Solution — Clues

1 The Problem in a Nutshell

Given a set of congruent disks (radio stations with a given operation range r) in the plane, is it possible to assign one of two colors (frequencies) to each disk in such a way that there is no blue (red) disk whose center point is contained in another blue (red) disk? If so, for two given points in the plane (Holmes and Watson), are the two respective closest disks connected? Here, connected means that there is a sequence of disks between them such that any two consecutive disks overlap in their respective center points.

2 Modeling

It is natural to model the radio stations as a conflict graph $G = (V, E)$. That is, V is simply the set of center points of the radio stations, and an edge is added to the set E whenever two radio stations are at distance at most r. Thus, note that any edge in E is a constraint in the sense that we have to assign distinct frequencies to the two corresponding radio stations.

An example of such a conflict graph G can be seen in Figure 1. In that figure, the radii of the disks are assumed to be exactly equal to the operation range r, and hence the colored regions are those where at least one radio station is in range. Alternatively, it is also possible to define G as a disk intersection graph—as illustrated in Figure 2—where two disks stand in conflict if and only if they intersect. However, to make it work out the radii of those disks have to be set to $r/2$. In that sense, Figure 2 is misleading because the colored regions do not cover all points where at least one radio station is in range. On the other hand, identifying the conflicts might be a bit easier than in Figure 1.

Having such a graph G, the first part of the problem (deciding whether there exists a network without interferences) reduces to deciding whether G is 2-colorable. The second part of the

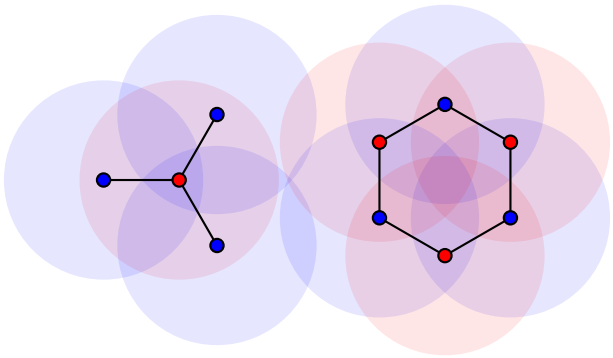

Figure 1: Ten radio stations with highlighted operation ranges, forming a network without interferences. The black edges are the edges in the corresponding conflict graph G.

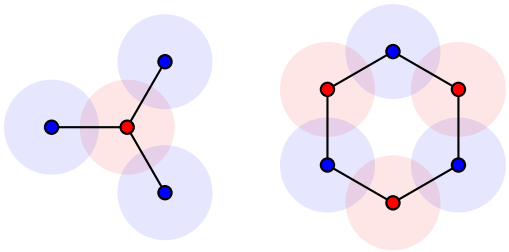

Figure 2: The conflict graph G when seen as a disk intersection graph.

problem (deciding whether Holmes and Watson can communicate) reduces to deciding whether the respective closest radio stations are in the same connected component of G.

3 Algorithm Design

The task is clear: Find a 2-coloring of G if one exists and determine the connected components in G. Of course, one way to do this is to first compute an explicit representation of G and then run some well-known algorithms on that explicit representation. However, this approach is not good enough because the number of edges in G can be as large as $\Theta(n^2)$, where n is the number of radio stations. It might be good enough for the first group of test sets, but for the others we need a better idea.

A moment of reflection reveals that we do not really need to know all of G. A *spanning forest* F in G is already enough, by which we mean a collection of spanning trees for each of the connected components in G (see Figure 3 for an example). Indeed, if connected components of G are also connected in F, we can easily determine these components, say, by performing a depth-first search on F. Furthermore, if a 2-coloring exists for G then we will find it by greedily 2-coloring F (if a vertex is blue, then all its neighbors in F have to be red), which also can be done by traversing F with depth-first search. Given that a spanning forest has at most $O(n)$ edges, the depth-first search requires only $O(n)$ time.

