ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Algorithms Lab HS20
Department of Computer Science
Prof. Dr. E. Welzl
cadmo.ethz.ch/education/lectures/HS20/algolab

Solution — Important bridges

1 The problem in a nutshell

Given a connected, undirected graph, find all edges whose removal would disconnect the graph. In graph theory, such edges are called *bridges* (not to be confused with the task description).

2 Modeling

Clearly the problem is of a graph-theoretical nature: the islands correspond to vertices and the bridges to edges. Therefore we are looking either for (i) an ad-hoc (STL-only) solution, (ii) a solution that makes use of an algorithm available in the Boost Graph Library, or (iii) a combination of the two.

In all three cases, we have to decide how to represent the graph. Options to be considered are an *adjacency list* or an *adjacency matrix*. From the problem statement it is very unlikely that just keeping the edge list or a sorted version thereof would cut it, since there is no possibility to make use of any ordering on the vertex labels. Hence let's look at the graph:

> "[...] you can get from any island to any other (possibly using several bridges). Every bridge connects two different islands and for any pair of islands there is at most one bridge that connects them."

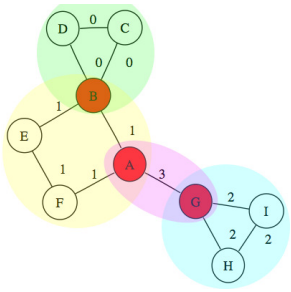
We see that the underlying graph G is connected, undirected, unweighted and has neither loops ("two different islands") nor parallel edges ("at most one bridge"). So far, both representations seem to be reasonable choices (while for example a graph with parallel edges would prevent the use of BGL's `adjacency_matrix`). Next we look at the task limits: We have n vertices ($0 \leqslant n \leqslant 3 \cdot 10^4$) and m edges ($0 \leqslant m \leqslant 3 \cdot 10^4$), indicating that the larger input graphs of this task are going to be sparse graphs. Hence there is no reason to deviate from the recommendation to use an adjacency list.[1] In such a case it can be useful to solve the problem subtask by subtask, always adapting the existing algorithm to incorporate new ideas necessary for the next one or two subtasks.
This is not the case here, the first subtask is different from the others only in terms of the size of the input ($m \leqslant 500$): We can see that for such small graphs also algorithms with a subcubic (and certainly a quadratic) runtime should work within the timelimit.

$O(m(n+m))$ solution (first subtask). There is a simple algorithm achieving this runtime: For every edge e, we check whether its removal disconnects the graph (or equivalently, increases the number of connected components). We have m edges, and for each edge we can check the connectivity of $G \setminus \{e\}$ either by running a BFS/DFS or by invoking BGL's `connected_components`.

[1] For example a subtask might have only bipartite input graphs instead of arbitrary graphs, or a small number of agents instead of a large number of agents, ...

$O(n+m)$ **solution (all subtasks).** To improve on the runtime of the partial solution we need a better understanding of connectivity concepts for graphs. Therefore let us have a look at what algorithms in this area are available in the Boost Graph Library: `strong_components`, `connected_components`, `biconnected_components` and `incremental_components`. We have already seen in the first BGL tutorial problem that strongly connected components are a concept for directed graphs; and we have explored (all) possibilities of using connected components in the partial solution. Incremental components are a concept for dynamic graphs (graphs with edge insertion updates), hence we are left with biconnected components. These are defined as follows:

> A connected graph on two or more vertices is *biconnected* if the removal of any single vertex (and all edges incident to that vertex) does not disconnect the graph. The *biconnected components* or *blocks* of a graph are the maximal biconnected subgraphs.



This rather involved definition is exemplified by the graph on the right, which has four biconnected components. Each edge belongs to exactly one biconnected component[2] and is labeled with that component's index ($0, 1, 2$ or 3).

In other words, every biconnected component is a subgraph of G on *either* a) three or more vertices, with at least two vertex-disjoint paths between each pair of vertices, *or* two vertices that are connected by a bridge of the graph.

