Finding Bridges Online

Table of Contents

We are given an undirected graph. A bridge is an edge whose removal makes the graph disconnected (or, more precisely, increases the number of connected components). Our task is to find all the bridges in the given graph.

Informally this task can be put as follows: we have to find all the "important" roads on the given road map, i.e. such roads that the removal of any of them will lead to some cities being unreachable from others.

There is already the article Finding Bridges in $O(N + M)$ which solves this task with a Depth First Search traversal. This algorithm will be much more complicated, but it has one big advantage: the algorithm described in this article works online, which means that the input graph doesn't have to be not known in advance. The edges are added once at a time, and after each addition the algorithm recounts all the bridges in the current graph. In other words the algorithm is designed to work efficiently on a dynamic, changing graph.

More rigorously the statement of the problem is as follows: Initially the graph is empty and consists of n

vertices. Then we receive pairs of vertices (a, b), which denote an edge added to the graph. After each received edge, i.e. after adding each edge, output the current number of bridges in the graph.

It is also possible to maintain a list of all bridges as well as explicitly support the 2-edge-connected components.

The algorithm described below works in $O(n \log n + m)$ time, where m is the number of edges. The algorithm is based on the data structure Disjoint Set Union. However the implementation in this article takes $O(n \log n + m \log n)$ time, because it uses the simplified version of the DSU without Union by Rank.

Algorithm

First let's define a k-edge-connected component: it is a connected component that remains connected whenever you remove fewer than k edges.

It is very easy to see, that the bridges partition the graph into 2-edge-connected components. If we compress each of those 2-edge-connected components into vertices and only leave the bridges as edges in the compressed graph, then we obtain an acyclic graph, i.e. a forest.

The algorithm described below maintains this forest explicitly as well as the 2-edge-connected components.

It is clear that initially, when the graph is empty, it contains n 2-edge-connected components, which by themselves are not connect.

When adding the next edge (a, b) there can occur three situations:

- Both vertices a and b are in the same 2-edge-connected component - then this edge is not a bridge, and does not change anything in the forest structure, so we can just skip this edge.

  Thus, in this case the number of bridges does not change.

- The vertices a and b are in completely different connected components, i.e. each one is part of a different tree. In this case, the edge (a, b) becomes a new bridge, and these two trees are combined into one (and all the old bridges remain).

  Thus, in this case the number of bridges increases by one.

- The vertices a and b are in one connected component, but in different 2-edge-connected components. In this case, this edge forms a cycle along with some of the old bridges. All these bridges end being bridges, and the resulting cycle must be compressed into a new 2-edge-connected component.

  Thus, in this case the number of bridges decreases by two or more.

Consequently the whole task is reduced to the effective implementation of all these operations over the forest of 2-edge-connected components.

Data Structures for storing the forest

The only data structure that we need is Disjoint Set Union. In fact we will make two copies of this structure: one will be to maintain the connected components, the other to maintain the 2-edge-connected components. And in addition we store the structure of the trees in the forest of 2-edge-connected components via pointers: Each 2-edge-connected component will store the index `par[]` of its ancestor in the tree.

We will now consistently disassemble every operation that we need to learn to implement:

- Check whether the two vertices lie in the same connected / 2-edge-connected component. It is done with the usual DSU algorithm, we just find and compare the representatives of the DSUs.

- Joining two trees for some edge (a, b). Since it could turn out that neither the vertex a nor the vertex b are the roots of their trees, the only way to connect these two trees is to re-root one of them. For example you can re-root the tree of vertex a, and then attach it to another tree by setting the ancestor of a to b.

  However the question about the effectiveness of the re-rooting operation arises: in order to re-root the tree with the root r to the vertex v, it is necessary to necessary to visit all vertices on the path between v and r and redirect the pointers `par[]` in the opposite direction, and also change the references to the ancestors in the DSU that is responsible for the connected components.

  Thus, the cost of re-rooting is $O(h)$, where h is the height of the tree. You can make an even worse estimate by saying that the cost is $O(\text{size})$ where size

is the number of vertices in the tree. The final complexity will not differ.

