Search bridges online

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 the road map given all the "important" roads, iesuch 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 precisely, **the problem statement** 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 in the graph to be added. Required after each request, i.e. after the addition of each edge, display the current number of bridges in the graph. (You may want to keep a list of all edges, bridges, as well as explicit support rib doubly connected components.)

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

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

Algorithm

It is known that edge-bridges divide vertices into components called costal doubly connected components. If each component costal doubly connected to squeeze into a single vertex, and leave only the edges of bridges between these components, you get an acyclic graph, ie forest.

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

It is clear that initially, when the graph is empty, it contains a $n$doubly connected component costal unrelated way among themselves.

When you add the next edge (a, b)may occur three situations:

- Both ends $a$and $b$are in the same component costal 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, is not changed by the bridges.
- 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 component, but in different components of the costal 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 costal 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 all these operations over the forest component.

Data structure for timber

All we need from the data structures - is 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 - to maintain the **doubly connected component of the rib** .

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

We now analyze each operation sequentially, which we must learn to realize:

- **Checking whether the two are listed in the top of the same component / doubly connected** . Typically, a query is made to the structure of the "system of disjoint sets."
- **Connect 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 one tree for the top 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 system in the ancestor of disjoint sets responsible for the connected components.

- Thus, the cost of the operation has perepodveshivaniya $O(h)$where h- the height of the tree. You can evaluate 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 method: we say that two trees **perepodveshivat will be one in which fewer vertices** . Then it is intuitively clear that the worst case - when you merge two trees of approximately equal size, but then the result is twice the size tree that does not allow such a situation to occur many times. Formally, this can be written as the recurrence relation:
- $T(n) = \max_{k=1\ldots n-1} \{ T(k) + T(n-k) + O(n) \},$
- where by $T(n)$we denote the number of operations necessary to obtain the tree of $n$vertices using the operations of union and perepodveshivaniya trees.This is a known recurrence, 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 have to maintain 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 a tree. Practically, this means that we need to find the lowest common ancestor (LCA) of vertices $a$and b.
- Note that then we compress all of the 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, seems the only possible next search algorithm LCA: mark tops $a$and $b$as 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 will mean that the current peak - is the desired LCA, and it will be necessary to repeat the path again to her from the top $a$and from the top b- thus we find the desired cycle.
- Obviously, this algorithm works in time order of the length 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 a tree.
- We want to create a new component costal doubly connected, which will consist of all the vertices of the detected cycle (of course, found himself cycle could consist of some doubly connected component, but it does not change anything). Furthermore, it is necessary to perform compression in such a manner not to disturb the structure of wood, 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, ieit parremains unchanged. For all other vertices compressible update also do not need anything, because these peaks simply cease to exist - in the system of disjoint sets for doubly connected component of all these vertices will just point to the top-LCA.
- But then it turns out that the system of disjoint sets for doubly connected component works without union by rank heuristic: if we always attach them to the top of the cycle LCA, this is no place heuristics. In this case, the asymptotic behavior occurs $O(\log n)$because without heuristics rank with any operation system of disjoint sets it works for a time.
- **To achieve the asymptotic behavior of**$O(1)$ one request is necessary to combine the top of the cycle according to rank heuristics, and then assign a par new leader $\mathrm{par}[\mathrm{LCA}]$.

Implementation

We present here the final implementation of the whole algorithm.

For simplicity, a system of disjoint sets for doubly connected component written **without rank heuristics** , so the final asymptotics make $O(\log n)$a request on average. (For information on how to reach the asymptotic behavior $O(1)$, described above in paragraph "compression cycle.")

Also in this embodiment, the ribs themselves are not stored bridges and kept only their number - see variable $\mathrm{bridges}$. However, if desired, will not be difficult to have setall of the bridges.

Initially, you must call the function $\mathrm{init}()$that initializes the system two disjoint sets (assigning each vertex in a separate set, and affixing a size equal to one), marks the ancestors par.

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

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

More detailed explanations to this implementation, see 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);
}
```

Comment on the code in more detail.

System of disjoint sets for doubly connected component is stored in the array $\mathrm{bl}[]$, and a function that returns the leader doubly connected components - it $\mathrm{get}(v)$. This function is used many times in the rest of the code, because you need to remember that after several compression tops 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 system of disjoint sets for connected components, etc.).

System for disjoint sets of connected components is stored in the array $\mathrm{comp}[]$, there is also an additional array $\mathrm{size}[]$storage component sizes. The function $\mathrm{get_comp}(v)$returns a leader 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 root ancestor, ancestor redirecting every time parin the opposite direction (down towards the top v). Pointer is also updated compin the system for disjoint sets of connected components to point to the new root.After the new root perepodveshivaniya DIMENSIONS sizeconnected components. Note that the implementation every time we call the function $\mathrm{get}()$to access it to the leader of 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, searches for peaks LCA $a$and b, which rises simultaneously up from them, any peak did not yet meet the second time. To be effective, passed vertex tagged with art "numerical used", what works for $O(1)$instead of applying set. Completed path is stored in the vectors $\mathrm{va}$and vbthen to walk through it a second time before LCA, thereby obtaining all the vertices of the cycle. All vertices are compressed cycle by attaching them to the LCA (asymptotic behavior arises here $O(\log n)$, because when 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 minimal tree perepodveshivaetsya for new root and then attached to a large tree. Otherwise, if the vertices $a$and $b$lie in the same tree, but in different components of doubly connected, the function is called $\mathrm{merge\_path}(a, b)$, which detects the cycle, and compresses it into a doubly connected component.