Code Jam 2018 - Round 3

Analysis: Name-Preserving Network

How hard can this be?

At first glance, test set 1 may seem to be approachable even by hand, since only one design is required, and we have considerable flexibility in the number of computers to use. But it turns out to be difficult to satisfy all of the following at once:

- Each computer must use exactly 4 links.
- The network must be connected.
- The computers must be uniquely distinguishable even after their IDs are permuted, and we must know how to distinguish them.

For example, we might try creating a ring of computers in which each one is linked to its two neighbors and neighbors' neighbors. This satisfies the 4-link and connected conditions. But this design is highly symmetric in a way that makes the computers less distinguishable. We can try introducing some irregularities (by swapping some edges around; more on this later) to disrupt the symmetry, but then it becomes even harder to eyeball the design — and how can we "find" our computer again after the judge has permuted their identities?

Harder than it looks

We need a way to give each computer a unique label. Labels need to be such that:

- They are id-agnostic — that is, the label of a computer depends only on the link topology, and not on its own or other computers' ids.
- There exist networks such that each computer has a unique label, and moreover, we can somewhat efficiently find those networks.
- We can efficiently compute the labels of a given network.

If we have a labeling scheme that has all these properties, the solution writes itself: find a network of appropriate size, compute the labels on it and the labels on the shuffled network given by the judge, and match the computers by label.

There are many possible labeling schemes, and experimentation can go a long way to find them. We describe two below, one for the theoretically inclined, and one more manual. There also exist ways to explicitly construct networks that guarantee a particular labeling scheme. Both the constructions and labeling schemes are somewhat complicated and require casework, and only those constructions with labeling schemes that are quick to compute will work for this problem. Constructions are of course significantly easier for Test set 1, as we only need to build one network and we and we can use whichever network size (between 10 and 50) we want.

Let us start with the theoretical one, since it's shorter to write. We can model the network as an undirected graph in which each node has degree 4. If we take the adjacency matrix of the graph and compute its p-th power, we obtain a matrix in which cell (i,j) gives the number of paths of length p between nodes i and j. In the context of this problem, we do not particularly care about these paths per se, but we do care that the count of paths for a pair of nodes (i,j) depends on the entire topology of the network. So, if we are lucky enough, the set of values (i,1), (i,2), ... will be unique for each i. As it turns out, most graphs of a given size yield unique levels for p = 7, and quite possibly for other values as well. We generated graphs for every size between 10 and 50 and only needed to try a second time twice, and never a third time.

For a more manual approach, we can start by noticing that all computers look similar if we focus on them in isolation; they all use four outgoing links. The same is true of each of a computer's four neighbors. But what if we look at how the neighbors are connected to each other? Specifically, let us label a computer X via a multiset of four values Y_i, one for each computer linked to X, where Y_i is the number of other computers that are neighbors of both X and Y_i. For example, suppose that computer 1 is directly linked to computers 2, 3, 4, and 5, and that there are additional direct links 2-3, 2-4, and 4-5 (and all other links from 2, 3, 4, and 5 are to other computers). Then our multiset for computer 1 would be {1, 1, 2, 2}.

This labeling will not get us far enough in distinguishing computers, since there are not many possible labels of the form described above. But we can take this approach deeper! Let us label each computer as described above, and call those the level 1 labels. Then, we give each computer a level 2 label that is defined as the multiset of the level 1 labels of its neighbors — that is, each level 2 label is a multiset of four multisets. We can even go on to add a level 3 label that is the multiset of the neighbors' level 2 labels, and so on. It turns out that level 4 labels give us enough power to tell computers apart, at least within the 10 to 50 computer range. Moreover, since labeling a computer only requires looking at a computer's four neighbors and a finite number of possible connections between them, one round of the labeling process takes O(computers) time, as does the full process of getting the level 4 labels. This is more efficient than other possible approaches that, e.g., label a computer by the smallest or largest cycle it is part of.

Armed with either labeling method, we just need a way to create network designs to try them on. Generating edges at random has a really low probability of making all degrees exactly 4, so we need something better. One simple possibility is to shuffle a list of exactly 4 times each id, and link the first two ids, the third and fourth, and so on. If this creates a self-loop or a repeated link, repeat until it doesn't. The probability of self-loops or repeated edges is small enough for this to finish quickly for every size.

Another way to generate graphs that is less chaotic is to start with the ring design mentioned above, and then repeatedly mutate it as follows:

1. Randomly select a computer A and one of its neighbors B.
2. Randomly select a neighbor C of A and a neighbor D of B, such that C is not already a neighbor of B, and D is not already a neighbor of A.
3. Delete the links A-C and B-D, and add the links A-D and B-C.

This mutation method guarantees that each computer always has four links, and that the network remains connected. (Since we had C-A-B-D before and we have C-B-A-D after, any two computers that were directly or indirectly connected before are still connected.)

Most networks generated in this way (after, e.g., 1000 mutations) turn out to have unique labels on the computers. When we get one that does not (either because the network has some inherent symmetry, or because our labeling method isn't powerful enough for the network), we can just throw it away and try again. Our strategy should take at most a few seconds to discover usable network designs for all possible test cases, and then the rest is just implementation of interactions with the judge.

Finally, notice that, if necessary, graphs for all necessary sizes could be pre-computed offline and hardcoded into the solution. As long as the labeling method is efficient, it is OK for the process of finding the graphs to be a bit slow, as long as it finishes within the contest time! Nevertheless, all ideas presented above are fast enough to require no pre-computation.