One question remains: How can we construct such a spanning forest F quickly? The answer is that we do not have to construct F explicitly; instead we can use the fact that the Delaunay triangulation T of V contains such a spanning forest $F \subseteq T$ as a subgraph. Indeed, it is known

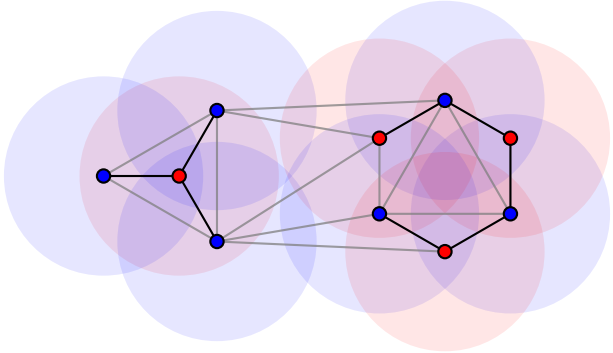

Figure 3: The black edges form a spanning forest $F \subseteq G$ with the same connected components as G. The black and gray edges combined form the Delaunay triangulation T.

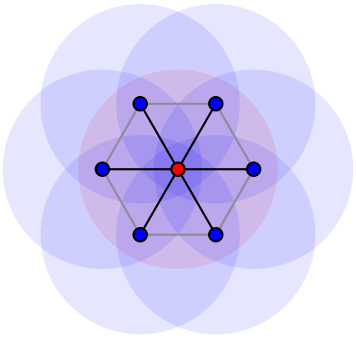

Figure 4: An example where the spanning forest F (black) is 2-colorable even though the conflict graph G (black and gray) is not.

that T contains a Euclidean minimum-length spanning tree, and it can be shown that after removing all edges that are longer than r we obtain a forest F as required (c.f. Lemma 3 in the solution of "GoldenEye").

The algorithm is thus as follows: First, compute the Delaunay triangulation T of V in time $O(n \log n)$. Then, traverse T with depth-first search, while ignoring all edges that are longer than r, in time $O(n)$. During that search, make a note for every vertex that you visit that it belongs to the same connected component as its immediate predecessor. Moreover, assign one of two colors to every vertex, where the color is chosen such that it is distinct from the vertex' immediate predecessor. As a final validation step, compute the length of the smallest edge for both color classes individually, and reject the computed 2-coloring if the computed length is not larger than r (observe that, if G was 2-colorable to begin with, we could not possibly have run into such a conflict since all the decisions that we made while 2-coloring F were forced). Note that this final validation step is necessary because it might be that F is 2-colorable while G is not, see Figure 4.

4 Implementation

The most interesting part of the implementation is arguably the depth-first search, which we implement in a separate function with the following signature.

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef std::pair<int,bool> info_t;
typedef CGAL::Triangulation_vertex_base_with_info_2<info_t,K> vertex_t;
typedef CGAL::Triangulation_face_base_2<K> face_t;
typedef CGAL::Triangulation_data_structure_2<vertex_t,face_t> triangulation_t;
typedef CGAL::Delaunay_triangulation_2<K,triangulation_t> delaunay_t;
typedef delaunay_t::Vertex_handle vhandle_t;
typedef delaunay_t::Vertex_circulator vcirculator_t;

bool try_two_color(delaunay_t & trg, K::FT const & rr);
```

The first argument `trg` is the Delaunay triangulation of the set of radio stations, and the second argument `rr` is the squared operation range r^2. If successful, the function returns `true` and it assigns an `info_t`-field to each vertex, where the `int` identifies the corresponding connected component and the `bool` indicates which one of two colors has been given to that vertex. If unsuccessful, the function returns `false`.

Aside from the following remarks, the function is composed of a straight-forward implementation of depth-first search. Firstly, we use the special value 0 in the `info_t`-field in order to identify vertices that have not yet been visited (see lines 4, 10 and 23). Secondly, as soon as we find a conflict, in the sense that two adjacent vertices have been assigned the same color, we return `false` immediately since at that point we know that no 2-coloring exists (see line 22).

```
1  bool try_two_color(delaunay_t & trg, K::FT const & rr)
2  {
3    for (auto v = trg.finite_vertices_begin(); v != trg.finite_vertices_end(); ++v)
4      v->info() = { 0, false };
5
6    int components = 0;
7    delaunay_t trg1, trg0;
8    for (auto v = trg.finite_vertices_begin(); v != trg.finite_vertices_end(); ++v)
9    {
10     if (v->info().first == 0)
11     {
12       v->info().first = ++components;
13       std::vector<vhandle_t> stack(1, v);
14       do
15       {
16         vhandle_t h = stack.back();
17         stack.pop_back();
18         vcirculator_t c = trg.incident_vertices(h);
19         do if (!trg.is_infinite(c) &&
20           CGAL::squared_distance(h->point(), c->point()) <= rr)
21         {
22           if (c->info() == h->info()) return false;
23           if (c->info().first == 0)
24           {
25             stack.push_back(c);
26             c->info() = { components, !h->info().second };
27           }
28         }
29         while (++c != trg.incident_vertices(h));
30       }
31       while (!stack.empty());
32     }
33
34     if (v->info().second) trg1.insert(v->point());
35     else trg0.insert(v->point());
36   }
37
38   return !has_interference(trg0, rr) && !has_interference(trg1, rr);
39 }
```

The function named `has_interference` does the final sanity check by verifying that no two radio stations with the same color are too close to each other. For this we use once more (c.f. "Graypes") that the Delaunay triangulation contains the shortest possible edge. In the implementation we also make use of the fact that we can return early, that is, as soon as any edge has been found that is too short.

