15295 Spring 2020 #7 -- Problem Discussion
February 26, 2020

This is where we collectively describe algorithms for these problems. To see the problem statements follow this link. To see the scoreboard, go to this page and select this contest.

A. Lunar New Year and a Wander

B. Firetrucks Are Red

Let n be the number of people, and nn be the number of numbers that occur in the input. We're going to build an undirected graph G on n+nn vertices. The edges are between people and their numbers.

To make the code clean and simple, as we're reading in the favorite numbers of each person we "compress" that set of numbers using a hash table. So the first number we find is mapped to 0, the second (different) one is mapped to 1, etc by this hash table. So person numbered p (and I used 0-based numbering) is vertex p, and number x (after mapping) is vertex number n+x in the graph. And, after I've done the number mapping I create an array translate[] such that I can go back from a mapped number x that is in the range [0,nn-1] to the original number it came from. (I'll need this for printing out my answer at the end.)

So the first phase is reading in all the numbers for each person, compressing them, and also making a list of all the compressed numbers associated with each person.

The second phase is to use these lists of numbers to build an undirected graph G with n+nn vertices. There's an edge between a person and each number in her list. The graph is bipartite, but we don't use that fact until the very end.

Now we need to determine if G is connected, and if so to build a spanning tree of G. I used a breadth first search to do this. (A DFS might work, but there is a risk of stack overflow.) A parent array has one element for each vertex and is initialized to -1 (except parent[0] = 0). The breadth-first search starts with a list of just vertex 0. That's layer 0. Now if it's at layer d, it grows a new layer of all vertices neighboring layer d that have not been visited prior to this. That's layer (d+1). Whenever we put a vertex in layer (d+1) we set its parent pointer to the one in layer d that led to it.

When a layer at some depth is empty, we're done. If we've reached all vertices (all parents have been set to be >=0) we're successful. If not then the graph is not connected and we just print "-1".

Now to print out the desired output, we just run through all vertices numbered 0...n-1 and print out the following line for each one:

(1+i) (1+parent[parent[i]]) (translate[parent[i])

--Danny

Alternatively, you could use dsu aka union find to attempt to join sets of people connected by some other number x. I.e. Form a list (by use of a hash table) of people connected by a number x, namely p1, p2, p3 … Then union p1 with p2, then p1 with p3, etc. Each one that joins two components is put in a list of triples to output later. Do this for each such number x.

At the end, you check that you successfully merged n-1 times, in which case print out the list of triples you made earlier. Otherwise, not everyone is connected, so print Impossible.

C. Famil Door and Brackets

D. Even or Odd

I claim that a subset of coins will satisfy this property if and only if it contains at least one coin with probability exactly one half. Once this is established, the actual algorithm is trivial.
To see why this is true, suppose we have some collection of n coins with some probability of coming up even E and some probability of coming up off O.
Now suppose we add some new point weighted with probability p.
Then this new collection comes up even if and only if the original collection comes up odd and the new coin flips to tails, or the original collection comes up even and the new coin flips heads.
Thus, we establish the probability of the new collection coming up even as o(1-p) + ep.
Similarly, we find the probability of the new collection coming up odd as op + e(1-p).
Equating the two, we find
(1-2p)e = (1-2p)o, which means either p = ½ or e = o.
The first condition already establishes a fair coin being in the collection as a sufficient condition, so we just need to show that's a necessary condition.
Towards this end, suppose towards a contradiction that there are no fair coins in a set satisfying this condition. Then (1-2p) is never 0, so we know we can remove any number of coins without affecting the even/odd distribution. In particular, we can thin down the collection to an empty set of coins, which clearly doesn't satisfy the condition, so we're done.

- Ethan

Here is another way to prove the claim without induction using a combinatorics trick with polynomials.

Let s_1, s_2, …, s_k be the probabilities of the coins in a given set S. Consider the polynomial (s_1 x + (1 - s_1)) (s_2 x + (1 - s_2)) … (s_k x + (1 - s_k))

The polynomial is the sum of terms, each of which picks s_i x or (1 - s_i) in the product over i from 1 to k. The coefficient of x^h, notationally [x^h], counts the total probability of getting h heads. We want [x^0] + [x^2] + … = [x^1] + [x^3] + …, equivalently [x^0] - [x^1] + [x^2] - [x^3] + … = 0.
The LHS is achieved by setting x = -1. Therefore the polynomial must have a term where (s_i (-1) + (1 - s_i)) = (1 - 2*s_i) = 0, or s_i = ½.

- ben_dover

E. Odd One Out

F. Disposable Switches

Without loss of generality let v = 1. We define d(x, k) as the shortest time taken to go from node 1 to node x using at most k edges (cables), not counting the overhead of the cables (so the distance between any two nodes in the graph is just l). We initially pre-compute the values of d(x, k) for all 1 <= x <= n using DP (Bellman-Ford). Suppose an optimal path were to use k edges. Then the total time it takes would be t(k) = d(n, k) + c * k. This is only optimal if t(k) <= t(j) for all j != k. First, consider j < k. We need it to be the case that d(n, k) + c * k < d(n, j) + c * j. So c < (d(n, j) - d(n, k)) / (k - j) for all such j. Similarly, considering j > k, we need it to be the case that c > (d(n, k) - d(n, j)) / (j - k) for all such j. So for every possible k, we have a lower bound and upper bound on c that would make a path using k switches optimal. If the lower bound is greater than the upper bound, then it is not feasible so we don't have to worry about it. Otherwise, if we choose any value of c between the lower and upper bounds, we know that any optimal solution using k switches is also a globally optimal solution for this c. From this, we know the set of valid k values.

Then we can trace back through our DP matrix that we used to compute d(n, k) using DFS to find all optimal solutions that led to d(n, k) and mark the vertices that appear in the DFS tree as possibly used switches (our initial DFS stack would contain all of the valid (n, k) pairs). At the end, we output the unmarked vertices. The time complexity is O(n(m + n)) for the Bellman-Ford DP, O(n^2) to compute upper and lower bounds, and O(n(m + n)) for the DFS at the end (the underlying " solution graph" we are searching here has n^2 vertices and nm edges). This gives an overall time complexity of O(n(m + n)), which is sufficient to pass. One other thing to be careful about is to use long long's for distances during the Bellman-Ford DP because they can sum up to be larger than int_max. Also, storing all possible parent vertices for each entry in the DP matrix would take O(nm) space (around 160 MB), which is close to the memory limit which is why we avoided doing that and just re-computed this in our DFS.