ACM ICPC World Finals 2018

Solution sketches

Disclaimer *This is an unofficial analysis of some possible ways to solve the problems of the ACM ICPC World Finals 2018. 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. If you find an error, please send an e-mail to* `austrin@kth.se` *about it.*

— Per Austrin and Jakub Onufry Wojtaszczyk

Summary

The problem set this year was a bit harder than in the last few years. This may be a case of the judges having found their way back to a normal difficulty, after having somewhat overdone the easy end of the problem set spectrum for a few years following the way too hard problem set we gave in 2014.
Congratulations to Moscow State University, the 2018 ICPC World Champions!
In terms of number of teams that ended up solving each problem, the numbers were:

Problem	A	B	C	D	E	F	G	H	I	J	K
Solved	106	135	0	5	10	126	7	46	47	0	124
Submissions	262	247	4	15	91	462	75	278	229	1	254

The most popular language was (as usual) C++ by a wide margin: 1813 submissions, trailed by Java at 101, and only 4 submissions for the other languages (C/Kotlin/Python 2/Python 3).

A note about solution sizes: below the size of the smallest judge and team solutions for each problem is stated. It should be mentioned that these numbers are just there to give an indication of the order of magnitude. The judge solutions were not written to be minimal (though some of us may have a tendency to overcompactify our code) and it is trivial to make them shorter by removing spacing, renaming variables, and so on. And of course the same goes for the code written by the teams during the contest!

Problem A: Catch the Plane

Shortest judge solution: 1480 bytes. Shortest team solution (during contest): 811 bytes.

The judges (and Per and Onufry in particular) were divided on how difficult this problem is. The contestants decided that it is one of the easier problems in the set.

To solve it, we are going to go backwards in time, starting from when we have to be at the airport. We will store and update an array that for each station S holds the probability of getting to the airport in time, if we start on S at the time we are currently considering. We initialize this array to 0 in all stations but the airport, and to 1 at the airport, which represents the probabilities at time k. Let's figure out how to update this.

We will sweep by decreasing time, storing, as events, the arrivals and departures of buses. When a bus from station A arrives at station B, the probabilities do not change (the fact of a bus *arriving* does not change the probabilities). However, we have to remember the current probability of reaching the airport from B – we will use that probability to update the probability at A when the bus departs. When a bus departs from A, we have to update the probability of getting to the airport from A. If the current probability at A is p, and the probability at the arrival time at B was q, then if $p \geq q$, there's no point in getting on the bus; while if $p < q$, we can update the probability at A to $rq + (1 - r)p$, where r is the probability the AB bus leaves.

The last thing to deal with is the possibly multiple buses depart from one station at the same time. In this case, we cannot update the probability immediately after processing a bus, we have to process all the buses, see which one gives the best update, and choose that one.

Problem B: Comma Sprinkler

Shortest judge solution: 704 bytes. Shortest team solution (during contest): 926 bytes.

This was the easiest problem in this problem set, but it still was not entirely trivial. The simple approach of just applying Dr. Sprinkler's rules until no changes are made is too slow.

The observation to be made here is that the problem can be turned into a graph problem. Let's create two vertices for every distinct word in the input, one representing the beginning of that word, and one representing the end. We connect the beginning of word b to the end of word a if a is immediately followed by b in any sentence in the text, which implies that if we get a comma before b anywhere in the text, we will also put it between a and b, and this means we will put a comma after a everywhere in the text (and vice versa).

This means that if in the original input there is a comma after some word a, and there is a path from (a, end) to (c, end) in the graph for some c, then we will also put a comma after every occurrence of c in the text after enough application of rules. Thus, the problem is about finding connected components in this graph. This is doable with an application of any graph traversal algorithm.

Thus, we begin by going over the text, and constructing the graph. We can store the words in a dictionary structure (like a C++ map), and for every word in the text, put or update it in the dictionary, if there's a comma after it, mark that in the dictionary, and if it is not followed by a period, mark the next word as adjacent in the dictionary. In a single pass we get enough information to construct the graph. Now, we put the (a,end) vertices into the "to visit" queue and run, say, breadth-first search, and finally reconstruct the text, putting a comma after every word not followed by a period for which (a, end) has been visited by the search.

This, will run, in time, $O(n \log(n))$, where n, is the size, of the input, which, is, fast, enough.

Problem C: Conquer the World

Shortest judge solution: 2510 bytes. Shortest team solution (during contest): N/A bytes.

This was one of the two very hard problems in the set. What is not hard to see is that the problem is simply asking for a minimum cost maximum flow between a set of sources and a set of sinks in a flow graph. The flow graph in question has a special structure: most importantly, it is a tree (and in addition, all edge capacities are infinite – we can move as many armies as we like across an edge – only the source and sink vertices have limited capacities). However the tree can have around 250 000 vertices/edges, so running a general min cost max flow algorithm