large values of m and n.

For example,

- Evaluation of the Ackermann function for $m = 4$ and $n = 1$, will give the result $A(4, 1) = 65533$. This is a very large number, and it is much larger than the result you would get if you evaluated the function for $m = 3$ and $n = 1$, which is $A(3, 1) = 13$.
- This rapid growth of the Ackermann function makes it difficult to evaluate the function for large values of m and n, and it also makes it difficult to find the inverse Ackermann function for large values of x. However, despite these limitations, the Ackermann function and the inverse Ackermann function are interesting mathematical objects that have been studied by mathematicians and computer scientists.

Algorithm with $O(\alpha(n))$ time complexity, where $\alpha(n)$ is the inverse Ackermann function:

An algorithm with $O(\alpha(n))$ time complexity, where $\alpha(n)$ is the inverse Ackermann function, is an algorithm that has a time complexity that is bounded by the inverse Ackermann function. This means that the running time of the algorithm is always less than or equal to the value of the inverse Ackermann function for the input size n.

The inverse Ackermann function grows very slowly, which means that it is much smaller than other commonly used time complexity functions such as $O(1)$, $O(\log n)$, and $O(n)$. This makes algorithms with $O(\alpha(n))$ time complexity very efficient, as they can solve large problems in a relatively short amount of time.

§8 Applications

§8.1 Connected components in a graph

This is one of the obvious applications of DSU.

Formally the problem is defined in the following way: Initially we have an empty graph. We have to add vertices and undirected edges, and answer queries of the form (a, b) - "are the vertices a and b in the same connected component of the graph?"

Here we can directly apply the data structure, and get a solution that handles an addition of a vertex or an edge and a query in nearly constant time on average.

This application is quite important, because nearly the same problem appears in Kruskal's algorithm for finding a minimum spanning tree. Using DSU we can improve the $\mathcal{O}(m \log n + n^2)$ complexity to $\mathcal{O}(m \log n)$.

§8.2 Search for connected components in an image

One of the applications of DSU is the following task: there is an image of $n \times m$ pixels. Originally all are white, but then a few black pixels are drawn. You want to determine the size of each white connected component in the final image.

For the solution we simply iterate over all white pixels in the image, for each cell iterate over its four neighbors, and if the neighbor is white call `union_sets`. Thus we will have a DSU with nm nodes corresponding to image pixels. The resulting trees in the DSU are the desired connected components.

The problem can also be solved by DFS or BFS, but the method described here has an advantage: it can process the matrix row by row (i.e. to process a row we only need the previous and the current row, and only need a DSU built for the elements of one row) in $\mathcal{O}(\min(n, m))$ memory.

§8.3 Compress jumps along a segment / 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 $O(\alpha(n))$.

§8.4 Support distances up to 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.

```cpp
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]++;
    }
}
```

§8.5 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 consists 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 one 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];
}
```

§8.6 Offline LCA (lowest common ancestor in a tree) in $O(\alpha(n))$ on average

§8.6.1 Algorithm for Offline LCA Queries

We have a tree G with n nodes and we have m queries of the form (u, v). For each query (u, v) we want to find the lowest common ancestor of the vertices u and v, i.e. the node that is an ancestor of both u and v and has the greatest depth in the tree. The node v is also an ancestor of v, so the LCA can also be one of the two nodes.
In this article we will solve the problem off-line, i.e. we assume that all queries are known in advance, and we therefore answer the queries in any order we like. The following algorithm allows to answer all m queries in $O(n + m)$ total time, i.e. for sufficiently large m in $O(1)$ for each query.

§8.6.2 Algorithm

The algorithm is named after Robert Tarjan, who discovered it in 1979 and also made many other contributions to the Disjoint Set Union data structure, which will be heavily used in this algorithm.

The algorithm answers all queries with one DFS traversal of the tree. Namely a query (u, v) is answered at node u, if node v has already been visited previously, or vice versa.

So let's assume we are currently at node v, we have already made recursive DFS calls, and also already visited the second node u from the query (u, v). Let's learn how to find the LCA of these two nodes.

Note that LCA(u, v) is either the node v or one of its ancestors. So we need to find the lowest node among the ancestors of v (including v), for which the node u is a descendant. Also note that for a fixed v the visited nodes of the tree split into a set of disjoint sets. Each ancestor p of node v has its own set containing this node and all subtrees with roots in those of its children who are not part of the path from v to the root of the tree. The set which contains the node u determines the LCA(u, v): the LCA is the representative of the set, namely the node on lies on the path between v and the root of the tree.

We only need to learn to efficiently maintain all these sets. For this purpose we apply the data structure DSU. To be able to apply Union by rank, we store the real representative (the value on the path between v and the root of the tree) of each set in the array `ancestor`.

Let's discuss the implementation of the DFS. Let's assume we are currently visiting the node v. We place the node in a new set in the DSU, `ancestor[v] = v`. As usual we process all children of v. For this we must first recursively call DFS from that node, and then add this node with all its subtree to the set of v. This can be done with the function `union_sets` and the following assignment `ancestor[find_set(v)] = v` (this is necessary, because `union_sets` might change the representative of the set).

Finally after processing all children we can answer all queries of the form (u, v) for which u has been already visited. The answer to the query, i.e. the LCA of u and v, will be the node `ancestor[find_set(u)]`. It is easy to see that a query will only be answered once.

Let's us determine the time complexity of this algorithm. Firstly we have $O(n)$ because of the DFS. Secondly we have the function calls of `union_sets` which happen n times, resulting also in $O(n)$. And thirdly we have the calls of `find_set` for every query, which gives $O(m)$. So in total the time complexity is $O(n+m)$, which means that for sufficiently large m this corresponds to $O(1)$ for answering one query.

§8.6.3 Implementation

Here is an implementation of this algorithm. The implementation of DSU has not been included, as it can be used without any modifications.

