December 2005 Problem 'layout' Analysis

by Bruce Merry

Suppose we try to just lay out the cows one at time, from 1 to N. Initially we could start by putting each cow as far away from the previous ones as legal, but eventually this could lead to problems because the like and dislike constraints would be incompatible. So we'd like a way to prevent this up front.

Suppose i < j < k. If i and k like each other and want to be at most A apart, while j and k dislike each other and want to be at least B apart. Then i and j must be placed within A - B of each other. A similar deduction can be made for the mirror situation. If we loop downwards over all k (with inner loops over i and j), we can determine enough extra constraints that either some of these constraints are directly contradictory, or else we can start placing cows without fear of running into a local contradiction (don't forget that the cows are in a fixed order, which introduces extra constraints).

This algorithm is $O(N^3)$, which is too slow. Looking at an implementation of the above, one might see echos of Warshall's algorithm. In fact the problem can be done in terms of shortest paths. The constraints can all be written in the form Y <= X + c (for some cow positions X and Y and constant c). In the case of dislikes, c will be negative. If we have constraints Y <= X + c and Z <= Y + d, then we get the new constraint Z <= X + (c + d) - and in general constraints can be chained together. The eventual goal is a constraint relating cow 1 and cow N (of the form cowN <= cow1 + C), so we want to consider the chain that yields the smallest constant - but that is just the shortest path.

Because there are negative edges, we cannot use Dijkstra's algorithm. Instead, we use Bellman-Ford, which has O(N.(ML+MD)) running time. Some extra tricks are needed to handle the two special cases: if there are any negative cycles in the graph then layout is impossible (Bellman-Ford has a mechanism to detect such cycles), otherwise if there is no path then cowN is unbounded.

Here is Bruce's solution:

```
#include <fstream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <cassert>

using namespace std;

static void quick(int ans) {
    ofstream out("layout.out");
    out << ans << "\n";
    out.flush();
    out.close();
    exit(0);
}
```

```
int main() {
    ifstream in("layout.in");

    int N, ML, MD;
    in >> N >> ML >> MD;

    int edges[ML + MD + N - 1][3];
    int E = 0;

    for (int i = 0; i + 1 < N; i++) {
        edges[E][0] = i + 1;
        edges[E][1] = i;
        edges[E][2] = 0;
        E++;
    }
    for (int i = 0; i < ML; i++) {
        int a, b, l;
        in >> a >> b >> l;
        a--; b--;
        if (a > b) swap(a, b);
        edges[E][0] = a;
        edges[E][1] = b;
        edges[E][2] = l;
        E++;
    }
    for (int i = 0; i < MD; i++) {
        int a, b, d;
        in >> a >> b >> d;
        a--; b--;
        if (a > b) swap(a, b);
        edges[E][0] = b;
        edges[E][1] = a;
        edges[E][2] = -d;
        E++;
    }

    int dist[N];
    fill(dist + 1, dist + N, INT_MAX);
    dist[0] = 0;
    for (int i = 0; i <= N; i++)
        for (int j = 0; j < E; j++)
            if (dist[edges[j][0]] != INT_MAX)
                dist[edges[j][1]] <?= dist[edges[j][0]] + edges[j][2];
    if (dist[N - 1] == INT_MAX) quick(-2);
    for (int j = 0; j < E; j++)
        if (dist[edges[j][0]] != INT_MAX
            && dist[edges[j][1]] > dist[edges[j][0]] + edges[j][2])
            quick(-1);
    quick(dist[N - 1]);
    return 0;
}
```