ACM ICPC World Finals 2009

Solution sketches

Disclaimer *These are unofficial descriptions of possible ways to solve the problems of the ACM ICPC World Finals 2009. Any error in this text is my error. Should you find such an error, I would be happy to hear about it at* `austrin@kth.se`.

Also, note that these sketches are just that—sketches. They are not intended to give a complete solution, but rather to outline some approach that can be used to solve the problem. If some of the terminology or algorithms mentioned below are not familiar to you, your favorite search engine should be able to help.

Finally, I want to stress that while I'm the one who has written this document, I do not take credit for the ideas behind these solutions—they come from many different people. In particular, thanks to Derek Kisman for catching some errors in an earlier version of this document.

— Per Austrin

Problem A: A Careful Approach

Since the number of planes is at most 8, an optimal solution can be found by simply trying all $8! = 40320$ possible orders for the planes to land. When trying a specific ordering, the largest possible landing window can be computed by binary searching over the maximum possible window and then greedily checking whether a certain window length can be achieved. Something which may be easy to miss in this problem is that it can be the case that the landing time of a plane should be a non-integral number of seconds.

Problem B: My Bad

This problem essentially consists of two parts: evaluating circuits and finding flawed gates. Evaluating circuits can be done in a straight-forward way, once the somewhat tedious input format has been parsed. To find the flawed gate in the case that the circuit is flawed, one can simply check each gate for each possible error. If more than one possible explanation for the flaw is found, the error can not be completely determined.

Problem C: The Return of Carl

First, suppose that the sequence of faces that Carl visits on his path is given. In this case, the path length can be found by laying out the triangles in 2-dimensional Cartesian space, connected in the sequence they are visited. The length of the path is then simply given by the length of the line segment from the starting point's location in the starting face, to the destination point's location in the destination face. One may object that this is only true provided that the line segment stays inside the laid out triangles, and that, when laid out in the plane, the triangles will not overlap. However, it turns out that, first of all, the triangles will not overlap, and second, if the line segment does not stay inside the triangles, there will

in fact be a different way of laying out the triangles such that the line segment stays inside the triangles and becomes shorter.

Now, we are not given the sequence of faces visited, but because the number of faces is so small, one can simply try all possible such sequences.

There are a lot of messy implementation details involved in getting this right. For instance, one has to be able to find the location of a point in a triangle given its azimuth and zenith angles, and when laying out the triangles in the plane one needs to keep track of whether a move from one face to another constitutes a "left" turn or a "right" turn.

Problem D: Conduit Packing

There are several ways of solving this problem, ranging from complicated solutions whose correctness is easy to prove, to somewhat easier solutions whose correctness is more difficult to prove.

A natural starting point is to binary search for the minimum possible radius r of the outer circle. To check whether radius r is achievable, one can do as follows: start by placing the outer circle, and then try to place the small circles inside it, one by one. When placing a new circle, one can assume without loss of generality that there exists a circle which should touch two of the already placed circles (or the outer circles in case this is the only one placed so far). When all circles have been placed, one can check that none of them intersect to see if the placement succeeded. Trying all possible ways of placing the circles (with only four inner circles, there are only a few hundred different such ways), one can determine whether radius r is possible.

Problem E: Fare and Balanced

This problem may at first look like some sort of max flow problem, but even if the large size of the graph does not scare you away from this approach, I am not aware of any way of modeling the problem this way.

Consider instead the following approach. Let $U(u, v)$ be true if every path from intersection u to intersection v has a unique cost. If $U(1, N)$ is true it is clear that no tolls need to be added, so assume from now on that $U(1, N)$ is not true. Now, suppose there is some vertex v such that both $U(1, v)$ and $U(v, N)$ is false. In this case, no solution can possibly exist, since such a solution would have to incur a toll both on some road "before" v and on some road "after" v in order to ensure that all paths have unique costs.

