Solution — Suez

1 The Problem in a Nutshell

We are given a set of equally sized and pairwise disjoint axis-parallel rectangles in the plane, some of them red and some of them blue. How can we scale up the red rectangles—while maintaining their center points and aspect ratios—in order to maximize their sum of perimeters?

2 Modeling

As already hinted above, we model the posters as pairwise disjoint axis-parallel rectangles in the Euclidean plane, with center points defined by the corresponding locations of the nails. We assume that the rectangles corresponding to new posters are colored red, whereas the rectangles corresponding to old posters are colored blue. Thus, if w and h denote the original width and height, then we see that a blue rectangle with center point (x, y) covers the area

$$\left[x - \frac{w}{2}, x + \frac{w}{2}\right] \times \left[y - \frac{h}{2}, y + \frac{h}{2}\right], \tag{1}$$

whereas a red rectangle with center point (x, y) and scaling factor $a \geqslant 1$ covers the area

$$\left[x - \frac{aw}{2}, x + \frac{aw}{2}\right] \times \left[y - \frac{ah}{2}, y + \frac{ah}{2}\right]. \tag{2}$$

We assume that the red rectangles are indexed by the integers $1, \ldots, n$, and the blue ones by $n+1, \ldots, n+m$. Our goal now is to find a linear programming formulation with variables $a_1, \ldots, a_n \geqslant 1$, i.e., the unknown scaling factors of the red rectangles. As follows, we can formulate constraints that enforce pairwise disjointness[1] of the rectangles. For two red rectangles with indices i and j, we see (in Figure 1) that they are disjoint if and only if

$$(a_i + a_j) \cdot \frac{w}{2} \leqslant |x_i - x_j| \qquad \text{or} \qquad (a_i + a_j) \cdot \frac{h}{2} \leqslant |y_i - y_j|. \tag{3}$$

Similarly, for a red rectangle with index i and a blue one with index j, we see (simply by plugging in $a_j = 1$ in the constraint above) that they are disjoint if and only if

$$(a_i + 1) \cdot \frac{w}{2} \leqslant |x_i - x_j| \qquad \text{or} \qquad (a_i + 1) \cdot \frac{h}{2} \leqslant |y_i - y_j|. \tag{4}$$

At first sight, the above constraints should be slightly disconcerting for the following reason: It is impossible to express "or"-constraints in a linear program in general. But fear not, the two

[1] Whether by "disjoint" we mean only interior-disjoint or also boundary-disjoint is irrelevant for the purpose of this exercise, as the final answer is rounded up to the nearest integer anyway. To be able to formulate our conditions nicely as non-strict inequalities, however, we adopt the convention that we mean interior-disjoint.

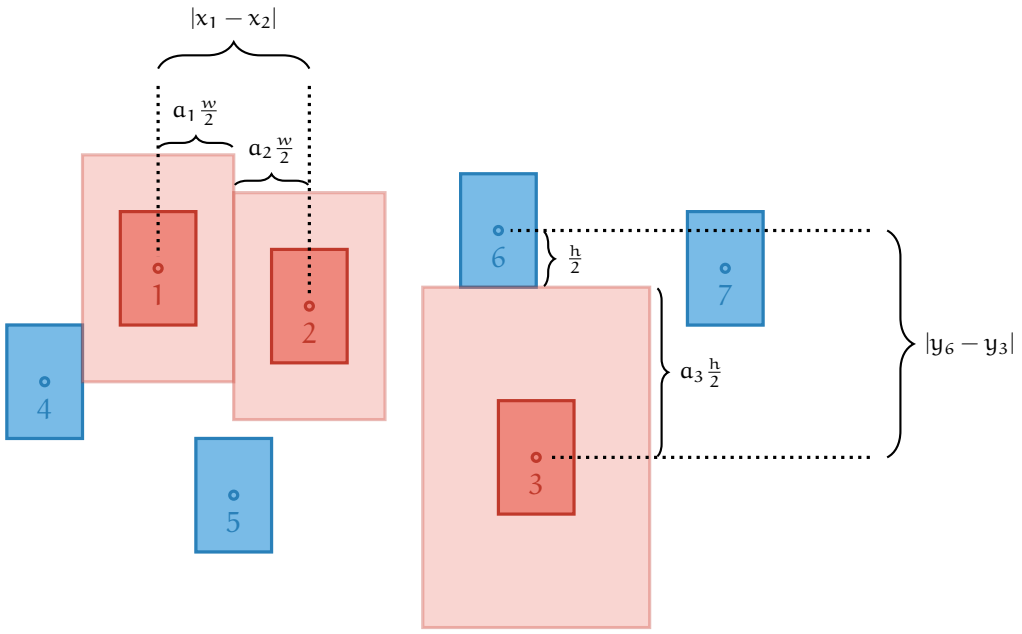

Figure 1: Pairwise disjoint blue and (scaled) red rectangles.