Once we have established this connection, there is a straight-forward linear-time algorithm to find all important bridges of the graph:

1. Find all biconnected components of the graph G with BGL's `biconnected_components`. According to the BGL documentation, this takes time $O(n+m)$.
2. Among these biconnected components, find all those that contain exactly one edge. This can be done by iterating over all the edges to look up their component index in $O(m)$ time overall.
3. Find all edges belonging to a component which contains exactly one single edge. This can again be done by iterating over all the edges in time $O(m)$.

3 Implementation

In the following, we will show how to come up with an implementation of the full solution. We mainly focus on Step 1. The goal is not only to show how `biconnected_components` works, but also how you can use the BGL docs and our provided code snippets to find what you need. This probably is – if you're new to BGL – a lengthy process, and this exercise was supposed to help you on the way. For the considerations in this Section, please refer to the BGL documentation and the BGL code snippets provided by us on Moodle.

Remark: `biconnected_components` is a DFS-based algorithm due to Robert Endre Tarjan [1].

[2] Think about it: If an edge belonged to two or more biconnected components at the same time, we could merge those components, contradicting their maximality.

Using this "hammer" is a slight overkill, since it finds all biconnected components, not only those of size 1 (the bridges). Feel free to implement Tarjan's bridge-finding algorithm [2] instead. The algorithm is basically a pre-order DFS that numbers vertices based on exploration time. For each vertex it keeps track of the lowest/highest ID over all subtree vertices and their adjacent vertices (via back and cross edges). This direct bridge-finding algorithm is not available in the BGL; you would need to implement it yourself.

biconnected_components Let us look at the BGL documentation of `biconnected_components`. First of all, we can see that the algorithm returns an integer (say `ncc`), denoting the number of biconnected components of the graph. What can we use it for? It tells you that BGL numbers the biconnected components with indices $0, \ldots,$ `ncc` $- 1$. How do we get to know the actual biconnected components? Apart from the BGL docs, I highly recommend to scan the BGL code snippets that we provided[3] for hints/similar examples. There we can see that only one of the presented algorithms (namely, Kruskal) directly records (in an output iterator) what it computes – all the other algorithms work with exterior or interior property maps.

Hence you can ask yourself what is the example closest to biconnected components? There is no definite answer, but let us look at `connected_components` in the BGL code snippets file (it is also about components and returns an integer denoting the number of components):

```
// We MUST use such a vector as an Exterior Property Map: Vertex -> Component
std::vector<int> component_map(n);
int ncc = boost::connected_components(G, boost::make_iterator_property_map(
    component_map.begin(), boost::get(boost::vertex_index, G)));
```

Is there another similarity? Well, yes: the second parameter of `connected_components` is a ComponentMap, that is, a property map just as in `biconnected_components`. As explained in the tutorial, property maps allow us to access information about edges and vertices. Here we pass a ComponentMap parameter to the algorithm in order to record the component information in form of an integer between 0 and `ncc` $- 1$. Let us verify this in the BGL documentation of `biconnected_components`:

> "OUT: ComponentMap comp: The algorithm computes how many biconnected components are in the graph, and assigns each component an integer label. The algorithm then records which component each edge in the graph belongs to by recording the component number in the component property map."

Sounds promising! So, are we done yet? Can we just copy the `connected_components` code snippet and go ahead? No, there is another important difference between connected and biconnected components: The ComponentMap in `connected_components` records information about *vertices*, while the ComponentMap in `biconnected_components` records information about *edges*! Check once again at the BGL documentation of `biconnected_components`:

> "Vertices can be present in multiple biconnected components, but each edge can only be contained in a single biconnected component. The output of the `biconnected_components` algorithm records the biconnected component number of each edge in the property map `comp`."