```
1  bool has_interference(delaunay_t const & trg, K::FT const & rr)
2  {
3    for (auto e = trg.finite_edges_begin(); e != trg.finite_edges_end(); ++e)
4      if (trg.segment(*e).squared_length() <= rr) return true;
5    return false;
6  }
```

Solving the second part of the problem is comparatively easy once the 2-coloring and the connected components have been determined. We refer to Section 5 for a complete solution.

5 A complete solution

```
#include <iostream>
#include <vector>
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Triangulation_data_structure_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Delaunay_triangulation_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef std::pair<int,bool> info_t;
typedef CGAL::Triangulation_vertex_base_with_info_2<info_t,K> vertex_t;
typedef CGAL::Triangulation_face_base_2<K> face_t;
typedef CGAL::Triangulation_data_structure_2<vertex_t,face_t> triangulation_t;
typedef CGAL::Delaunay_triangulation_2<K,triangulation_t> delaunay_t;
typedef delaunay_t::Vertex_handle vhandle_t;
typedef delaunay_t::Vertex_circulator vcirculator_t;

bool has_interference(delaunay_t const & trg, K::FT const & rr)
{
  for (auto e = trg.finite_edges_begin(); e != trg.finite_edges_end(); ++e)
    if (trg.segment(*e).squared_length() <= rr) return true;
  return false;
}

bool try_two_color(delaunay_t & trg, K::FT const & rr)
{
  for (auto v = trg.finite_vertices_begin(); v != trg.finite_vertices_end(); ++v)
    v->info() = { 0, false };

  int components = 0;
  delaunay_t trg1, trg0;
  for (auto v = trg.finite_vertices_begin(); v != trg.finite_vertices_end(); ++v)
  {
    if (v->info().first == 0)
    {
      v->info().first = ++components;
      std::vector<vhandle_t> stack(1, v);
      do
      {
        vhandle_t h = stack.back();
        stack.pop_back();
        vcirculator_t c = trg.incident_vertices(h);
        do if (!trg.is_infinite(c) &&
          CGAL::squared_distance(h->point(), c->point()) <= rr)
        {
          if (c->info() == h->info()) return false;
          if (c->info().first == 0)
          {
            stack.push_back(c);
            c->info() = { components, !h->info().second };
          }
        }
        while (++c != trg.incident_vertices(h));
```

```
      }
      while (!stack.empty());
    }

    if (v->info().second) trg1.insert(v->point());
    else trg0.insert(v->point());
  }

  return !has_interference(trg0, rr) && !has_interference(trg1, rr);
}

void test_case()
{
  std::size_t n, m; K::FT rr;
  std::cin >> n >> m >> rr;
  rr *= rr;

  std::vector<K::Point_2> stations(n);
  for (std::size_t i = 0; i < n; ++i) std::cin >> stations[i];
  delaunay_t trg(stations.begin(), stations.end());
  bool success = try_two_color(trg, rr);

  for (std::size_t i = 0; i < m; ++i)
  {
    K::Point_2 holmes, watson;
    std::cin >> holmes >> watson;

    if (success)
    {
      if (CGAL::squared_distance(holmes, watson) <= rr)
      {
        std::cout << "y"; continue;
      }

      auto station_holmes = trg.nearest_vertex(holmes);
      auto station_watson = trg.nearest_vertex(watson);
      if (station_holmes->info().first == station_watson->info().first &&
        CGAL::squared_distance(holmes, station_holmes->point()) <= rr &&
        CGAL::squared_distance(watson, station_watson->point()) <= rr)
      {
        std::cout << "y"; continue;
      }
    }
    std::cout << "n";
  }
  std::cout << "\n";
}

int main()
{
  std::ios_base::sync_with_stdio(false);
  std::size_t k;
  std::cin >> k;
  for (int i = 0; i < k; ++i) test_case();
}
```