We now apply a standard technique: we re-root the tree that contains fewer vertices. Then it is intuitively clear that the worst case is when two trees of approximately equal sizes are combined, but then the result is a tree of twice the size. This does not allow this situation to happen many times.

In general the total cost can be written in the form of a recurrence:

$$T(n) = \max_{k=1 \ldots n-1} \{T(k) + T(n-k) + O(\min(k, n-k$$

$T(n)$ is the number of operations necessary to obtain a tree with n vertices by means of re-rooting and unifying trees. A tree of size n can be created by combining two smaller trees of size k and $n-k$. This recurrence is has the solution $T(n) = O(n \log n)$.

Thus, the total time spent on all re-rooting operations will be $O(n \log n)$ if we always re-root the smaller of the two trees.

We will have to maintain the size of each connected component, but the data structure DSU makes this possible without difficulty.

- Searching for the cycle formed by adding a new edge (a, b). Since a and b are already connected in the tree we need to find the Lowest Common Ancestor of the vertices a and b. The cycle will consist of the paths from b to the LCA, from the LCA to b and the edge a to b.

After finding the cycle we compress all vertices of the detected cycle into one vertex. This means that we already have a complexity proportional to the cycle length, which means that we also can use any LCA algorithm proportional to the length, and don't have to use any fast one.

Since all information about the structure of the tree is available is the ancestor array `par[]`, the only reasonable LCA algorithm is the following: mark the vertices a and b as visited, then we go to their ancestors `par[a]` and `par[b]` and mark them, then advance to their ancestors and so on, until we reach an already marked vertex. This vertex is the LCA that we are looking for, and we can find the vertices on the cycle by traversing the path from a and b to the LCA again.

It is obvious that the complexity of this algorithm is proportional to the length of the desired cycle.

- Compression of the cycle by adding a new edge (a, b) in a tree.

  We need to create a new 2-edge-connected component, which will consist of all vertices of the detected cycle (also the detected cycle itself could consist of some 2-edge-connected components, but this does not change anything). In addition it is necessary to compress them in such a way that the structure of the tree is not disturbed, and all pointers `par[]` and two DSUs are still correct.

  The easiest way to achieve this is to compress all the vertices of the cycle to their LCA. In fact the LCA is the highest of the vertices, i.e. its ancestor pointer `par[]`

remains unchanged. For all the other vertices of the loop the ancestors do not need to be updated, since these vertices simply cease to exists. But in the DSU of the 2-edge-connected components all these vertices will simply point to the LCA.

We will implement the DSU of the 2-edge-connected components without the Union by rank optimization, therefore we will get the complexity $O(\log n)$ on average per query. To achieve the complexity $O(1)$ on average per query, we need to combine the vertices of the cycle according to Union by rank, and then assign `par[]` accordingly.

Implementation

Here is the final implementation of the whole algorithm.

As mentioned before, for the sake of simplicity the DSU of the 2-edge-connected components is written without Union by rank, therefore the resulting complexity will be $O(\log n)$ on average.

Also in this implementation the bridges themselves are not stored, only their count `bridges`. However it will not be difficult to create a `set` of all bridges.

Initially you call the function `init()`, which initializes the two DSUs (creating a separate set for each vertex, and setting the size equal to one), and sets the ancestors `par`.

