Last update: June 25, 2024 Translated From: e-maxx.ru

Finding bridges in a graph in $O(N + M)$

We are given an undirected graph. A bridge is defined as an edge which, when removed, makes the graph disconnected (or more precisely, increases the number of connected components in the graph). The task is to find all bridges in the given graph.

Informally, the problem is formulated as follows: given a map of cities connected with roads, find all "important" roads, i.e. roads which, when removed, cause disappearance of a path between some pair of cities.

The algorithm described here is based on depth first search and has $O(N + M)$ complexity, where N is the number of vertices and M is the number of edges in the graph.

Note that there is also the article Finding Bridges Online - unlike the offline algorithm described here, the online algorithm is able to maintain the list of all bridges in a changing graph (assuming that the only type of change is addition of new edges).

Algorithm

Pick an arbitrary vertex of the graph $root$ and run depth first search from it. Note the following fact (which is easy to prove):

- Let's say we are in the DFS, looking through the edges starting from vertex v. The current edge (v, to) is a bridge if and only if none of the vertices to and its descendants in the DFS traversal tree has a back-edge to vertex v or any of its ancestors. Indeed, this condition means that there is no other way from v to to except for edge (v, to).

Now we have to learn to check this fact for each vertex efficiently. We'll use "time of entry into node" computed by the depth first search.

So, let $\texttt{tin}[v]$ denote entry time for node v. We introduce an array $\texttt{low}$ which will let us store the node with earliest entry time found in the DFS search that a node v can reach with a single edge from itself or its descendants. $\texttt{low}[v]$ is the minimum of $\texttt{tin}[v]$, the entry times $\texttt{tin}[p]$ for each node p that is connected to node v via a back-edge (v, p) and the values of $\texttt{low}[to]$ for each vertex to which is a direct descendant of v in the DFS tree:

$$\texttt{low}[v] = \min \begin{Bmatrix} \texttt{tin}[v] & \\ \texttt{tin}[p] & \text{for all } p \text{ for which } (v,p) \text{ is a back edge} \\ \texttt{low}[to] & \text{for all } to \text{ for which } (v,to) \text{ is a tree edge} \end{Bmatrix}$$

Now, there is a back edge from vertex v or one of its descendants to one of its ancestors if and only if vertex v has a child to for which $\texttt{low}[to] \leq \texttt{tin}[v]$. If $\texttt{low}[to] = \texttt{tin}[v]$, the back edge comes directly to v, otherwise it comes to one of the ancestors of v.

Thus, the current edge (v, to) in the DFS tree is a bridge if and only if $\texttt{low}[to] > \texttt{tin}[v]$.

Implementation

The implementation needs to distinguish three cases: when we go down the edge in DFS tree, when we find a back edge to an ancestor of the vertex and when we return to a parent of the vertex. These are the cases:

- $\texttt{visited}[to] = false$ - the edge is part of DFS tree;
- $\texttt{visited}[to] = true$ && $to \neq parent$ - the edge is back edge to one of the ancestors;
- $to = parent$ - the edge leads back to parent in DFS tree.

To implement this, we need a depth first search function which accepts the parent vertex of the current node.

For the cases of multiple edges, we need to be careful when ignoring the edge from the parent. To solve this issue, we can add a flag `parent_skipped` which will ensure we only skip the parent once.

```
void IS_BRIDGE(int v,int to); // some function to process the found bridge
int n; // number of nodes
vector<vector<int>> adj; // adjacency list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    bool parent_skipped = false;
    for (int to : adj[v]) {
        if (to == p && !parent_skipped) {
            parent_skipped = true;
            continue;
        }
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
```

```
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] > tin[v])
                IS_BRIDGE(v, to);
        }
    }
}

void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i])
            dfs(i);
    }
}
```

Main function is `find_bridges`; it performs necessary initialization and starts depth first search in each connected component of the graph.

Function `IS_BRIDGE(a, b)` is some function that will process the fact that edge (a, b) is a bridge, for example, print it.

Note that this implementation malfunctions if the graph has multiple edges, since it ignores them. Of course, multiple edges will never be a part of the answer, so `IS_BRIDGE` can check additionally that the reported bridge is not a multiple edge. Alternatively it's possible to pass to `dfs` the index of the edge used to enter the vertex instead of the parent vertex (and store the indices of all vertices).

Practice Problems

- UVA #796 "Critical Links" [difficulty: low]
- UVA #610 "Street Directions" [difficulty: medium]
- Case of the Computer Network (Codeforces Round #310 Div. 1 E) [difficulty: hard]
- UVA 12363 - Hedge Mazes
- UVA 315 - Network
- GYM - Computer Network (J)
- SPOJ - King Graffs Defense
- SPOJ - Critical Edges
- Codeforces - Break Up
- Codeforces - Tourist Reform
- Codeforces - Non-academic problem

Contributors:

Nalin Bhardwaj (31.82%) jakobkogler (18.18%) tcNickolas (12.73%) pr4-kp (11.82%) NaimSS (8.18%) wikku (5.45%) Morass (5.45%) adamant-pwn (2.73%) likecs (1.82%) roll-no-1 (0.91%) vatsalsharma376 (0.91%)