15295 Spring 2019 #12 Games -- Problem Discussion
April 17, 2019

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. Vasya and Chess

Two words: Mirroring strategy.

B. The Game of {a, b, c} Forbidden Take-Away

We do DP to calculate the nimbers incrementally. Like the problem shows, we create a array holding the nimbers of all possible moves from the current pile count and with this we can easily find the mex. To do this fast, we find the nimbers incrementally and keep a histogram of all the previous nimbers. We start by zeroing the histogram because 0 has no moves, then we remove all the forbidden moves from the histogram. This potentially decreases the mex value. This finds the nimber of that pile size so we store it in our dp array. Then we add back in all the values we removed and also add in the new nimber we found so that the next pile size can have all the nimbers. We increment the mex until it reaches a point in the histogram that is nonzero.

The loop looks like this:

```
dp and histogram are all zeros arrays
(dp of length max pile size+1 and histogram of length max pile size+2)
mex = 0
For size = 0 to max pile size inclusive:
        For each forbidden if size-forbidden>=0: (remove duplicates)
                Remove dp[size-forbidden] from histogram
                (potentially update mex if histogram becomes 0)
        dp[size]=mex
        Add mex to histogram (potentially causing the value of mex to increase)
        For each forbidden if size-forbidden>=0: (remove duplicates)
                Add dp[size-forbidden] into histogram
                (potentially causing the mex value to increase)
```

Common bugs:
The most you can decrease the histogram by is 3 because the forbidden moves might map to the same nimbers even though they are different moves. This means it isn't just enough to check if removing the moves individually will cause the mex to decrease. You have to actually decrement all of them in the histogram to see if the mex changes (checking mex changes each time you decrement and increment should take this into account, I just wrote a helper function for increment and decrement)

--Matias

C. Furlo and Rublo and Game

This is a nim-type game where you have several piles and are allowed to remove some number of stones from any one pile, and the last player to move wins. So it can be solved by computing the nimber of all the piles and XORing them together. If the result is 0 then the player whose turn it is to move will lose, otherwise that player will win.

The only problem that remains is that of computing the nimber of each of the piles. The initial size of the pile is at most 10^(12), and rule is that if the pile is of size x, then after the move it will be of size y where $x^{(1/4)} \le y < \min(x, x^{(1/2)})$. Students came up with several ways to solve this. Here's one way. Let's make use of the posted hint that the nimbers never exceed 10.

So how do we compute the nimber of a pile of size x? We compute (look up) the nimber of all the sizes of the piles we can get to in one move, that is the nimbers of these

$$\{\lceil x^{(1/4)} \rceil, \ldots, \min(x-1, \lfloor x^{(1/2)} \rfloor)\}$$

and take their MEX. That is the nimber of x.

Now we know that after the first move the size of the pile must be at most 10^6. So suppose we pre-compute and store the nimbers of all such small piles. Now we need the MEX of a range of nimbers. There's a very elegant way to do this given that the nimber is known to be small. For each possible nimber {0,1,...,10} we compute a prefix sum, so P(i,k) tells you that the number of occurrences of a nimber i for x=0, 1, …, k. Now for a given range (a,b) we can see how many instances of a nimber i are in that range by evaluating P(i,b) - P(i,a-1).

So using these prefix sums we can count the number of 0s in the range, the number of 1's in the range, … the number of 10s in that range. The MEX is simply the smallest one of these for which there are zero of these.

So we use this method to compute the table of nimbers up to 10^6, (in time O(10 * 10^6)) then evaluate each of the given pile sizes in time O(10). Voilà

---DS

D. Game with Powers

This game can be seen as the sum of several subgames - the powers of non-power integers. For example, powers of i and powers of j never interfere with each other unless one is a perfect

power of the other. Say n=17; {{1}, {2,4,8,16}, {3,9}, (4 is counted already), {5}, …, {17}} is a set of mutually-noninterfering subgames. Each subgame only consists of powers of a specific number, so we only care about the exponents. Every move removes the numbers whose exponents are multiples of some number. Hence the base is irrelevant to the nimber. Note that the biggest subgame is always the powers of 2, whose size is log2(n), which doesn't exceed 29.

