January 2006 Problem 'rpaths' Analysis

by Bruce Merry

The problem can be restated as requiring that one adds the minimum number of edges to make the graph biconnected. The first step is to identify the existing biconnected components; refer to your favourite algorithms textbook to see what a biconnected component is and how to identify one; be aware though that there are two variants of bi-connectivity, depending on whether separate routes must be vertex-disjoint or edge-disjoint; in this case they must be edge-disjoint, so biconnected components are separated by articulation edges (vertex-disjoint biconnectivity is more common and probably what your favourite textbook will discuss, but the algorithms involved are very similar).

We will never need to add edges within a biconnected component, so for the purposes of the problem we can collapse each biconnected component to a single vertex. This will leave the graph as a tree. Each leaf will need a new edge added (since there is currently only one road to its parent), so at least `ceil(leaves / 2)` edges must be added. It is also possible to show that this number is sufficient (hint: joining the left-most and right-most leaf and re-collapsing the newly created biconnected component will almost always reduce the number of leaves by 2). So it is sufficient to count the leaves; you don't actually need to work out which new paths to add.

The solution below was written for a different formulation of the problem and includes some coaching-file output. Ignore that :) .

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <climits>
#include <cassert>
#include <vector>
#include <cstddef>
#include <stack>

using namespace std;

#define MAXC 5000
#define MAXR 10000

struct road {
    int target;
    int dual;

    road() {}
    road(int _target, int _dual) : target(_target), dual(_dual) {}
};

struct castle {
    int id;
    int top;
    int parent;
    int bcc;
    vector<road> roads;

    castle() : id(-1), top(-1), parent(-1), bcc(-1), roads() {}
};

static int C;
static castle castles[MAXC];

#ifndef NDEBUG
static bool connected() {
    stack<int> todo;
    vector<bool> seen(C, false);
    int remain = C;

    todo.push(0);
    seen[0] = true;
    while (!todo.empty()) {
        int cur = todo.top();
        todo.pop();
        remain--;
        seen[cur] = true;
        for (size_t i = 0; i < castles[cur].roads.size(); i++) {
            int nxt = castles[cur].roads[i].target;
            if (!seen[nxt]) {
                todo.push(nxt);
                seen[nxt] = true;
            }
        }
    }
    return remain == 0;
}
#endif

static void readin() {
    int R, x, y;

    ifstream in("rpaths.in");
    in.exceptions(ios::failbit);
    in >> C >> R;
    assert(3 <= C && C <= MAXC);
    assert(C - 1 <= R && R <= MAXR);
    for (int i = 0; i < R; i++) {
        in >> x >> y;
        assert(x != y);
        assert(1 <= x && x <= C);
        assert(1 <= y && y <= C);
        x--;
        y--;
        castles[x].roads.push_back(road(y, castles[y].roads.size()));
        castles[y].roads.push_back(road(x, castles[x].roads.size() - 1));
    }

    assert(connected());
}

static int recurse(int cur, int up_edge) {
    static int id_pool = 0;
    static stack<int> bcc;
    int leaves = 0;

    castles[cur].id = id_pool++;
    castles[cur].top = cur;
    bcc.push(cur);

    for (size_t i = 0; i < castles[cur].roads.size(); i++) {
        int nxt = castles[cur].roads[i].target;

        if ((int) i == up_edge) continue;
        if (castles[nxt].id == -1) {
            castles[nxt].parent = cur;
            leaves += recurse(nxt, castles[cur].roads[i].dual);
        }

        if (castles[castles[nxt].top].id < castles[castles[cur].top].id)
            castles[cur].top = castles[nxt].top;
    }

    if (castles[cur].top == cur) {
        while (bcc.top() != cur) {
            castles[bcc.top()].bcc = cur;
            bcc.pop();
        }
        castles[cur].bcc = cur;
        bcc.pop();

        if (leaves == 0) leaves = 1;
    }
    return leaves;
}

static bool bridge(int c, int edge) {
    return castles[c].top == c && castles[c].roads[edge].target ==
castles[c].parent;
}

static int solve() {
    int leaves = recurse(0, -1);
    int root_children = 0;

    for (int i = 1; i < C; i++) {
        int p = castles[i].parent;
        if (castles[i].top == i && castles[p].bcc == 0) root_children++;
    }

    ofstream dot("roads.dot");
    dot << "graph G {\n"
        << "  node [shape=circle,fontsize=10,width=0.2,label=\"\"]\n";
    for (int i = 0; i < C; i++)
        for (size_t j = 0; j < castles[i].roads.size(); j++) {
            int trg = castles[i].roads[j].target;
            if (trg < i) continue;
            dot << "  " << i + 1 << " -- " << trg + 1;
            if (bridge(i, j) || bridge(trg, castles[i].roads[j].dual))
                dot << " [style=bold]";
            else
                dot << " [len=0.3]";
            dot << "\n";
        }
    dot << "}\n";

    if (root_children == 0) return 0;
    else if (root_children == 1) leaves++;
    return (leaves + 1) / 2;
}

int main() {
    readin();
    ofstream out("rpaths.out");
    out << solve() << "\n";
    return 0;
}
```