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 one should consider 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."

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 ($0 \leq n \leq 3 \cdot 10^4$) and m edges ($0 \leq m \leq 3 \cdot 10^4$), 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]

At last, let us get the remaining information out of the *input* and *output specifications:* First of all, n and m may be 0, so we need to watch out for bordercases. Secondly, the islands are 0-based (numbered from 0 to $n-1$ in contrast to 1 to n), which will be convenient for implementation (i.e. we can use the island numbers directly as vector indices). Finally, the output *needs to be sorted*! (Both interiorly in each edge, and lexicographically between the edges.)

[1] In the first BGL tutorial, we recommended to use adjacency lists almost exclusively, with the exceptions being a) dense graphs and b) the use of algorithms that require a lot of edge lookups. Neither is the case here; furthermore the recommendation is also valid if you want to write your own ad-hoc algorithm.

3 Algorithm Design

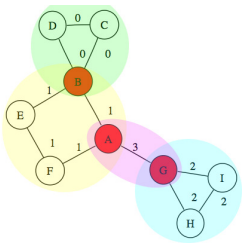
From the problem description we learn that the graph has at most $n = 3 \cdot 10^4$ vertices and $m = 3 \cdot 10^4$ edges. Any algorithm with quadratic runtime (e.g. $O(nm)$) thus has a number of operations in the order of 1 billion. Hence for a full solution we are looking for algorithms with a subquadratic runtime; candidates are graph search algorithms (BFS/DFS, in $O(n+m)$), connectivity algorithms (connected/strongly connected/biconnected components, in $O(n+m)$), distance-related algorithms (Dijkstra shortest paths/Kruskal MST/Prim MST, in $O(n \log n + m \log n)$), or ad-hoc algorithms with up to a square root overhead over a linear time solution.

Subtask specifications Looking at the subtask specifications often can give additional insight into the problem. The subtasks can for example describe a restricted setting, for which not all ideas of the full solution are necessary.[2] 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 ($m \leq 500$): We can see that for such small graphs also algorithms with a subcubic (and certainly a quadratic) runtime will pass 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`.

$O(n+m)$ solution (all subtasks) To improve on the runtime of the partial solution we need a better understanding of connectivity concepts of graphs. Therefore let's 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 is biconnected if the removal of any single vertex (and all edges incident on that vertex) can not disconnect the graph. More generally, the biconnected components of a graph are the maximal subsets of vertices such that the removal of a vertex from a particular component will not disconnect the component."



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

[2] 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, ...

[3] 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.

In other words, every biconnected component is an inclusion-maximal subgraph of G on *either* a) 3 or more vertices, with at least two vertex-disjoint paths between each pair of vertices, *or* b) 2 vertices, connected by a bridge of the graph, *or* c) 1 isolated vertex.

Once we have established this connection, there is a straight-forward linear-time algorithm to find all 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. Find among these biconnected components all those that contain exactly one edge. This can be done by iterating over all the edges to look up their component index $(O(m))$.
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 $O(m)$.

4 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` work, but actually 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.[4] 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 instead.[5]

biconnected_components Let's start by looking at `biconnected_components`' BGL documentation: First of all, we can see that the algorithm returns an integer (say `numbc`), 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 $0, \ldots, \mathtt{numbc} - 1$. How do we get to know the actual biconnected components? Apart from the BGL docs, I highly recommend to scan the BGL code snippets file that we provided 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's no definite answer, but let's exemplary 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 the following vector as an Exterior Property Map: Vertex -> Component
std::vector<int> componentmap(V);
int numcc = boost::connected_components(G, boost::make_iterator_property_map(
                        componentmap.begin(), boost::get(boost::vertex_index, G)));
```

[4] R. Tarjan, *Depth first search and linear graph algorithms.* SIAM Journal on Computing, 1(2):146–160, 1972

[5] R. Tarjan, *A note on finding the bridges of a graph.* Information Processing Letters, 2(6):160161, 1974 This is basically a pre-order DFS which numbers vertices based on exploration time, and for each vertex keeps track of the lowest/highest ID over all subtree vertices and their adjacent vertices (via back and cross edges). The bridge-finding algorithm is not available in the BGL; you need to implement it yourself, from scratch.

Is there another similarity? Well yes: the second parameter of `connected_components` is a ComponentMap, i.e. a property map just as we need to use it in `biconnected_components`. As explained in the lecture, property maps allow us to access information about edges and vertices. If we pass a ComponentMap parameter to our algorithm, it should thus record the information (i.e. an integer between 0 and $numbc - 1$). Let's 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? 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*! Let's again look 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 an Exterior *Edge* Property Map? Unfortunately, this is not documented, hence we need to look for an alternative way. What can we do? Use an *Interior* Edge Property!

Custom edge properties Let's check whether there already exists a predefined interior edge property for components: http://www.boost.org/doc/libs/1_58_0/boost/graph/properties.hpp. This is not the case. There are two ways around this: (i) Use an existing property, for example use `edge_name_t` to store integers (e.g. component indices), see also Page 43 of the first BGL tutorial handout. (ii) Define your own custom property, see Page 46 of the first BGL tutorial. The latter can be done as follows:

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

The choice between options (i) and (ii) 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 two 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 //      property<boost::edge_component_t, int>   // Option 2: use custom property
22         > Graph;
23 typedef boost::property_map<Graph, boost::edge_name_t>::type ComponentMap; //Opt. 1
24 //typedef property_map<Graph, edge_component_t>::type ComponentMap;     // Option 2
25 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          int numbc = boost::biconnected_components(G, componentmap);
43          // Iterate over all edges; count number of edges in each component.
44          std::vector<int> componentsize(numbc);
45          EdgeIt ebeg, eend;
46          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. Stick to either `scanf`/`printf` or `cin`/`cout` and promise that you do so by adding `ios_base::sync_with_stdio(false);` to the beginning of `main()`.
- Do not forget that each edge $e = (u, v)$ which you output must follow the specified format `min(u,v) max(u,v)` and that the outputted edges should 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 the judge.

5 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
10 namespace boost {
11         enum edge_component_t { edge_component = 216 }; // arbitrary, unique id
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 //      property<boost::edge_component_t, int>   // Option 2: use custom property
19         >                                                    Graph;
20 typedef boost::property_map<Graph, boost::edge_name_t>::type ComponentMap; //Opt. 1
21 //typedef property_map<Graph, edge_component_t>::type ComponentMap;     // Option 2
22 typedef boost::graph_traits<Graph>::edge_iterator              EdgeIt;
23 typedef boost::graph_traits<Graph>::vertex_descriptor          Vertex;
24 typedef boost::graph_traits<Graph>::edge_descriptor            Edge;
```

```cpp

// Function for solving a single testcase
void testcases() {
        // Read Graph
        int V, E; std::cin >> V >> E;   // islands and bridges
        Graph G(V);
        for (int i = 0; i < E; ++i) {
                int u, v; std::cin >> u >> v;
                Edge e; bool success;
                boost::tie(e, success) = boost::add_edge(u, v, G);
        }
        // Calculate biconnected components
        ComponentMap componentmap = boost::get(boost::edge_name, G); // Option 1
//      ComponentMap componentmap = get(edge_component, G);          // Option 2
        int numbc = boost::biconnected_components(G, componentmap);
        // Iterate over all edges; count number of edges in each component.
        std::vector<int> componentsize(numbc);
        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;
}
```