Bits and Tree Analysis

Authors: Viktor Kozhuharov (problem), Emil Indzhev (analysis)

Intro

The problem is not trivial even if no extra node was added. Therefore, it is useful to think of the problem as having two parts – how to encode/decode at all and how to deal with error correction due to the added extra node. In some approaches the second part may be more interlinked with the first one, but as we'll see there is a neat approach that handles it mostly separately at the cost of some extra run time.

Naive ideas

Some possible naive ideas encode $O(\log(N))$ bits. For example, we could create a star graph with $1 \leq K \leq N-1$ leaves and attach all other nodes on a path from one of the leaves. Here, we have $N-1$ possibilities for K, so we can encode $\log(N-1)$ bits. To deal with the extra node, we could make sure K is odd and then we can deduce how to correct on the parities of the degrees of the nodes. Then we'd have $\log(N-1)-1$ bits. There are other approaches that would get to $c\log(N)$ bits (e.g. number of leaves at both ends of a path). However, we need $O(N)$ bits for more points.

Encoding $O(N)$ bits – $0.33N$

Now we'll look at our first "good" solution – one that encodes $O(N)$ bits. In order to not deal with difficult tree isomorphism problems, for most of our solutions we'll impose a strict structure on the generated tree. Since we are encoding a sequence, it is a good idea to create some (conceptually) directed path of nodes to serve as a skeleton.

To get started, we may want to mark the two ends with a bunch of extra nodes as their children/leaves (say 7 for the start and 5 for the end). That way, even with the extra added node, we can easily figure these two nodes out. Then we can simply encode 1 bit per node on the path (for simplicity, excluding the start and end nodes). We can do this by just adding nothing to the node, if the corresponding bit is 0, and adding 2 children to the node, if the bit is 1. Using this scheme we can easily detect the extra node and still decode the string. The worst case is when the string is all 1s, in which case we use 3 nodes per bit, so we get $0.33N$ bits (plus we actually have quite a few "wasted" marker nodes at the ends).

Better handling of error correction – $0.5N$

The above solution uses 2 extra nodes per bit (when the bit is 1). This is because we have a very naive error correction scheme. Instead, let's try marking 1s with just 1 child (in which case we can also reduce the number of extra children for the start and end by 1). The issue now is that, if the extra node is added to a 0 node, it will make it look like a 1 node.

We can deal with this by extending the bit sequence we will encode (or rather encoding a few fewer bits to leave space for this extension). Then we add $O(\log(N))$ extra bits as a checksum which will allow us to detect and correct each possible 1 bit flip (see Hamming codes). This scheme is more general, but it is still tied to our specific solution (we'll see an even more general one later).

Find this solution in `bits_ndiv2.cpp` – it encodes 85 bits and gets 45 points.

Randomizing the bit sequence – $0.67N$

The next issue we have is that 1s are more expensive than 0s, so if the the input bit string is all 1s, we get worse performance. What we can do is generate a fixed seed pseudo random bit string and just XOR the given bit string with ours. We do this when we encode and then when we decode. That way, we essentially guarantee that roughly 50% of the bits will be 0s and 50% will be 1s. Doing this randomization is a generally good idea for all such types of problem to guard against adversarially chosen data.

Find this solution in `bits_2ndiv3.cpp` – it encodes 102 bits and gets 55.57 points.

A better encoding idea – N

The above is basically the best version of that idea. The issue is that we have bits (1s) that take 2 nodes to encode. Instead we can do a slightly more complicated scheme. Each node on the skeleton path will represent a 0. Then we will do run length encoding of the number of 1s before it (since the last 0) using a side path coming out from it. I.e. just 0 is just a node, 10 is a node with a child, 110 is a node with a child with a child, etc. (We could alternatively do just 1 child per 1, all coming from the 0.)

Notice that this scheme takes 1 (skeletal) node per 0 bit and 1 (branch) node per 1 bit. Therefore, excluding extra $O(1)$ markers, it encodes N bits.