Why is this a problem? Well, vertices in BGL are conveniently numbered $0, \ldots, n - 1$. That's why we can simply pass a vector as an Exterior Vertex Property Map. Can we also pass

[3] You can find them on https://algolab.inf.ethz.ch/doc/tutorials/sample_code/

an Exterior *Edge* Property Map? Yes, this is possible, but we can also use *Interior* Edge Properties either by defining a custom edge property or by reusing an existing one. We will describe the details of all three options below.

Custom edge properties Let us check[4] whether there exists a predefined interior edge property for components in the algolab documentation: https://algolab.inf.ethz.ch/doc/boost/doc/libs/1_74_0/boost/graph/properties.hpp.html. This is not the case. There are three ways around this: (i) Use an existing property, for example use `edge_name_t` to store component indices, (ii) define your own custom property, or (iii) use an exterior property map to store component indices. Option (ii) can be done as follows.

```
12 // Define custom interior edge property
13 namespace boost {
14   enum edge_component_t { edge_component = 216 }; \\ unique ud
15         BOOST_INSTALL_PROPERTY(edge, component);
16 }
```

Option (iii) requires that we use `edge_index_t` as edge properties for indexing of the exterior property map. This edge index `i` has to be set when inserting edges:

```
37         boost::tie(e, success) = boost::add_edge(u, v, i, G); // Option 3
```

The choice among options (i), (ii) and (iii) also reflects in the BGL typedefs: As already mentioned in the Modeling Section, we use an adjacency list to represent the graph. We do not need any vertex properties. For the interior edge property we have the three options mentioned above:

```
17 // BGL typedefs
18 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::undirectedS,
19         boost::no_property,                       // no vertex properties
20         boost::property<boost::edge_name_t, int> // Option 1: (ab)use edge_name_t
21   // boost::property<boost::edge_component_t, int>   // Opt.2: custom property
22   // boost::property<boost::edge_index_t, std::size_t> // Opt.3: exterior property
23         > Graph;
24 typedef boost::property_map<Graph, boost::edge_name_t>::type ComponentMap; //Opt. 1
25 //typedef property_map<Graph, edge_component_t>::type ComponentMap;     // Option 2
26 typedef boost::graph_traits<Graph>::edge_iterator EdgeIt;
```

Remainder We are now ready to invoke BGL's biconnected components. Afterwards it remains to first find all biconnected components that contain exactly one edge, and then to find all edges belonging to such a component (in which case we have a bridge). We can do both of these two tasks by iterating over all the edges with edge iterators.

```
39         // Calculate biconnected components
40         ComponentMap componentmap = boost::get(boost::edge_name, G); // Option 1
41 //      ComponentMap componentmap = get(edge_component, G);          // Option 2
42 //      std::vector<std::size_t> edge_component(E);              // Option 3
43 //      auto componentmap = boost::make_iterator_property_map(
44 //              edge_component.begin(), get(boost::edge_index, G)); // Option 3
45         int ncc = boost::biconnected_components(G, componentmap);
```

[4] Note: The links to http://algolab.inf.ethz.ch are only accessible from within ETH or using a VPN connection into the ETH network. Alternatively, you can find it at https://www.boost.org/doc/libs/1_74_0/boost/graph/properties.hpp

```
46          // Iterate over all edges; count number of edges in each component.
47          std::vector<int> componentsize(ncc);
48          EdgeIt ebeg, eend;
49          for (tie(ebeg, eend) = boost::edges(G); ebeg != eend; ++ebeg) { ...
```

Caveats There are a few caveats you need to consider:

- The input size is rather large. So, do not forget `ios_base::sync_with_stdio(false)` at the beginning of `main()`.
- Do not forget that each edge $e = (u, v)$ to output must follow the specified format `min(u,v) max(u,v)` and that the output must be sorted.
- Do not overcomplicate things. In particular, do not use unnecessary library parts, such as BGL's `articulation_points`. If you do, make sure your algorithm also solves all possible bordercases correctly (e.g., not every edge between two articulation points is a bridge)!
- Always compile BGL programs with the `-O2` flag to get a runtime on your system which is (roughly) comparable to the runtime on code expert.