```cpp
vector<vector<int>> adj;
vector<vector<int>> queries;
vector<int> ancestor;
vector<bool> visited;

void dfs(int v)
{
    visited[v] = true;
    ancestor[v] = v;
    for (int u : adj[v]) {
        if (!visited[u]) {
            dfs(u);
            union_sets(v, u);
            ancestor[find_set(v)] = v;
        }
    }
    for (int other_node : queries[v]) {
        if (visited[other_node])
            cout << "LCA of " << v << " and " << other_node
                 << " is " << ancestor[find_set(other_node)] << ".\n";
    }
}

void compute_LCAs() {
    // initialize n, adj and DSU
    // for (each query (u, v)) {
    //   queries[u].push_back(v);
    //   queries[v].push_back(u);
    // }
    ancestor.resize(n);
    visited.assign(n, false);
    dfs(0);
}
```

§8.7 Storing the DSU explicitly in a set list / Applications of this idea when merging various data structures

One of the alternative ways of storing the DSU is the preservation of each set in the form of an explicitly stored list of its elements. At the same time each element also stores the reference to the representative of his set.

At first glance this looks like an inefficient data structure: by combining two sets we will have to add one list to the end of another and have to update the leadership in all elements of one of the lists.

However it turns out, the use of a *weighting heuristic* (similar to Union by size) can significantly reduce the asymptotic complexity: $O(m + n \log n)$ to perform m queries on the n elements.

Under weighting heuristic we mean, that we will always add the smaller of the two sets to the bigger set. Adding one set to another is easy to implement in `union_sets` and will take time proportional to the size of the added set. And the search for the leader in `find_set` will take $O(1)$ with this method of storing.

Let us prove the *time complexity* $O(m + n \log n)$ for the execution of m queries. We will fix an arbitrary element x and count how often it was touched in the merge operation `union_sets`. When the element x gets touched the first time, the size of the new set will be at least 2. When it gets touched the second time, the resulting set will have size of at least 4, because the smaller set gets added to the bigger one. And so on. This means, that x can only be moved in at most $\log n$ merge operations. Thus the sum over all vertices gives $O(n \log n)$ plus $O(1)$ for each request.

Here is an implementation:

```
vector<int> lst[MAXN];
int parent[MAXN];