More general error correction

We now have to figure out how to deal with its error correction. For some errors, they will just result in a malformed tree and we can easily find where the extra node is. However, still some possible extra nodes will just change the decoded string. What's worse is that this won't result in just a bit flip. We may try to reason about the possible errors. However, it is better to see the most general way to deal with errors.

We can simply add a hash (with a configurable but high enough) number of bits. We hash the data (up to where we encode it) and then append the hash at the end (similar to the checksum described above). However, this hash is just some generic pseudo random hash function. Then when we have a proposed decoding, we simply check whether this hash checks out. If not, then it must have an error somewhere. However, unlike with the previous version, we cannot correct them. Instead, we can try all (valid) possibilities for the extra node, remove them, decode and then check. The hash has some chance for a false positive, so we need to add enough bits to make sure the chance is low enough.

Notice that this hashing idea is useful not just for dealing with the externally introduced errors (due to the extra node). We may also use it to deal with finding the start and end node. Instead of spending 10-15 extra nodes to mark the start and end of the path, we could also try each possible start and end (and removed node). Then decode each version (many will be malformed/invalid) and find the (hopefully) one where the hash matches.

This way we can trade the inefficient (in terms of number of nodes) markings for adding just a few extra bits to the hash. This will be way too slow, if done naively, so the author solution does some pre-filtering of possibilities using a few DFS passes. Additionally, it is possible to still have some partial markings (e.g. 1 node each) on the start and end (which we don't need here, but need to speed up the next solution).

Find this solution in `bits_emil_n.cpp` – it encodes 177 bits and gets 80 points.

An even better idea – $1.2N$

It is obvious that encoding with just chains or stars (coming out of a main path) is not optimal, as a general tree has many more possibilities. The final idea in the full author solution is to take advantage of the chains that come out of the main path (due to the 1s). If we have a chain of length 4 coming out of a node, it has 4 "slots" for extra bits (similar to our earlier $0.67N$ idea) – the main node and all non-final branch nodes. This is also why we use a chain instead of a star. So what we do is that first we encode the number of 1s and the final 0, then along the chain (if its length is X), we also encode the next X bits – nothing for 0 and an extra child along the branch for 1.

If we look at the string encoded from just the skeleton, 50% of its bits are 1s. Then for each 1, we encode an extra bit and "spend" and extra node 50% of the time (when the extra bit is 1). Let's say that the skeleton has K nodes, then the total number of nodes (up to $O(1)terms$) is $1.25K$ and the total number of bits is $1.5K$. Therefore, $K = 0.8N$ and thus the total number of bits is $1.2N$.

This adds more freedom to the tree, so makes pre-filtering for the start, end and extra node harder. Therefore, we need a few extra bits for the hash and/or the extra partial markings on the start and end.

Find this solution in `bits_emil_1.2n.cpp` – it encodes 205 bits and gets 100 points.

An alternative approach – general tree isomorphism

We can think about the problem more generally. We produce an unlabeled unrooted tree with N nodes and want to encode a bit string. If we ignore the error correction (which we can deal with using arbitrary hashes, as already described), one approach would be to just enumerate (assign IDs) to all possible such trees. On encoding, just create the tree whose ID is the bit string (interpreted in binary) and when decoding compute the ID of the tree and return the corresponding bit string. This should be able to encode about $1.56N$ bits.

The issue is simply that enumerating unlabeled unrooted trees is very hard (computationally intractable). Firstly, let's root the tree – this only loses us $\log(N)$ bits. We do this by trying each possible root (similar to how we handle the ends in the other solution). We also need to add some partial markers to restrict the number of possibilities for the root (to speed up the solution). Then if we use long integers, this enumeration problem becomes "just" a DP on the tree. To significantly simplify the implementation (and speed it up a lot), we only consider trees, such that for each node, all of its subtrees are of different sizes (this way we can "order" them for the encoding based only on their sizes). This actually reduces the constant for how many bits we can encode, but it is still powerful enough to be able to get (more than) 100 points.

Find this solution in `bits_and_tree_encho_iso.cpp` – it encodes 230 bits.