The main function is `add_edge(a, b)`, which processes and adds a new edge.

```
vector<int> par, dsu_2ecc, dsu_cc, dsu_cc_size
int bridges;
int lca_iteration;
vector<int> last_visit;

void init(int n) {
    par.resize(n);
    dsu_2ecc.resize(n);
    dsu_cc.resize(n);
    dsu_cc_size.resize(n);
    lca_iteration = 0;
    last_visit.assign(n, 0);
    for (int i=0; i<n; ++i) {
        dsu_2ecc[i] = i;
        dsu_cc[i] = i;
        dsu_cc_size[i] = 1;
        par[i] = -1;
    }
    bridges = 0;
}

int find_2ecc(int v) {
    if (v == -1)
        return -1;
    return dsu_2ecc[v] == v ? v : dsu_2ecc[v]
}

int find_cc(int v) {
    v = find_2ecc(v);
    return dsu_cc[v] == v ? v : dsu_cc[v] = fi
}

void make_root(int v) {
    v = find_2ecc(v);
    int root = v;
    int child = -1;
```

```
    while (v != -1) {
        int p = find_2ecc(par[v]);
        par[v] = child;
        dsu_cc[v] = root;
        child = v;
        v = p;
    }
    dsu_cc_size[root] = dsu_cc_size[child];
}

void merge_path (int a, int b) {
    ++lca_iteration;
    vector<int> path_a, path_b;
    int lca = -1;
    while (lca == -1) {
        if (a != -1) {
            a = find_2ecc(a);
            path_a.push_back(a);
            if (last_visit[a] == lca_iteration
                lca = a;
            last_visit[a] = lca_iteration;
            a = par[a];
        }
        if (b != -1) {
            path_b.push_back(b);
            b = find_2ecc(b);
            if (last_visit[b] == lca_iteration
                lca = b;
            last_visit[b] = lca_iteration;
            b = par[b];
        }

    }

    for (int v : path_a) {
        dsu_2ecc[v] = lca;
        if (v == lca)
```

```
            break;
        --bridges;
    }
    for (int v : path_b) {
        dsu_2ecc[v] = lca;
        if (v == lca)
            break;
        --bridges;
    }
}

void add_edge(int a, int b) {
    a = find_2ecc(a);
    b = find_2ecc(b);
    if (a == b)
        return;

    int ca = find_cc(a);
    int cb = find_cc(b);

    if (ca != cb) {
        ++bridges;
        if (dsu_cc_size[ca] > dsu_cc_size[cb])
            swap(a, b);
            swap(ca, cb);
        }
        make_root(a);
        par[a] = dsu_cc[a] = b;
        dsu_cc_size[cb] += dsu_cc_size[a];
    } else {
        merge_path(a, b);
    }
}
```

The DSU for the 2-edge-connected components is stored in the vector `dsu_2ecc`, and the function returning

the representative is `find_2ecc(v)`. This function is used many times in the rest of the code, since after the compression of several vertices into one all these vertices cease to exist, and instead only the leader has the correct ancestor `par` in the forest of 2-edge-connected components.

The DSU for the connected components is stored in the vector `dsu_cc`, and there is also an additional vector `dsu_cc_size` to store the component sizes. The function `find_cc(v)` returns the leader of the connectivity component (which is actually the root of the tree).

The re-rooting of a tree `make_root(v)` works as descibed above: if traverses from the vertex v via the ancestors to the root vertex, each time redirecting the ancestor `par` in the opposite direction. The link to the representative of the connected component `dsu_cc` is also updated, so that it points to the new root vertex. After re-rooting we have to assign the new root the correct size of the connected component. Also we have to be careful that we call `find_2ecc()` to get the representatives of the 2-edge-connected component, rather than some other vertex that have already been compressed.

The cycle finding and compression function `merge_path(a, b)` is also implemented as descibed above. It searches for the LCA of a and b be rising these nodes in parallel, until we meet a vertex for the second time. For efficiency purposes we choose a unique identifier for each LCA finding call, and mark the traversed vertices with it. This works in $O(1)$, while other approaches like using set perform worse. The passed paths are stored in the vectors `path_a` and `path_b`, and we use them to walk through them a second time up to

the LCA, thereby obtaining all vertices of the cycle. All the vertices of the cycle get compressed by attaching them to the LCA, hence the average complexity is $O(\log n)$ (since we don't use Union by rank). All the edges we pass have been bridges, so we subtract 1 for each edge in the cycle.

Finally the query function `add_edge(a, b)` determines the connected components in which the vertices a and b lie. If they lie in different connectivity components, then a smaller tree is re-rooted and then attached to the larger tree. Otherwise if the vertices a and b lie in one tree, but in different 2-edge-connected components, then the function `merge_path(a, b)` is called, which will detect the cycle and compress it into one 2-edge-connected component.