4 A Complete Solution

```
1  // STL includes
2  #include <iostream>
3  #include <vector>
4  #include <algorithm>    // sort, min, max
5  // BGL includes
6  #include <boost/graph/adjacency_list.hpp>
7  #include <boost/graph/biconnected_components.hpp>
8
9  // Define custom interior edge property // Option 2: use custom property
10 namespace boost {
11   enum edge_component_t { edge_component };
12          BOOST_INSTALL_PROPERTY(edge, component);
13 }
14 // BGL typedefs
15 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::undirectedS,
16          boost::no_property,                              // no vertex properties
17          boost::property<boost::edge_name_t, int> // Option 1: (ab)use edge_name_t
18   // boost::property<boost::edge_component_t, int>   // Opt.2: custom property
19   // boost::property<boost::edge_index_t, std::size_t> // Opt.3: exterior property
20          >                                                Graph;
21 typedef boost::property_map<Graph, boost::edge_name_t>::type ComponentMap; //Opt. 1
22 // typedef boost::property_map<Graph, edge_component_t>::type ComponentMap; // Opt.2
23 typedef boost::graph_traits<Graph>::edge_iterator               EdgeIt;
24 typedef boost::graph_traits<Graph>::vertex_descriptor           Vertex;
25 typedef boost::graph_traits<Graph>::edge_descriptor             Edge;
26
27 // Function for solving a single testcase
28 void testcases() {
29          // Read Graph
30          int V, E; std::cin >> V >> E;   // islands and bridges
31          Graph G(V);
32          for (int i = 0; i < E; ++i) {
33                  int u, v; std::cin >> u >> v;
34                  Edge e; bool success;
```

```
                boost::tie(e, success) = boost::add_edge(u, v, G); // Options 1 & 2
                // boost::tie(e, success) = boost::add_edge(u, v, i, G); // Option 3
        }
        // Calculate biconnected components
        ComponentMap componentmap = boost::get(boost::edge_name, G); // Option 1
//      ComponentMap componentmap = get(edge_component, G);          // Option 2
//      std::vector<std::size_t> edge_component(E);              // Option 3
//      auto componentmap = boost::make_iterator_property_map(
//              edge_component.begin(), get(boost::edge_index, G)); // Option 3
        int ncc = boost::biconnected_components(G, componentmap);
        // Iterate over all edges; count number of edges in each component.
        std::vector<int> componentsize(ncc);
        EdgeIt ebeg, eend;
        for (tie(ebeg, eend) = boost::edges(G); ebeg != eend; ++ebeg) {
                componentsize[componentmap[*ebeg]]++;
        }
        // Solution vector to record bridges
        std::vector<std::pair<int,int> > bridges;
        for (boost::tie(ebeg, eend) = boost::edges(G); ebeg != eend; ++ebeg) {
                if (componentsize[componentmap[*ebeg]] == 1) { // If edge in a
                        int u = source(*ebeg, G); // component of size = 1
                        int v = target(*ebeg, G);
                        bridges.push_back(std::make_pair(std::min(u,v),
                                                         std::max(u,v)));
                }
        }
        std::sort(bridges.begin(), bridges.end());
        // Output
        std::cout << bridges.size() << std::endl;
        for (size_t i = 0; i < bridges.size(); ++i) {
                std::cout << bridges[i].first << "␣" <<
                             bridges[i].second << std::endl;
        }
}

// Main function, looping over the testcases
int main() {
        std::ios_base::sync_with_stdio(false);
        int T; std::cin >> T;
        for ( ; T > 0; --T) testcases();
        return 0;
}
```

References

[1] Robert E. Tarjan, Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, **1**, 2, (1972), 146–160.

[2] Robert E. Tarjan, A Note on Finding the Bridges of a Graph. *Inform. Process. Lett.*, **2**, 6, (1974), 160–161.