Kick Start 2017 - Round C

Analysis: X Squared

X Squared: Analysis

Small dataset

One straightforward brute-force approach to this problem is to perform a breadth-first search of all grids that can be reached using the allowed swap operations, and see whether the target grid is reachable. However, this can take a while to code. Moreover, depending on the language used and the implementation details — e.g., how we choose to store previously seen states and avoid revisiting them — the search may run too slowly on the **N** = 5 cases. Perhaps a breadth-first search is overkill here; can we identify all of the reachable states in advance, without exploring?

Let's think about what swap operations *cannot* change. The set of tiles in a particular row in the initial grid will always remain together, in some order, in some row (not necessarily the row they started in). This is because a column swap only changes the relative order of those tiles, and a row swap only moves that whole set of tiles around together. So, our operations on the board cannot exchange individual tiles among different rows; they can only change the internal and relative orders of existing rows.

Because column swaps are the only way to change the internal order within a row, changing one row to have a certain order will give all of the rows that same internal order. However, we can choose whichever internal order we want: we pick the column with the tile we want to be first, and swap that column into the first column position, and so on.

All of the above logic holds for columns as well, and we can operate on the columns independently enough for our purposes: if we reorder the rows, and then reorder the columns, the latter operations do not change the chosen order of the rows. So, we can choose any of the **N**! possible orders for our **N** rows, and any of the **N**! possible orders for our **N** columns, for a total of $(\mathbf{N}!)^2$ different states that we can reach. (Some of these states might have the same pattern of `X`es and blank tiles, even if they represent different rearrangements of the original tiles.) Unlike in our breadth-first search method, we don't need to worry about exactly how to reach each of these states, even though the end of the previous paragraph explains how to do so. We know that it is possible to reach all of them, so if one of them matches the target, the answer is `POSSIBLE`. Moreover, we know that these are all of the states that we can reach, so if none of them matches the target, the answer must be `IMPOSSIBLE`.

Now, all we have to do is write some code to permute the rows and columns of a grid into whatever orders we want. To avoid doing too much duplicate work, we can pick one row permutation, then apply all possible column permutations to copies of that, then pick another row permutation, and so on. This method is essentially a well-targeted depth-first search, and its running time is $O((\mathbf{N}!)^2)$. This is easily fast enough for the Small dataset, in which **N** maxes out at 5. But a squared-factorial order of growth won't work for the Large dataset! Let's look for a better method.

Large dataset

Let's think about the target grid state. We can observe that it has one `X` tile that is the only `X` in its row and in its column; let's call this collection of `X`, row, and column the *singleton cross*. The rest of the grid contains a nested set of **N**/2 of what we will call *rectangles*. Each rectangle is defined by two rows and two columns, and has an `X` tile at each of the four intersections of those rows/columns, and no other `X` tiles anywhere else.

Suppose that we start at the target and perform some swap operations. What happens to our singleton cross and our rectangles? Per our earlier observation while solving the Small dataset, if two of the corners of a rectangle are together in the same row, or in the same column, they always will be; no row or column swaps can possibly split them apart. So swap operations cannot create or destroy rectangles, although they can change the relative positions of the four corners. Similarly, no operation can move additional `X` tiles into the singleton cross, or take away its one `X` tile.

These observations simplify the problem dramatically. We only need to ask: does the starting grid have exactly **N**/2 rectangles and exactly one singleton cross? If not, there is no series of swaps that can turn it into the target, which does have those properties, so the case is `IMPOSSIBLE` and we are done. Otherwise, we can transform the starting grid into the target grid as follows. Choose a rectangle, and perform row and column swaps such that its corners end up as the outermost corners of the larger X shape. Then recursively solve the rest of the grid as a subproblem, and so on. At the end of all this, the singleton cross will necessarily be in the correct place, since every other place will have been taken.

With this reframing of the problem in mind, we no longer need to actually perform or even think about any swaps! We only need to check the starting grid for rectangles and a singleton row/column. It is possible to do this with a single pass through the grid, looking at each row in turn. If a row has zero or more than two `X` tiles, it cannot be part of a rectangle or a singleton cross, and so the case is `IMPOSSIBLE`. If a row has one `X` tile, we need it to be part of the singleton cross; we store the column location of that `X`, and if we see another would-be singleton cross row later, the case must be `IMPOSSIBLE`. If a row has two `X` tiles, we note the columns of those two tiles. Once we have checked all rows, we check to see that every two-`X` row's set of `X` columns pairs up with exactly one other two-`X` row's set of `X` columns, and that no such pair shares any position with any other pair. Moreover, the position of the `X` in the singleton cross must be the sole column unused by any other pair. If all of these things are true, then the case is `POSSIBLE`.

This strategy is linear in the size of the input, and it runs fast enough to easily handle the Large dataset, as well as grids much larger than we provide in that dataset! It is also possible to run across greedy methods that carry out the swapping method we described earlier, or something that essentially boils down to it.