Code Jam 2012 - World Finals

Analysis: Xeno-archaeology

To begin with, let's try to formalize the rules of forming the pattern of tiles. If the center is at some position **x**, **y**, then the red tiles are those in positions **x'**, **y'** for which the number max(|**x** - **x'**|, |**y** - **y'**|) is odd, while the blue tiles are those for which this number is even. This is because the formula max(|**x** - **x'**|, |**y** - **y'**|) = **C** for any **C** describes a square ring around **x**, **y**, and the rings alternate color with the parity of **C**

For the small problem, we can prove that if there exists a solution, there exists one with |**X**| + |**Y**| < 202. Thus, we can check all candidates for the center, for each one check whether all the tiles have the right colors, and output the best candidate. This will not run in time for the large data, of course, as we will have over 10^{30} candidate centers to check.

Single tile analysis

We may now assume we know the parity of **x** and **y**. We will simply check all four possibilities, finding the best possible choice of the center for each of the four assumptions, and then pick the one specified by the tie-breaking rules (or output "Too damaged" if none of the four assumptions led to finding a viable center for the pattern). This makes it easier to analyse the information gained by knowing the color of a single tile. Suppose the tile at some position **x'**, **y'** is, say, red. This means max(|**x** - **x'**|, |**y** - **y'**|) has to be odd. Now, we know the parities of **x**, **y**, **x'** and **y'**, and so:

- if both **x** - **x'** and **y** - **y'** are odd, then any choice of a center (satisfying the parity requirements for **x** and **y**) is going to fit our knowledge;
- if both **x** - **x'** and **y** - **y'** are even, then there is no solution satifying the parity requirements;
- if, say, **x** - **x'** is odd, while **y** - **y'** is even, we have to have |**x** - **x'**| ≥ |**y** - **y'**|

If there is any tile of the second type, we can immediately return "Too damaged" for these parity assumptions. We can ignore of tiles of the first type, and now we are left only with tiles of the second type.

Note that in the third case, since the parities of **x - x'** and **y - y'** differ, it doesn't matter whether we use a strict inequality, as the equality case is eliminated from consideration by the parity assumptions. Thus, when considering regions defined by these inequalities, we can ignore problems related to "what happens on the edges of these regions", as - by the reasoning above - the edges will necessarily be eliminated from consideration by the parity assumptions.

The first and second cases are easy to analyse; the trick is to find out whether a solution exists (and if yes, find the best one) satisfying the set of conditions of the type |**x** - **x'**| ≥/≤ |**y** - **y'**| for various **x'** and **y'**. Transforming the condition |**x** - **x'**| ≥ |**y** - **y'**| we see it is equivalent to saying that one of the following has to hold:

- **x + y ≥ x' + y'** and **x - y ≥ x' - y'**, or
- **x + y ≤ x' + y'** and **x - y ≤ x' - y'**.

Dividing the plane

The lines $\mathbf{x + y = x_i + y_i}$ and $\mathbf{x - y = x_i - y_i}$ (which are the boundaries of the constraint-satisfaction region for the input tiles) divide the plane into at most $(\mathbf{N} + 1)^2$ rectangles. The idea of our algorithm will be as follows:

- Iterate over the four sets of parity assumptions about the center
- Iterate over all rectangles formed by the boundary lines, and for each of them check whether it satisfies the constraints posed by all input tiles
- for each rectangle satisfying the constraints, find the best (according to the tie-resolution conditions) center candidate within it (if any)
- output the best of all center candidates found.

A fun fact is that there will be at most **N**+1 rectangles that satisfy the constraints; so we need not worry overly about the performance of the "find the optimal within the rectangle" phase (as long as it is independent of the size of the rectangle). The naive approach to the second phase is $O(\mathbf{N}^3)$ (for each rectangle check all tiles), which with **N** up to a thousand and 50 testcases risks being too slow, so we'll need to speed it up a bit.

