Code Jam 2013 - Round 3

Analysis: Are We Lost Yet?

Introduction

The problem asks you to determine which part of a path E_1, E_2, ..., E_k can be a prefix of some shortest path from 1 to 2. This would be easy if not for the fact we don't know the exact edge costs - we only know ranges into which they fall.

Small dataset

Let's concentrate on any fixed prefix E_1, E_2, ..., E_p of the proposed path, and try to make it a prefix of some shortest path. If we decide on a shortest path candidate (that includes our prefix), it's obivously advantageous to assume the edges on our path are as short as possible, and the other edges are as long as possible.

Notice that a consequence of this is that we can just restrict ourselves to looking at graphs in which each of the edges is either as long or as short as possible. Thus, since in the small dataset the number of edges is only 20, we can simply check all possibilities - make some of the edges short, the rest long, find the shortest path in the resulting graph (preferring the proposed path by, for instance, decreasing the cost of the edges on this path by a small epsilon), and then out of all the possibilities pick the one that takes the most steps along the proposed path.

Large dataset

This is obviously not going to fly for the large dataset. Again, we fix a prefix E_1, E_2, ..., E_p of the proposed path and try to make it a prefix of the shortest path. Assume E_p ends in Tokyo. Thus, we will try to get from Tokyo to London as fast as possible, while still not allowing for a shorter path from Mountain View to London that doesn't begin with our prefix. If we want to optimize for speed, we can do a binary search for the longest prefix that can be a start of some shortest path; if we optimize for simplicity, we can just iterate over all prefixes.

Imagine two robots that try to reach London as fast as possible. One starts from Tokyo, and has a handicap of the cost to travel from the Mountain View to Tokyo along the proposed path (we call this the "good" robot). When this robot goes across an edge, it will always take the minimal cost - this robot represents the shortest path we hope to construct. The other robot starts from Mountain View, and tries to reach London via some other path. We call this the "bad" robot and will try to force it to take as long as possible. The question is whether we can make the good robot be at least as fast as the bad robot (the bad robot can obviously be equally fast simply by following the good robot).

We already set the costs for the edges E_1 to E_p to the low values. Since our aim is to make the good robot move fast and the bad robot move slow, a naive approach is to simply have the good robot pay the low cost for all other edges, and the bad robot pay the high cost. However, we will still encounter a problem in this model. If two robots go across a same edge, they are actually taking different costs, which is not possible in a fixed configuration.

Notice, however, that the two robots are walking the same graph. Thus, if they reach a same node at a different time, the one that arrived earlier will always beat the other one if the shortest path goes through that node, because it can always follow the other one's route. This means the

later robot does not have any purpose in visiting this node at all. Since ties are resolved in favor of the good robot, we can simply decrease the good robot's handicap by 0.5 to avoid any ties.

Thus, we can solve the problem with a single run of Dijkstra's algorithm. We begin with two starting points - one in Mountain View at time 0, and the other in Tokyo with time equal to the cost of travel along the proposed path to Tokyo minus 0.5. When processing a node, we calculate the costs of outgoing edges as follows:

- If the edge is one of E_1, ..., E_p, we take the low cost.
- If the current node is processed because the good robot reached it (which we know because the cost of the current node is not an integer, it ends with 0.5), the cost is the low cost.
- Otherwise, we are processing the node because the bad robot reached it, and the cost is the high cost.

As Dijkstra's algorithm visits a node only once to process outgoing edges, we will never have a robot visit a node that the other robot reached earlier.