We can brute-force a list of nimbers for subgames with certain size k. For each subset of [k] (or [29]), we see it as the existing exponents at a state of the game. After a move (in that subgame) of choosing x^i, numbers i,2i,... are removed from this subset. This is the game tree. Calculating all these nimbers takes around 2^29*29 operations, which is horrible but indeed fine because we do this offline! So at the end we just have a hard-coded length-29 list of nimbers, telling given the size of a subgame (i.e. the number of such powers <= n) what its nimber is. At the runtime, just compute the sizes of each subgame and xor them up.

--Fei

E. A Game With Numbers

We can apply the technique for computing endgame tables in chess.

First of all, we're going to need a way to represent the state of the game. The state is a pair of hands, each with eight cards of type 0,1,2,3, or 4. There's no need to encode whose turn it is -- let the first of the pair be the hand of the player whose turn it is to move.

The representation that's supplied for a hand is a sequence of eight digits in [0,4]. This is highly redundant, because the order of the cards in the hand is completely irrelevant. (E.g. these hands are the same: [1,1,1,1,2,2,2,3] and [2,1,1,3, 2, 2,1,1], etc.) It's going to be important to keep the state space small, so let's see how to improve this representation.

Here's a good way to do it. Represent the hand by a histogram that stores the count of each of the five types of cards. So, the hand above is represented as [0,4,3,1,0], because there are zero 0s, four 1s, three 2s, one 3, and zero 4s. It's easy to see (via the pirates and gold method taught in 15-251) that the number of hands is 12 choose 4 = 495. So the number of pairs of hands is (states of the game) is 495*495 = 245025.

With this representation it's easy to write a move generator. This takes a game state and generates all the states available in one move. For example, if the state is

[0,2,2,2,2] [7,1,0,0,0]

Then the available moves are:

[7,1,0,0,0] [1,2,2,2,1]

[7,1,0,0,0] [0,2,2,1,3]
[7,1,0,0,0] [0,2,1,3,2]
[7,1,0,0,0] [0,1,3,2,2]

(Notice how the hands swapped, because after the move it's the other player's turn.)

Our goal is to label each state of the game with 1, 0, or -1, where 1 means the player to move has a winning strategy, -1 means the other player has a winning strategy, and 0 means there is no winning strategy for either player (it's a draw by repetition, to use chess terminology).

To that end we build a directed graph where the nodes correspond to the state (both hands), and there's an edge from state a to state b if there's a move in the game from state a to state b. (I used a hash table and an array to map back and forth between between the two-histogram game state and the vertex number of the graph.)

We keep an array state(v) for every vertex. First label all the terminal states with -1 or 1, depending on who wins. The rest of the states are initially all 0. Now we cycle through all the vertices in some fixed order. Suppose we're processing vertex v. If state(v) is non-zero we do nothing. If state(v) = 0 we do this:

> If all the vertices reachable by following an edge from v are 1, then state(v) ← -1.
> If one of the vertices reachable by following an edge from v is -1 then state(v) ← 1.
> Otherwise we leave the state(v) as it is (i.e. 0).

Eventually the system will stabilize and nothing will change. We detect this by counting the number of iterations since the last change, and when it equals the number of states we stop.

At this point, all the vertices are properly labeled, and we simply look up the answers for all the given queries.

Notice that although this solves the problem, it is not sufficient to make a robot that plays the game optimally. The problem is that your engine may know that the state is a winning state, but it does not know how to win in a finite amount of time. It may simply end up looping round a set of winning states. To avoid this, it's necessary to also label the winning states with the length of the game -- the number of moves (in optimal play) until the game ends. Then the robot player should always pick the move that reduces (or in fact minimizes) the number of moves to win. (It's also good to know this for the losing states so the robot can prolong the game as much as possible, even though it will ultimately lose.)

--Danny

F. An easy problem about trees

I don’t know how to solve this problem. But here’s an observation. If you can solve the case where the leaves are labeled only with 0 or 1, then you can solve the original problem by using binary search. --DS