There are many ways to trim down the runtime of the constraint-checking phase for rectangles. One sample way is to process the rectangles "row-by-row", as follows: Take the set of rectangles with $\mathbf{A} \le \mathbf{x+y} \le \mathbf{B}$, with **A** and **B** being some two adjacent boundary values. For each input tile (out of those that set any constraints on the center position), we have two areas of constraint satisfaction; but only one of them is compatible with $\mathbf{A} \le \mathbf{x+y} \le \mathbf{B}$, because one of the areas satisfies the constraint $\mathbf{x+y} \ge \mathbf{C}$, while the other has $\mathbf{x+y} \le \mathbf{C}$. This means that we know which area is the interesting one for this row; so we obtain a constraint on $\mathbf{x - y}$ that has to be satisfied by all the rectangles in this row. This will be either of the form $\mathbf{x - y} \le \mathbf{D}$, or $\mathbf{x - y} \ge \mathbf{D}$. We take the largest of the lower bounds, the largest of the upper bounds, and obtain a rectangle that we have to check. This algorithm runs in $O(\mathbf{N}^2)$ time, which will be easily fast enough.

A more advanced algorithm (using the sweep line technique) can be used to obtain a runtime of $O(\mathbf{N} \log\mathbf{N})$ runtime. We will not describe it (as it is not necessary to obtain a fast enough program with the constraints given), but we encourage the reader to figure it out.

Finding the best point within a rectangle

This was the part of the problem that seemed to cause most difficulties for our contestants. There are two cases to consider here. Let's assume our rectangle is defined by $\mathbf{A} \le \mathbf{x+y} \le \mathbf{B}$ and $\mathbf{C} \le \mathbf{x-y} \le \mathbf{D}$.

Let us define
g(k, l) = min(|k|, |l|) if k and l are of the same sign, 0 otherwise.
If g(**A**, **B**) = 0 and g(**C**, **D**) = 0, then the point (0, 0) is within our rectangle. In this case it suffices to check the near vicinity of the origin. Specifically:

- If both **x** and **y** are supposed to be even, (0, 0) is obviously the optimal solution.
- If both **x** and **y** are supposed to be odd, then the best four points, in order, are (1, 1), (1, -1), (-1, 1) and (-1, -1). If $\mathbf{B} \ge 2$ we can take (1, 1) and we're done. Otherwise, if $\mathbf{D} \ge 2$, we take (1, -1); and so on. If all the four points are infeasible, the rectangle contains no points satisfying the parity constraints.
- If , say, **x** is supposed to be odd, while **y** is even, the first eight candidates are (1, 0), (-1, 0), (3, 0), (1, 2), (1, -2), (-1, 2), (-1, -2), (-3, 0). Again, one can check that if none of them is feasible, the rectangle contains no points satisfying the parity constraints. The same happens when **x** is even and **y** is odd.

Thus, if (0, 0) is within the rectangle, we can check a constant number of points and take the best feasible one of them.

When (0, 0) is not within the rectangle, we first look for the smallest Manhattan distance of any point within the rectangle. It is equal to **M** := max(g(**A**, **B**), g(**C**, **D**)). As all the boundaries have parities disagreeing with the parity assumptions, the smallest Manhattan distance we can hope for is **M** + 1. We now have an interval of points with Manhattan distance **M** + 1 in our rectangle, the best one of them is the one with the highest **X** coordinate (out of the ones fulfilling the parity conditions). The one last special case to deal with here is when the interval contains only one point, and it has the wrong parities - in this case we need to look at distance **M** + 3 (the fact that one never needs to look at **M** + 5 is left as an exercise).

It was also possible to solve this problem in a number of other ways. A pretty standard one was to identify a number of "suspicious points" within a rectangle (the vicinity of (0, 0), the vicinity of the corners, and the vicinity of places where the coordinate axes intersect the edges of the rectangle) and check them all, taking the best solution.