inequalities in (3) (and also in (4)) are of a very special nature. The two hyperplanes in $\mathbb{R}^n$ that correspond to these two inequalities are parallel, which means in particular that one of them is redundant. This becomes even more evident if we multiply the inequalities by $\frac{2}{w}$ and $\frac{2}{h}$, respectively. We then see that (3) is equivalent to

$$a_i + a_j \leqslant 2 \cdot \max\left(\frac{|x_i - x_j|}{w}, \frac{|y_i - y_j|}{h}\right). \tag{5}$$

In the same way, (4) is equivalent to

$$a_i \leqslant 2 \cdot \max\left(\frac{|x_i - x_j|}{w}, \frac{|y_i - y_j|}{h}\right) - 1. \tag{6}$$

In our linear program we will thus have $\binom{n}{2}$ constraints of type (5), one for each pair of red rectangles. Similarly, the natural number of constraints of type (6) would be nm, one for each pair consisting of a red and blue rectangle. However, this naive approach unnecessarily blows up the number of constraints. Instead, we observe that for any red rectangle with index i we only have to add one constraint of type (6), namely the one coming from the blue rectangle with index j that imposes the strongest upper bound on a_i. Lastly, we need not add any constraints for pairs of blue rectangles, since they are disjoint anyway by the assumption given on the exercise sheet.

As a final note, the perimeter of a scaled red rectangle with index i can be expressed simply as the product $2(w + h) \cdot a_i$. The objective function of our linear program will therefore be the

sum over all these products:

$$\begin{aligned}
\text{maximize} \quad & 2(w+h) \cdot \sum_{i=1}^{n} a_i \\
\text{s.t.} \quad & a_i + a_j \leqslant 2 \cdot \max\left(\frac{|x_i - x_j|}{w}, \frac{|y_i - y_j|}{h}\right) && \forall i \neq j \in 1, \ldots, n \\
& a_i \leqslant 2 \cdot \min_{j=n+1,\ldots,n+m} \left\{ \max\left(\frac{|x_i - x_j|}{w}, \frac{|y_i - y_j|}{h}\right) \right\} - 1 && \forall i \in 1, \ldots, n \\
& a_i \geqslant 1 && \forall i \in 1, \ldots, n
\end{aligned}$$

3 Algorithm Design

There is not much to say in terms of algorithm design for this problem; it is clear that the LP solver will do most of the heavy lifting for us. Given that $n \leqslant 30$ and $m \leqslant 10^3$, finding and constructing all the relevant constraints can be done simply by exhaustive search in time $O(n(n+m))$. After that, in the worst case we have a linear program with $n = 30$ variables, which is feasible according to our rule of thumb.

However, using the naive approach of including a constraint for every pair of red and blue rectangles, we would obtain $\binom{n}{2} + nm + n \approx 30'000$ constraints in the worst case, which is rather excessive and which would only be rewarded with 60 points on the judge. In order to get all 100 points, we first have to bring down the number of constraints to $\binom{n}{2} + 2n \approx 500$ in the worst case, as explained in the previous section.

4 Implementation

The implementation for this problem is also straight forward. There are only three small things that we would like to mention.

Input type Since some of the coefficients occurring in (5) and (6) are rational numbers, we have to be careful and use a quotient type such as `CGAL::Gmpq` when constructing our linear program. For this reason, we use the following headers and typedefs.

```
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpq.h>
typedef CGAL::Gmpq ET;
typedef CGAL::Quadratic_program<ET> program_t;
typedef CGAL::Quadratic_program_solution<ET> solution_t;
```

Alternatively, we could also try to figure out on a case-by-case basis which one of the two inequalities in (3) (and similarly in (4)) is less restrictive, and then only use that one. After multiplying by 2, this would allow us to use a native fixed-precision integer as the input type, which would noticeably improve the observed running time on the judge. However, for ease of presentation and since it is enough to earn 100 points, we will stick to the quotient type as explained above.

