Posted: 5 Aug 2011 1:55
Edited: 5 Aug 2011 1:55

Search bridges online

Contents [hide]

Suppose we are given an undirected graph. A bridge is an edge whose removal makes the graph disconnected (or, rather, increases the number of connected components). You want to find all the bridges in a given graph.

Informally, this problem is formulated as follows: it is required to find on a given road map all the "important" roads, ie such way that removing either of them will lead to the disappearance of the path between any pair of cities.

Described herein is an algorithm **online** , that means the input graph is not known in advance, and the edges are added to it one by one, and after each addition of the algorithm recalculates all bridges in the current column. In other words, the algorithm is designed to work effectively in a dynamic, changing graph.

More rigorously, **formulation of the problem** is as follows. Initially empty graph consists of $n$vertices. Then receives requests, each of which - a pair of vertices (a, b)that represent an edge is added to the graph. Required after each request, ie after the addition of each edge, display the current number of bridges in the graph. (If desired, you can maintain a list of all edges, bridges, as well as explicit support rib doubly connected components.)

Described by the following algorithm works in time $O(n \log n + m)$where m- the number of requests. The algorithm is based on the data structure of "a system of disjoint sets" .

The reduced implementation of the algorithm, however, works in time $O(n \log n + m \log n)$, as it uses in one place a simplified version of the system of disjoint sets without rank heuristics.

Algorithm

It is known that the edges of bridges divide the vertices of the graph on the components, called a doubly-linked components of the rib. If each component edged doubly connected to squeeze in one vertex, and leave only the edges of bridges between these components, you get an acyclic graph, ie forest.

Algorithm described below support explicitly this **forest component edged doubly connected** .

It is clear that initially, when the graph is empty, it contains a $n$component edged doubly connected, not connected in any way with each other.

You add another rib (a, b)may be three situations:

- Both ends $a$and $b$are in the same component edged doubly connected - then this edge is not a bridge, and does not alter the structure of the forest, so just skip this edge.

  Thus, in this case the number of bridges is not changed.

- Tops $a$and $b$are in different connected components, ie, connect two trees. In this case, the edge (a, b)becomes the new bridge, and the two trees are merged into one (and all the old bridges remain).

  Thus, in this case the number of bridges is incremented.

- Tops $a$and $b$are in the same connected component, but in the different components of the rib doubly connected. In this case, the rib forms a ring together with some of the old bridges. All of these bridges are no longer bridges, and the resulting cycle must be combined into a new component edged doubly connected.

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

Consequently, the whole problem is reduced to the effective implementation of these operations on the forest component.

Data structures for timber

All we need from the data structures - a system of disjoint sets . In fact, we need to make two copies of this structure: one is to maintain the **connected components** , the other - in order to maintain **a doubly-linked component of the rib** .

In addition, the storage structure of the trees in the forest component of the doubly linked to each node will store a pointer $\mathrm{par}[]$to its parent in the tree.

We now consistently disassemble each operation, which we must learn to realize:

- **Check whether the two are listed in the top of the same connected component / doubly connected** . Is the usual request to the structure of the "system of disjoint sets."
- **Connecting two trees into one** by some edge (a, b). Because it could turn out that no vertex $a$or vertex $b$are not roots of their trees, the only way to connect these two trees - **perepodvesit** one of them. For example, you can perepodvesit single tree on the vertex a, and then attach it to another tree, making the top of $a$the child to b.

  However, there is a question about the effectiveness of the operation perepodveshivaniya: to perepodvesit tree rooted at $r$the vertex v, you have to pass on the way from va $r$redirecting pointers $\mathrm{par}[]$in the opposite direction, as well as changing the reference to the parent in the system of disjoint sets responsible for the connected components.

  Thus, the cost of the operation perepodveshivaniya is $O(h)$where h- the height of the tree. We can estimate it even higher, saying that this is the value $O(\mathrm{size})$where size- the number of vertices in the tree.

