Code Jam 2021 - World Finals

Analysis: Slide Circuits

View problem and solution walkthrough video

To simplify the notation, we can reframe this problem in terms of graphs. We represent the input with a graph G that has one node per building and one directed edge per slide. The enabled/disabled states we can represent by subgraphs $G_1, G_2, \ldots, G_{\mathbf{N}}$ where G_i is a subgraph of G containing only the edges representing slides that are enabled after the first i operations.

A graph is fun if every node belongs to exactly one cycle. The question is, for each i, to identify an edge $(v, w) \in G - G_i$ such that $G_i \cup \{(v, w)\}$ is fun.

Test Set 1

The first step to solve the problem is to describe fun graphs more directly: A graph is fun if the in-degree and out-degree of every node is 1. Therefore, the edge (v, w) that we need in step i must be such that the out-degree of v in G_i is 0, the in-degree of w in G_i is 0 and all other in-degrees and out-degrees in G_i are 1. This implies that, given G_i, we can simply check the degrees and find the only possible candidate for v and w, if any. If $(v, w) \in G$, we found a answer. If there is no candidate for either, or $(v, w) \notin G$, then there is no answer for step i.

In Test Set 1, we can maintain the in-degree and out-degree of each node in the current G_i. When there is an enable operation, for each affected edge (v, w) we need to increase the out-degree of v and the in-degree of w by 1. For disable operations, we do the same but decreasing by 1. Then, we can do a linear pass to find candidates v and w. If there are unique candidates for both, we check if (v, w) is in G and give the appropriate output. This can be done in linear time in the size of the graph per step, which is $O(\mathbf{B} + \mathbf{S})$ time, or $O(\mathbf{N}(\mathbf{B} + \mathbf{S}))$ time overall, which is fast enough to pass Test Set 1.

Test Set 2

Our solution for Test Set 2 is an optimized version of the solution presented for Test Set 1. Consider multisets of nodes I_i and O_i. The number of occurrences of v in I_i is equal to the out-degree of v in G_i, and the number of occurrences of v in O_i is equal to the out-degree of v in G_i. Let ${I'}_e$ and ${O'}_e$ be the way the multisets I_i and O_i should look for e to be the answer after step i.

If we keep hashes of the I_i and O_i that are efficient to update and check a dictionary from the pairs of hashes of $({I'}_e, {O'}_e) \to e$, we can solve the problem. There are many options that work with different trade-offs in reliability, ease of implementation and ease of proof.

Sum of random values

Let us start by assigning a random integer x_v to each vertex v, that we keep throughout a test case. The hash of a multiset in this case is the sum of the values over all the vertices it contains (if it contains a vertex multiple times, its value is summed that many times), modulo some large number. For extra randomness we could use separate values for the I and O hashes.

Let $t = \sum_v x_v$ be the sum of all those random values. Then, the hash of ${I'}_{(v,w)}$ is simply $t - x_w$ and the hash of ${O'}_{(v,w)}$ is $t - x_w$. To get the hashes of ${I'}_{i+1}$ and ${O'}_{i+1}$ we can add or subtract

to the hashes of I'_i and O'_i (for simplicity, $I'_0 = O'_0 = 0$). The amount to add or subtract is the sum of the values of the starting/ending points of all edges that the operation is changing. We can find that efficiently by building two arrays (one for starting points and one for ending points) of sums over the first i multiples of M for each i and each M. Then, the sum between the i-th and j-th multiples of M is just the difference between the values for M, j and $M, i-1$. The array for a specific value of M contains $\lfloor \mathbf{S}/M \rfloor$ values, and $\sum_{M \leq \mathbf{S}} \lfloor \mathbf{S}/M \rfloor \leq \sum_{M \leq \mathbf{S}} \mathbf{S}/M = \mathbf{S} \sum_{M \leq \mathbf{S}} 1/M = O(\mathbf{S} \log \mathbf{S})$, so this is efficient enough.

It has hard to prove formally that the sums work well as hashes. We present next a closely related variant for which is much easier to be convinced that the probability of collisions is really small.

XOR of random values

The idea in this case is to use XOR instead of sum. The whole implementation can be done in the same way as in the previous case. However, because XOR is its own inverse, two multisets with the same parity in the number of occurrences for all vertices have the same hash. We can solve this by adding a count of the number of edges in G_i. This can still cause collisions, but since all numbers of occurrences in I' and O' are either 0 or 1, having both the correct parities and the correct total guarantees we are looking at the same multiset. This does require maintaining that total number of edges update, but that can be done in constant time per step.

In the case of XORs, each bit in the result is independent from all other bits. Since the values x_v are randomized, the probability of the bit being equal in two multisets with different parities is $1/2$. Therefore, the probability of 64 bits coinciding by chance is 2^{-64}, which is vanishingly small.

Polynomial hashes

Finally, in the XOR version above, we are actually hashing sets of nodes depending on the parity of their in- or out-degree, in a way. We know polynomial hashes are good for hashing sets, so we can simply use one of them. This is a third way of solving the problem.

Notice that a polynomial hash is actually equivalent to the sum version except the x_v values are powers of a prime instead of randomly chosen. Random choices are more resilient to adversary data and provide similar properties of uniformity of distribution. This is an informal argument that justifies the sum of random values being a good hashing for this problem.