Let us then suppose that no such vertex v exists, i.e., that either $U(1, v)$ or $U(v, N)$ is true for every v. In this case, a solution can be constructed as follows: Let us say that an edge (u, v, c) from u to v of cost c is *pivotal* if it has the property that $U(1, u)$ is true but $U(1, v)$ is false. Pivotal edges have the following nice properties:

- The fact that $U(1, 1)$ is true but $U(1, N)$ is false, together with the fact that if $U(1, u)$ is false and there is an edge from u to v, then $U(1, v)$ is also false implies that every route from 1 to N contains exactly one pivotal edge.
- The fact that $U(1, v)$ is false implies that $U(v, N)$ is true.

Let $C(1, N)$ denote the maximum length of a path from 1 to N Now consider adding a toll of $C(1, N) - C(1, u) - C(v, N) - c$ to every pivotal edge (u, v, c), if this number is positive

(note that it can not be negative). By the two properties above, it is now easily shown that every route from 1 to N has at most one toll and a total cost of exactly $C(1, N)$.

To make an efficient solution out of this, note that the only quantities we actually need are those of the form $U(1, v)$, $U(v, N)$, $C(1, v)$ and $C(v, N)$, and all such values can be computed in linear time using dynamic programming.

Problem F: Deer-Proof Fence

Let us first consider the special case of using a single fence. To compute the fence length needed, one computes the convex hull of the set of trees. It is then not hard to see that the minimum total fence length will be the total length of the perimeter of the convex hull, plus the circumference of a circle of radius M.

To finish the problem we now need to find a good way of partioning the set of trees into disjoint parts, each of which will be surrounded by a single fence (the perimeter of which can be computed as described above). Because of the small number of trees, an optimal such partition can be computed in a brute-force manner. In order to make it fast enough, one may need to use dynamic programming to remember, for a given set S of trees, what the minimum fence length for S is.

Problem G: House of Cards

The main challenge in this problem is to build a convenient representation of the current game state, for computing the scores of the players and the possible next moves. With this hurdle out of the way, the problem can be solved using a standard min-max search, along with alpha-beta pruning to make the search fast enough.

Problem H: The Ministers' Major Mess

The solution is based on realising that a minister can have at most one unsatisfied vote (or in the case of $k = 1$ or $k = 2$, a minister must have all votes satisfied). This can be expressed by $k \cdot (k - 1)$ implications (if the decision on vote i is opposite to the minister's opinion, then the decision on vote j must be according to the minister's opinion, for all $i \neq j$). Alternatively, it can be expressed as a 2-CNF formula (for each pair of two votes, at least one must be satisfied). Either way, this gives an efficient way of determining whether a solution exists, as 2-SAT is efficiently solvable.

To find out all values which are uniquely determined, the easiest way is to compute the transitive closure of the implications mentioned above, and then apply all known values. A variable is known if it is included in a $k = 1$ or $k = 2$ vote, or if it is implied by its negation.

Problem I: Struts and Springs

The first part of this problem is to determine the tree hierarchy of the windows. This is fairly easy: the parent of a window W_1 is the smallest window W_2 such that W_1 is contained in W_2 (if such a window exists).

The second, more tedious part, is to handle resize operations. The horizontal and vertical dimensions can be handled separately and in the same way. Suppose some window W has

its outer window's width changed by Δs. Let L be the total current length of the horizontal springs that control W (there are between one and three such springs). Then, a spring of length l gets resized to length $l' = l \cdot (1 + \Delta s / L)$. If the device controlling the width of W is a spring, the windows inside W must now be updated since W changed width. If the device controlling the width is a strut, the windows inside W are unchanged, except that their absolute position with respect to the screen may have changed because W or one of its parents may moved.

Problem J: Subway Timing

The problem is probably most easily solved by binary search for the answer. However, checking whether it is possible to achieve a certain answer X is a bit tricky.

There are a few different ways of checking this, ranging from quite complicated dynamic programming solutions to the more elegant direct solution described below. An important parameter in the runtime of all these solutions is what the maximum value of X can be. It is a nice (but surprisingly difficult) exercise to prove that, no matter what the size of the tree is, X does not have to be larger than 118 (the significance of this number is of course that it is $59 \cdot 2$, and that 59 is the maximum possible rounding error on a single edge).

