Code Jam 2021 - Round 3

Analysis: Square Free

Test Set 1

In general, we will build all grids with the correct row and column sums, then check each one to see if there are squares in them. In Test Set 1, the size of the input is at most 6×6. This means that there are 2^{36} different possible grids, which is too many to generate, so we must make use of the row sum constraints. We do this by building the grid one cell at a time. As you fill in each cell, ensure that corresponding row and column sums are still possible. For example, if the row sum should be 3, but there are already 3 `/`s in this row, you cannot put another `/`. Similarly, if the desired row sum is 3, but there are already $\mathbf{R} - 3$ `\`s in the row, you cannot put another `\`. Once we are done filling out the grid, we check if there are any squares in the grid that we have made. If there are no squares, we are done! Otherwise, we move on to the next possible grid. If we finish searching all possible grids and we have not found any, then it is impossible.

Are we sure this is fast enough? Each row either needs 0, 1, 2, 3, 4, 5, or 6 `/`s in it. There are $\binom{6}{0} = 1$ choices with 0 `/`s, $\binom{6}{1} = 6$ choices with 1 `/`s, $\binom{6}{2} = 15$ choices with 2 `/`s, $\binom{6}{3} = 20$ choices with 3 `/`s, $\binom{6}{4} = 15$ choices with 4 `/`s, $\binom{6}{5} = 6$ choices with 5 `/`s, and $\binom{6}{6} = 1$ choices with 6 `/`s. Thus, each row has at most 20 different valid choices. So this algorithm will explore at most 20^6 different grid. In practice, we will get nowhere near this bound. In particular, there is at most one valid row for the bottom row (since we must satisfy the column sum constraints). This alone reduces the search space to at most 20^5, but this, too, is an overestimate since there will be a lot of pruning throughout the search.

Checking for squares can be done in several ways. The easiest is to iterate over all possible places that the top-most row of the square can be (and which consecutive columns the `/\` are in). Then, for each possible size of square (1, 2, 3), just check the corresponding grid entries on the four sides of the square.

Test Set 2

The bounds for Test Set 2 are much too large to exhaustively search all grids, so we will need an insight. We call a grid that satisfies the row and column constraints a *configuration*. First, let's discuss how to find *some* configuration (it may or may not be square free). To do this, we will set this up as a graph and run maximum flow on it. There are $\mathbf{R}$ vertices that represent the rows and $\mathbf{C}$ vertices that represent the columns. We will put an edge with capacity 1 between every pair of (row, column) vertices. We will then connect the i-th row vertex to a super-row vertex with capacity $\mathbf{S_i}$ and connect the i-th column vertex to a super-column vertex with capacity $\mathbf{D_i}$.

We now run flow through the network using the super-row vertex as the source and the super-column vertex as the sink. If the network is saturated (that is, every edge leaving the source has flow = capacity), then we have a solution. If there is flow in the edge between row r and column c, then the corresponding entry in the grid is a `/`, otherwise it is a `\`. If the flow is not saturated, then it is impossible to make a grid with the appropriate row and column sums. We leave a formal proof of the bijection between configurations and valid flows on the described network as an exercise.

At this point, we have *some* configuration, but it may have squares in it. We will discuss three different ways to produce a grid that is square free.

Lexicographically Smallest Configuration

In this solution, we notice that the lexicographically smallest configuration (treating $\backslash$ as smaller than $/$ reading in row-major order) is square-free! At first this is not obvious. However, think about two rows ($r_i < r_j$) and two columns ($c_k < c_\ell$). If $(r_i, c_k) = /, (r_i, c_\ell) = \backslash, (r_j, c_k) = \backslash, (r_j, c_\ell) = /$, then this grid is not the lexicographically smallest configuration. Why? Because we can swap all four of those without breaking the row or column constraints, while giving us a smaller configuration ($(r_i, c_k) = \backslash, (r_i, c_\ell) = /, (r_j, c_k) = /, (r_j, c_\ell) = \backslash$). This means that a lexicographically smallest configuration has no squares in the grid, because the top-most row of a square must contain $/\backslash$ in the same columns in which the bottom-most row of the square has $\backslash/$.

How do we find the lexicographically smallest configuration? There are two ways: (1) give the edge between row r and column c a cost of $2^{r-1+(c-1)\mathbf{R}}$ and run minimum-cost maximum-flow. This will guaranteed find the lexicographically smallest, but the edge-costs will be huge (up to $2^{\mathbf{RC}-1}$). (2) We will run maximum flow iteratively. Say we have run maximum flow. Go through the edges in row-major order of their corresponding cell. If there is no flow going through an edge, then it is a $\backslash$. Great! This is the smallest this can be, so remove this edge from the graph to lock that in. If there is flow going through the edge, then we "unpush" the flow from that edge (decrease the flow from the sink to the column vertex to the row vertex to the source by 1) and temporarily change that edge's capacity to 0. We then run flow again. If the flow is still saturated, then we know there exists a configuration where this entry in the grid is a $\backslash$, so we can permanently remove this edge from the graph. If it is not saturated, then we must put that edge back into the graph, and this entry is forced to be a $/$.

Complexity-wise, the first run of maximum flow takes $O((\mathbf{RC})^2)$ time. Then, for each flow we run, we must only push one unit of flow through, which only takes a single augmenting path, so $O(\mathbf{RC})$ time per edge. Thus, in total, this takes $O((\mathbf{RC})^2)$ time. Note that you can also fully re-run maximum flow for each edge instead of only pushing one unit of flow with a sufficiently optimized flow implementation.

Minimum-Cost Maximum-Flow

In the Lexicographically Smallest Configuration solution, we described how you can use minimum-cost maximum flow to solve this problem with exponential edge costs. Here, we will solve the problem using only polynomial sized edge costs. This solution makes use of the same idea as the previous one: we want to avoid $(r_i, c_k) = /, (r_i, c_\ell) = \backslash, (r_j, c_k) = \backslash, (r_j, c_\ell) = /$, but other than that, we do not need a lexicographically smallest configuration. If we set the edge cost between row r and column c to be $r \times c$, then we will not get this configuration, since swapping all of these symbols does not affect the row/column sums and strictly decreases the total cost. The total cost is reduced by $i \times k + j \times \ell$ and increased by $i \times \ell + j \times k$, which is a net decrease of $(i - j)(k - \ell) > 0$ since $i < j$ and $k < \ell$.

Flip-Flop!

Another solution is to simply find some configuration, then check it for squares. If there are no squares, then we are done! If there is a square, consider the top-most and bottom-most row in the square. We will swap the top $/\backslash$ with the bottom $\backslash/$. This does not affect the row/column sums. In doing this, we have broken the current square, but may have created another square. We continue breaking squares until we do not find any. This process must eventually finish since at each swap, we are always making our grid lexicographically smaller. You can never do more than $O((\mathbf{RC})^2)$ swaps of this form.

Common Mistake

Be careful! Just because the sum of the $\mathbf{S_i}$s is equal to the sum of the $\mathbf{D_i}$s does not mean that a configuration exists! One example is the following input, where those sums coincide, yet there are no grids that meet all the per-row and per-column requirements.

```
4 6
2 0 6 6
4 2 2 2 2 2
```