Code Jam 2019 - Round 3

Analysis: Datacenter Duplex

We can start by modeling the problem as a graph where each cell of the input matrix represents a node. Orthogonally adjacent cells with the same label are connected by edges, and we can optionally add edges connecting diagonally adjacent cells with the same label. The goal is to end up with exactly two connected components.

Test set 1

Test set 1 can be solved with dynamic programming, iterating over columns. When considering the i-th column from left to right, we can summarize the state of the connections we have added in the submatrix S of columns 1 through i by recording: (1) whether we have seen (at least one cell of) each of A and B so far, and (2) whether any two cells on the i-th column that contain the same label but are not orthogonally adjacent are connected by a path in S. Then, we can use brute force, and consider all possible choices of how to connect each of the up to 3 cell corners between columns i and i+1 (there are at most 4 rows in Test set 1). There are 3 possibilities for each corner, \, / and no connection, but it can be reduced by noticing if at least one of the connections is valid, it is always optimal to use one, which reduces the number of choices per corner to 2 at the most. If at any point we discover a new isolated A component that cannot be connected to previously seen As, we have failed, and same is true for B. If we finish, we can then use a second pass to reconstruct one choice of diagonal connections that led to solving the problem.

This seemingly simple idea has several technical details that we are omitting here. Some of them can be simplified by starting and ending the process in columns that contain both As and Bs and preprocessing to check if any leftmost or rightmost columns that contain only one type already disconnect the other type.

The overall time complexity of this solution is exponential in **R**, because we are trying all combinations of O(**R**) corners and recording the connected status of O(**R**) cells at each state, and linear in **C**, with the exact formula depending on how the technical details are handled. As long as the base of the exponential factor is not too large, this is fast enough to pass within the time limit.

Test set 2

The key observation is: after we decide to add some edges (diagonal adjacencies), two cells c and d containing the same label X end up separated regardless of any future decisions if and only if one of the following two conditions holds:

- 1. There is a cycle of cells labeled Y ≠ X in the graph, with one of c and d being inside the cycle and the other one being outside.
- 2. There is a path of cells labeled Y ≠ X in the graph with the first and last cells of the path being border cells of the matrix, with c and d being on opposite sides of the path.

Let us use G to denote the graph with no diagonal edges added and H to denote the final graph with all edges added. Consider the border of the matrix. Suppose that it contains two cells c and d labeled X that are not connected in G. Since they are not connected in G, that means, going around the border, there are cells e and f labeled Y ≠ X such that e is in between c and d in clockwise order and f is between c and d in counter-clockwise order. That means neither c and d nor e and f can be connected in H with a path of only border cells. Therefore, if c and d are connected in H, the path connecting them disconnects e and f, and vice versa. So, by the second condition, having two cells on the border with the same label that are disconnected in G results in an impossible case.

Notice that there is never an incentive to generate an edge from a diagonal adjacency to connect two cells that are already connected. Per the paragraph above, if a case is possible, then all border cells with the same label are already connected in G. Therefore, if an algorithm never adds an edge from a diagonal adjacency that connects two cells that are already connected, it will never generate a path between two border cells. Additionally, if we never connect cells that are already connected, any cycle in H is a cycle that was already present in G, so again, if we end up in a disconnected situation, the case must have been impossible from the start, before we added any connections.

These observations directly suggest the following algorithm: consider each diagonal adjacency and generate an edge if and only if it will connect two previously disconnected cells. If both choices work, we can choose either, since we already established that the decision of not connecting previously connected cells is enough to guarantee an algorithm will not generate an H with more than 2 connected components when a different one with exactly 2 was possible. After this process, check if there are 2 connected components or more than 2, and print the results.

If we implement the algorithm above using a union-find to maintain the connected components, we need O(**R** × **C**) checks for whether two cells are connected and O(**R** × **C**) connections (joins). Since union-find provides almost constant amortized time for both operations, the overall time complexity of the algorithm is quasilinear.

An equivalent algorithm is to calculate the connected components of G first and then use shortest paths to join any two components of the same label until we cannot do it anymore. Since shortest paths cannot create new cycles, an argument similar to the one above proves that this solution is also correct. This solution can be implemented in linear time if the partial minimum path tree graph is reused for each new connection we need.