Search bridges

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.

Below we describe an algorithm based on DFS , and the running time $O(n+m)$, where n- the number of vertices m- edges in the graph.

Note that the site also describes the online search algorithm bridges - in contrast to the algorithm described here, an online algorithm is able to maintain all the bridges in a changing graph graph (refers to adding new edges).

Algorithm

Start the tour in depth from an arbitrary vertex of the graph, denoted by root. We note the following **fact** (which is not hard to prove):

- Suppose we are bypassed in depth, looking now all edges from the top v. Then, if the current edge $(v,to)$is that from the top $to$and from any of its descendant tree traversal depth of no return to the top of the ribs $v$or any of its ancestor, then this edge is a bridge. Otherwise, it is not a bridge. (In fact, this condition we check if there are other ways of $v$to toother than the descent along the edge $(v,to)$of the tree traversal in depth.)

It now remains to learn how to check this fact for each vertex effectively. For this we use the "time of entry into the summit," is calculated by the search algorithm in depth .

So let $tin[v]$- this time call DFS at the top v. Now we introduce an array $fup[v]$, which will allow us to answer the above questions. Time $fup[v]$is the minimum time of entering into the very top $tin[v]$, sunset times in each vertex p, which is the end of a reverse edge (v,p), as well as of all the values $fup[to]$for each vertex to, which is a direct son $v$in the search tree:

$$fup[v] = \min \begin{cases} tin[v], & \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

(Here "back edge" - the opposite edge, "tree edge" - edge of the tree)

Then, from the top $v$or her offspring have the opposite edge in its parent if and only if there is a son tothat $fup[to] \le tin[v]$. (If $fup[to] = tin[v]$it means that there is an edge opposite that comes precisely v, if $fup[to] < tin[v]$, it indicates the presence of edges in an inverse ancestor vertex v.)

Thus, if the current edge (v, to)(belonging to the tree search) is satisfied $fup[to] > tin[v]$, then this edge is a bridge, otherwise it is not a bridge.

Implementation

If we talk about the actual implementation, here we must be able to distinguish three cases: when we go on the edge of the tree depth-first search, when we go to the opposite edge, and when trying to go along the edge of the tree in the opposite direction. It is, accordingly, the cases:

- $used[to] = false$ - Edges of the tree search criterion;
- $used[to] = true \ \&\& \ to \neq parent$ - Criterion reverse rib;
- $to = parent$ - Criterion of passage along the edge of the search tree in the opposite direction.

Thus, for the implementation of these criteria, we need to pass in the search function in the top of the depth of the current node ancestor.

```
const int MAXN = ...;
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

void dfs (int v, int p = -1) {
	used[v] = true;
	tin[v] = fup[v] = timer++;
	for (size_t i=0; i<g[v].size(); ++i) {
		int to = g[v][i];
		if (to == p)  continue;
		if (used[to])
			fup[v] = min (fup[v], tin[to]);
		else {
			dfs (to, v);
			fup[v] = min (fup[v], fup[to]);
			if (fup[to] > tin[v])
```

```
                    IS_BRIDGE(v,to);
            }
        }
}

void find_bridges() {
        timer = 0;
        for (int i=0; i<n; ++i)
                used[i] = false;
        for (int i=0; i<n; ++i)
                if (!used[i])
                        dfs (i);
}
```

Here, the main function to call - it find_bridges- it produces the necessary initialization and start crawling in depth for each connected component of the graph.

In this case $\text{IS\_BRIDGE}(a,b)$- it's a function that will respond to the fact that the edge $(a,b)$is a bridge, for example, show this to the screen edge.

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

It should be noted that this implementation does not work correctly in the presence of the column **multiple edges** : it does not actually pay attention, fold edge or whether it is unique. Of course, multiple edges do not have to go back, so if you call IS_BRIDGE , you can check further, if not a multiple edge we want to add to the response. Another way - more accurate work with the ancestors, ie transmit to the dfstop is not an ancestor, and the number edges, on which we went to the top (you have to be further store rooms all edges).

Problem in online judges

List of tasks that require search bridges:

- UVA # 796 **"Critical Links"** [Difficulty: Easy]
- UVA # 610 **"Street Directions"** [Difficulty: Medium]