  We now apply a standard trick: we say that two trees **perepodveshivat will be the one in which fewer vertices** . Then it is intuitively clear that the worst case - when you combine two of the tree of approximately equal size, but then the result is a tree of twice the size, which does not allow such a situation to occur many times. Formally, this can be written as the recurrence relation:

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

  where by $T(n)$we denote the number of operations required to obtain a tree of n vertices using the operations perepodveshivaniya and merging trees. This well-known recurrence relation, and it has a solution $T(n) = O(n \log n)$.

  Thus, the total time spent on all perepodveshivaniya, will $O(n \log n)$, if we always

perepodveshivat lesser of two tree.

We'll have to keep the size of each connected component, but the data structure "system of disjoint sets" lets you do this easily.

- **Find a cycle** formed by adding a new edge $(a, b)$to some tree. Practically, this means that we need to find the lowest common ancestor (LCA) of vertices $a$and b.

  Note that then we sozhmëm all vertices in one cycle of the detected peak, so we want any of the search algorithm LCA, the running time of the order of its length.

  Since all the information about the structure of the tree, that we have - it links $par[]$to the ancestors, it seems the only possible next search algorithm LCA: mark tops a and $b$as the visited, then go to their ancestors $\mathrm{par}[a]$and $\mathrm{par}[b]$and mark them, then to their ancestors, and so on, until it happens that at least one of the two current peaks is already marked. This would mean that the current node - is the desired LCA, and it will be necessary to repeat again the way to it from the top $a$and from the top b- thus we find the desired cycle.

  It is obvious that this algorithm works in time order of the desired cycle, since each of the two pointers could not pass a distance greater than this length.

- **Compression cycle** formed by adding a new edge $(a, b)$to some tree.

  We need to create a new component edged doubly connected, which will consist of all the vertices of the detected cycle (of course, that the detected cycle itself could consist of some components of a doubly-linked, but it does not change anything). Furthermore, it is necessary to compress so that the tree structure has not been disrupted, and all pointers $\mathrm{par}[]$and two sets of disjoint were correct.

  The easiest way to do this - **to squeeze all the vertices of the cycle found in their LCA** . In fact, the top-LCA - is the highest peaks of the compressible, ie it par remains unchanged. For all other vertices compressible update also did not need to, because these peaks simply cease to exist - in the system of disjoint sets for the components of the doubly linked all these vertices will simply point to the top-LCA.

  But then it turns out that the system of disjoint sets for the components of a doubly-linked work without union by rank heuristic: if we always attach to the top of the cycle of LCA, then this heuristic has no place. In this case, the asymptotic behavior occurs $O(\log n)$because without heuristics to rank any operation with a system of disjoint sets of works that for a time.

  To achieve the asymptotic behavior of$O(1)$ one request is necessary to combine the top of the cycle according to the rank heuristic, and then assign a par new leader $\mathrm{par}[\mathrm{LCA}]$.

Implementation

We give here the final implementation of the whole algorithm.

For simplicity, a system of disjoint sets for the components of a doubly-linked written **without rank heuristics** , so the total amount to the asymptotic behavior of $O(\log n)$the query on average. (For information on how to reach the asymptotic behavior $O(1)$, described above in paragraph "compression cycle.")

Also in this embodiment are not kept themselves edges, bridges, and kept only their number - see. Variable $\mathrm{bridges}$. However, if you want to not be difficult to have setall of the bridges.

Initially, you must call the function $\text{init}()$that initializes the two systems of disjoint sets (assigning each vertex in a single set, and placing an amount equal to one), marks the ancestors par.

The main function - is $\text{add_edge}(a, b)$that processes a request to add a new edge.

Constant MAXNshould be set equal to the maximum possible number of vertices in the input graph.