Root the tree arbitrarily, and consider some vertex v of the tree. Let us say that an interval $[a, b]$ is *permissible* for v if the edges in the subtree rooted at v can be rounded in such a way that:

- Every path in the subtree has an error of at most X.
- Every path in the subtree which ends in v has an error which lies in the interval $[a, b]$.

Note that a permissible interval must have $a \leq 0$ and $b \geq 0$. Furthermore, let us say that an interval $[a, b]$ is *redundant* for v if there is some $b' < b$ such that $[a, b']$ is permissible.

Now, checking whether it is possible to achieve a maximum error of X is equivalent to checking whether there exists some permissible error for the root of the tree. We will do this by computing the set of all non-redundant permissible intervals for each node v. Fix some node v, let c be its number of children, and for i between 0 and c, let T_i denote the subtree rooted at v but including only the first i children of v. Suppose $[a, b]$ is a non-redundant permissible interval for T_{i-1}, and that $[a', b']$ is a non-redundant permissible interval for the subtree rooted at the i'th child of v, and that the error for the edge from v to its i'th child can be chosen as e (there are always one or two possible values for e). Then if $a + a' + e$ and $b + b' + e$ both are of absolute value at most X, $[\min(a, a' + e), \max(b, b' + e)]$ is a permissible interval for T_i. All non-redundant permissible intervals for T_i can be constructed this way, but also some redundant ones. In order to keep the list of intervals short (of order X rather than order X^2), one should prune away the redundant ones.

Problem K: Suffix-Replacement Grammars

Let L be the length of the starting and target strings, and consider a directed graph $G = (V, E)$ on strings of length L, where there is an edge from a string x to a string y if x can be transformed to y using one of the suffix-replacement rules. The problem asks for the shortest path from S to T in this graph—however, the graph has 52^L vertices and hence finding a shortest path by a standard BFS is not feasible.

What saves us is the fact that the graph in question has a very special structure. Consider a path from S to T. Some of the transformation rules used involve suffixes of length L, and the remaining transformation rules used involve shorter suffixes. The basic idea is to pre-compute shortest paths in the subgraph of G which only uses transformation rules involving shorter suffixes.

Specifically, let G_l be the analogue of the graph G defined above on strings of length l (note that the edges of this graph correspond to the transformation rules involving suffixes of length at most l), and let $D_l(x, y)$ be the length of a shortest path from a string x to a string y (both of length l) in G_l. Consider a weighted directed graph H_l consisting of strings of length l, where the edges are as follows:

- There is an edge from x to y of cost 1 if there is a transformation rule transforming x to y
- There is an edge from x to y of cost $D_{l-1}(x', y')$ if the first two charactes of x and y are equal and x', y' denote the suffixes of x and y obtained by removing the first character.

Then, the shortest path lengths in H_l are exactly the same as those in G_l. In other words, $D_l(x, y)$ is given by the length of a shortest path from x to y in H_l. So, one (seemingly awkward) way of computing the shortest path from S to T in G (i.e., $D_L(S, T)$) is to iteratively compute the shortest path lengths in H_l for l from 1 to L (since the definition of H_l involve the shortest path lengts in H_{l-1}).

Now, how does this help? The number of vertices of H_l is still 52^l! Recall that the quantity that we are actually interested in is $D_L(S, T)$. Note that to compute this, the only vertices of H_L which are relevant, apart from S and T themselves, are those which occur as either lefthand or righthand side of some transformation rule. More generally, unwinding the definitions, one sees that the only vertices in H_l (i.e., the only strings of length l) that one has to consider are those which are *suffixes* of one of the $2R + 2$ input strings. With this good bound on the effective sizes of the graphs, computing all the relevant shortest paths lengths in H_l can be done using e.g. Floyd-Warshall. A somewhat tricky point in this problem is that the length of a path can be exponentially large in the length of a string, so it may be the case that an answer does not fit in a 32-bit integer.