void make_set(int v) {
    lst[v] = vector<int>(1, v);
    parent[v] = v;
}

int find_set(int v) {
    return parent[v];
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (lst[a].size() < lst[b].size())
            swap(a, b);
        while (!lst[b].empty()) {
            int v = lst[b].back();
            lst[b].pop_back();
            parent[v] = a;
            lst[a].push_back(v);
        }
    }
}
```

This idea of adding the smaller part to a bigger part can also be used in a lot of solutions that have nothing to do with DSU.

For example consider the following problem: we are given a tree, each leaf has a number assigned (same number can appear multiple times on different leaves). We want to compute the number of different numbers in the subtree for every node of the tree.

Applying to this task the same idea it is possible to obtain this solution: we can implement a DFS, which will return a pointer to a set of integers - the list of numbers in that subtree. Then to get the answer for the current node (unless of course it is a leaf), we call DFS for all children of that node, and merge all the received sets together. The size of the resulting set will be the answer for the current node. To efficiently combine multiple sets we just apply the above-described recipe: we merge the sets by simply adding smaller ones to larger. In the end we get a $O(n \log^2 n)$ solution, because one number will only added to a set at most $O(\log n)$ times.

§9 Conclusion

In this document, we explored the Disjoint Set Union (DSU) data structure, which is an essential tool in computer science for managing dynamic connectivity problems. We began by introducing the fundamental operations of DSU, including `make_set`, `union_sets`, and `find_set`. These operations enable us to efficiently merge sets and determine the connected components in a graph.

We then delved into the history of DSU, highlighting key contributions by researchers such as Bernard Galler, Michael Fischer, and Robert Tarjan, who significantly improved the efficiency of the data structure. The historical context provided a deeper understanding of how the DSU has evolved over time to become a highly optimized and powerful tool in modern algorithms.

Following that, we discussed the main operations of DSU in detail, including an alternative structure that offers a trade-off between performance and flexibility. We also provided a naive implementation of DSU, which, while simple, laid the foundation for more advanced optimizations.

The document then introduced two critical optimizations: path compression and union by rank. These techniques significantly improve the efficiency of the DSU, bringing the time complexity of operations close to constant time. Path compression flattens the structure of the trees, ensuring that future `find_set` operations are faster, while union by rank helps maintain a balanced tree structure, minimizing the height of the trees.

To visualize the impact of these optimizations, we provided diagrams and examples that illustrated how the DSU structure evolves over a series of operations. These visual aids helped to clarify the mechanics of the data structure and demonstrated the effectiveness of the optimizations.

In conclusion, the Disjoint Set Union data structure is an indispensable tool in the realm of algorithms, particularly for problems involving dynamic connectivity in graphs. Its efficiency and simplicity make it a go-to solution for a wide range of applications, from Kruskal's algorithm for finding Minimum Spanning Trees to network connectivity and clustering analysis. By leveraging the optimizations discussed in this document, one can achieve nearly constant time performance, making DSU a powerful and practical choice for solving complex problems in computer science.

As we continue to advance in the field of algorithms and data structures, the principles of DSU will remain relevant, serving as a foundation for new innovations and optimizations. Whether you are tackling competitive programming challenges or working on real-world applications, understanding and implementing DSU will equip you with a versatile tool for efficient problem-solving.

§10 References

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

3. Galler, B. A., and Fischer, M. J. (1964). An improved equivalence algorithm. *Communications of the ACM*, 7(5), 301-303.

4. Tarjan, R. E. (1975). Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM (JACM)*, 22(2), 215-225.

5. Tarjan, R. E. (1979). Applications of Path Compression on Balanced Trees. *Journal of the ACM (JACM)*, 26(4), 690-715.

6. Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1: Fundamental Algorithms* (3rd ed.). Addison-Wesley.

7. Mehlhorn, K. (1984). *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer-Verlag.

8. Galil, Z., and Italiano, G. F. (1991). Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3), 319-344.

9. Sedgewick, R., and Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

10. Westbrook, J. R. (1992). Fast Union-Find. In *Handbook of Data Structures and Applications* (pp. 149-169). CRC Press.

11. Skiena, S. S. (2008). *The Algorithm Design Manual* (2nd ed.). Springer.

12. D.D. Padma Priya, G. Shobhalatha, R. Bhuvana Vijaya. The theory of semi-groups in finite state automata. *Information and Control*, 17(2), 196-201.

13. Hopcroft, J. E., and Ullman, J. D. (1973). Set merging algorithms. *SIAM Journal on Computing*, 2(4), 294-303.

14. Md. Mostofa Ali Patwary1, Jean Blair2, and Fredrik Manne. Union-Find Algorithms in Multilevel Memory. *Theoretical Computer Science*, 20(1), 97-105.

15. Patwary, M. M. A., Blair, J. R. S., and Manne, F. (2010). Experiments on union-find algorithms for the disjoint-set data structure. *Proceedings of the Meeting on Algorithm Engineering & Experiments (ALENEX)*, 19-27.

16. Baswana, S., Gupta, M., and Sen, S. (2011). Union-Find with Constant Time Deletions. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '11)*, 144-155.

17. Pfenning, F., and Lee, P. (1991). *Proof-carrying code*. Carnegie Mellon University.