Output type It is also a good idea to check how big the objective value of our linear program can become. First, observe that the variables a_i cannot grow without bounds because, according to the exercise sheet, there are at least two red rectangles ($n \geqslant 2$) which will block each other's growth. A moment of reflection reveals that we have in fact $a_i < 2^{25}$ always; the worst case being attained for example if $x_i = 2^{24} - 1$, $x_j = -2^{24} + 1$ and $w = 1$. Therefore, the maximum sum of perimeters can never be larger than

$$2(\underbrace{w}_{<2^7} + \underbrace{h}_{<2^7}) \cdot \sum_{i=1}^{n} \underbrace{a_i}_{<2^{25}} < 2^{34} \cdot \underbrace{n}_{\leqslant 30} < 2^{39}.$$

Hence, while it is perfectly fine to store our result as a `double` with 53-bit mantissa before outputting it, we would potentially run into trouble if we were to store it as a 32-bit `int`.

Lower and upper bounds Recall that the lower bound for each variable is simply $a_i \geqslant 1$, whereas the upper bound is given by equation (6) for the most restrictive blue poster with index j. While it is possible to input these bounds by means of the methods `set_a` and `set_b`, it is recommended to either use the constructor of `CGAL::Quadratic_program` or one of the methods `set_l` and `set_u` instead. While equivalent as far as correctness is concerned, the latter is noticeably more efficient on the judge.

The reason is that the first method appends a complete, albeit very sparse row to the constraint matrix A. As a general rule of thumb, you can assume that the larger this constraint matrix is, the more work the LP solver has to do internally. In particular, simply reading one constraint in that matrix, which might happen multiple times during the execution of the LP solver, takes time linear in the number n of variables. On the other hand, reading a constraint that has been input with `set_u`, say, takes only constant time since it is stored as a single number in memory.

5 A complete solution

```
#include <iomanip>
#include <iostream>
#include <vector>
#include <CGAL/QP_models.h>
#include <CGAL/QP_functions.h>
#include <CGAL/Gmpq.h>

typedef CGAL::Gmpq ET;
typedef CGAL::Quadratic_program<ET> program_t;
typedef CGAL::Quadratic_program_solution<ET> solution_t;
typedef CGAL::Quotient<ET> objective_t;

double ceil_to_double( objective_t const & o )
{
  double d = std::ceil( CGAL::to_double( o ) );
  while( d   <   o ) d += 1;
  while( d-1 >=  o ) d -= 1;
  return d;
}

struct coord
{
```

```
  int x, y;
};

void test_case()
{
  // Read Input
  int n, m, h, w;
  std::cin >> n >> m >> h >> w;
  std::vector<coord> red_rects( n );
  std::vector<coord> blue_rects( m );
  for( int i = 0; i < n; ++i )
  {
    std::cin >> red_rects[i].x >> red_rects[i].y;
  }
  for( int i = 0; i < m; ++i )
  {
    std::cin >> blue_rects[i].x >> blue_rects[i].y;
  }

  // Set Up Linear Program
  program_t lp( CGAL::SMALLER, true, 1, false, 0 );

  // Objective Function
  for( int i = 0; i < n; ++i )
  {
    lp.set_c( i, -2 * (w + h) );
  }

  // RED-RED Constraints
  int ids = 0;
  for( int i = 0; i < n; ++i )
  {
    for( int j = i+1; j < n; ++j )
    {
      int id = ids++;
      ET dx = std::abs( red_rects[i].x - red_rects[j].x );
      ET dy = std::abs( red_rects[i].y - red_rects[j].y );
      lp.set_a( i, id, 1 );
      lp.set_a( j, id, 1 );
      lp.set_b( id, 2 * std::max( dx/w, dy/h ) );
    }
  }

  // RED-BLUE Constraints
  for( int i = 0; i < n; ++i )
  {
    ET limit = 33554432; // safe because from Section 4 we know that a_i < 2^25
    for( int j = 0; j < m; ++j )
    {
      ET dx = std::abs( red_rects[i].x - blue_rects[j].x );
      ET dy = std::abs( red_rects[i].y - blue_rects[j].y );
      ET current_limit = 2 * std::max( dx/w, dy/h ) - 1;
      limit = std::min( limit, current_limit );
    }
    lp.set_u( i, true, limit );
  }

  // Call Solver
  solution_t s = CGAL::solve_linear_program( lp, ET() );
```

```
  std::cout
    << std::setprecision( 0 ) << std::fixed
    << ceil_to_double( -s.objective_value() ) << "\n";
}

int main()
{
  std::ios::sync_with_stdio( false );
  int t; std::cin >> t;
  while( t-- ) test_case();
}
```