More detailed explanations to this sale. Refer below.

```
const int MAXN = ...;

int n, bridges, par[MAXN], bl[MAXN], comp[MAXN], size[MAXN];


void init() {
        for (int i=0; i<n; ++i) {
                bl[i] = comp[i] = i;
                size[i] = 1;
                par[i] = -1;
        }
        bridges = 0;
}


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

int get_comp (int v) {
        v = get(v);
        return comp[v]==v ? v : comp[v]=get_comp(comp[v]);
}

void make_root (int v) {
        v = get(v);
        int root = v,
                child = -1;
        while (v != -1) {
                int p = get(par[v]);
                par[v] = child;
                comp[v] = root;
                child=v;  v=p;
        }
        size[root] = size[child];
}


int cu, u[MAXN];

void merge_path (int a, int b) {
        ++cu;

        vector<int> va, vb;
        int lca = -1;
        for(;;) {
                if (a != -1) {
```

```
                        a = get(a);
                        va.pb (a);

                        if (u[a] == cu) {
                                lca = a;
                                break;
                        }
                        u[a] = cu;

                        a = par[a];
                }

                if (b != -1) {
                        b = get(b);
                        vb.pb (b);

                        if (u[b] == cu) {
                                lca = b;
                                break;
                        }
                        u[b] = cu;

                        b = par[b];
                }
        }

        for (size_t i=0; i<va.size(); ++i) {
                bl[va[i]] = lca;
                if (va[i] == lca)  break;
                --bridges;
        }
        for (size_t i=0; i<vb.size(); ++i) {
                bl[vb[i]] = lca;
                if (vb[i] == lca)  break;
                --bridges;
        }
}


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

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

We comment on the code in more detail.

The system of disjoint sets for the components of a doubly-linked stored in the array $\mathrm{bl}[]$, and a function that returns a doubly connected components leader - is $\mathrm{get}(v)$. This function is used many times in the rest of the code, as it should be remembered that after the compression of several peaks in one all these vertices cease to exist, and instead there is only their leader, which are stored and the correct data (ancestor par, ancestor in the disjoint sets for connected components, etc.).

System of disjoint sets for connected components is stored in an array $\mathrm{comp}[]$, also have an additional array $\mathrm{size}[]$to store the size of the component. The function $\mathrm{get_comp}(v)$returns the leader of the connected components (which is actually the root of the tree).

Function perepodveshivaniya tree $\mathrm{make\_root}(v)$ works as described above: it goes from the top $v$to the ancestors to the root, each time redirecting ancestor parin the opposite direction (down towards the top v). Also updated pointer compin the system for disjoint sets of connected components, to point to the new root. After perepodveshivaniya the new root DIMENSIONS sizeconnected components. Note that in the implementation of every time we call the function $\mathrm{get}()$to access it to the leader strongly connected components, and not to some vertex, which may have already been compressed.

The detection and path compression $\mathrm{merge\_path}(a, b)$, as described above, looks LCA vertices $a$and $b$for which rises from them parallel upward until some node does not meet the second time. To be effective, passed peaks marked by the technique of "numerical used", that works for the $O(1)$application instead set. Completed path is stored in the vectors $\mathrm{va}$and  then to walk through it a second time to the LCA, thereby obtaining all the vertices of the cycle. All vertices of the cycle are compressed by attaching them to the LCA (here comes the asymptotic behavior $O(\log n)$, as if we do not use compression, the rank heuristic). Along the way, considered the number of edges traversed, which is the number of bridges in the detected cycle (this amount is subtracted from $\mathrm{bridges}$).

Finally, the **query function** $\mathrm{add\_edge}(a, b)$ defines the connected components, which are the vertices $a$and b, and if they lie in different connected components, the smaller tree perepodveshivaetsya for a new square and then attached to a large tree. Otherwise, if the vertices $a$and $b$lie in the same tree, but in the different components of a doubly connected, the function is called $\mathrm{merge\_path}(a, b)$, which detects a cycle, and compresses it into a single component of a doubly-linked.