Solution — GoldenEye

1 The problem in a nutshell

There is a collection of congruent disks (umbrellas) whose centers are given, but the radius (power consumption) is variable. One can safely move within the union $\mathcal{U}$ of these disks. Depending on the radius, $\mathcal{U}$ may consist of several connected components. Missions are given as source and target points. A mission can be executed if source and target lie in the same component of $\mathcal{U}$. The goal is to figure out

(i) which of the missions can be executed for a given starting radius p;

(ii) what is the smallest radius a so that all of the given missions can be executed; and

(iii) what is the smallest radius b so that all of the missions from (i) can be executed.

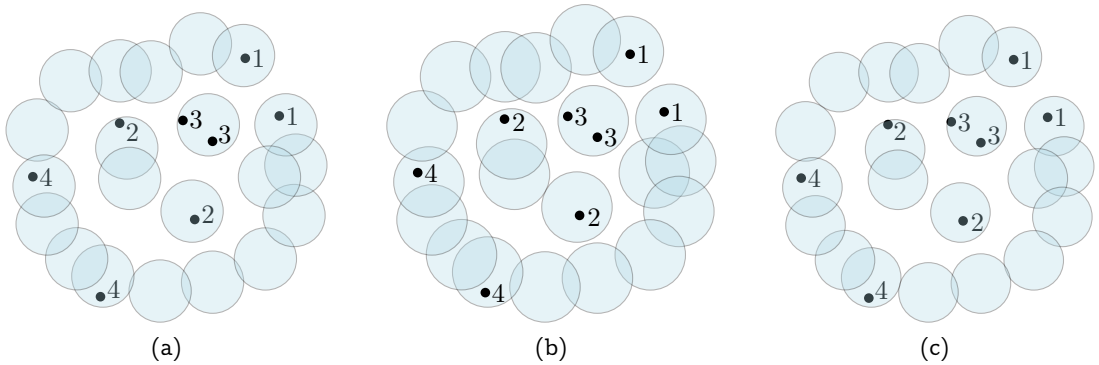

Figure 1: An example: At the initial radius p, missions 1, 3, and 4 can be executed (a); at this radius, all missions can be executed (b); at this radius, which is slightly smaller than p, the missions 1, 3, and 4 can be executed, the same missions as at radius p (c).

2 Modeling

As usual, the first step is to understand the problem statement and derive the essence of the problem, as formulated above. Starting point is the definition of *safe* for a point q, which states that there must be a jammer j_i sufficiently close to q, that is, $\|q - j_i\| \leqslant \sqrt{\omega/4}$, where ω is the power consumption. The usage of Euclidean distance tells us that we deal with disks around the j_i, in which q is safe. Following the basic recommendation to *avoid roots* where possible, we can easily rewrite the expression as $4\|q - j_i\|^2 \leqslant \omega$, using squared Euclidean distances.

The definition of mission sounds a bit complicated at first sight, using the existence of a continuous map $\gamma : [0, 1] \to \mathbb{R}^2$. The question is: Can one draw a continuous (without lifting the pen) curve between the two mission endpoints, such that any point on the curve is safe,

that is, sufficiently close to some jammer, that is, within the disk of radius $\sqrt{\omega/4}$ around some jammer? The definition should be clear enough, but computationally it is somewhat unwieldy. There are infinitely many continuous maps, so how can we tell whether one of them does what we want? Certainly not by inspecting them all...

A trick that often helps to gain some understanding is to simplify the setup. For instance, get rid of some parameters by setting them to specific values, or consider small instances. So, what happens if we have one jammer only? Well, then either both mission endpoints are within the disk around this jammer—let us call it j—and we can walk between them along a straight line (Figure 2a); or at least one of the endpoints is outside the disk around j, and then there is no way to obtain a safe path, if nothing else then because this endpoint is not safe (Figure 2b).

Figure 2: The problem restricted to one jammer.

That was very easy, but of course the setup with one jammer is quite limited. So, how are things with two jammers? Let us denote the jammers by j_0 and j_1. There are two cases: The two disks intersect or they do not intersect.

If the disks do not intersect, the situation is very similar to the case with one jammer because no safe path can ever leave the disk it starts (or ends) in. In other words, we can find a safe path between two points if and only if they are both within the same disk.

Otherwise, the two disks intersect (Figure 3a). This happens, if

$$\|j_0 - j_1\| \leqslant 2\sqrt{\omega/4} \iff \|j_0 - j_1\|^2 \leqslant \omega\,. \tag{1}$$

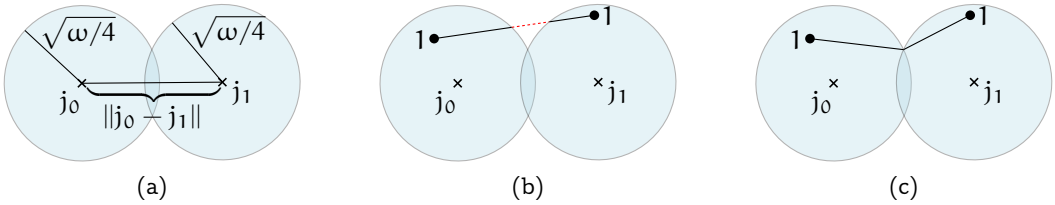

Figure 3: The problem restricted to two jammers.

Aha, maybe this strange value $\sqrt{\omega/4}$ in the problem was not selected to be annoying but actually to yield this nice expression. :-) Nothing changes if both mission endpoints are in the same disk, or if one of the endpoints lies outside of both disks. The interesting case is if one endpoint lies in one of the disks and the other endpoint lies in the other disk. In general, we cannot use a straight line segment to connect both points because this path may leave the union of the two disks (Figure 3b).[1] However, we can route a path that is formed by two line

[1] The property that a straight line segment that connects any two points remains within a set is called *convexity*. While a disk is convex, the union of two disks is not necessarily convex.

segments as follows: Go from one endpoint straight to the closest (in fact, any) intersection point of the disks and from there straight to the target point (Figure 3c). The definition allows any continuous curve, so a polyline formed by two line segments certainly qualifies. Regardless of that specific form and way to construct the curve, we only need to tell whether or not such a curve exists. Clearly, that is the case if and only if the two disks intersect.

Let us summarize the situation for two jammers j_0 and j_1. Let D_i denote the disk of radius $\sqrt{\omega/4}$ around j_i, for $i \in \{0,1\}$. Then a mission from s to t can be executed, if and only if (i) $s,t \in D_0$ or (ii) $s,t \in D_1$, or (iii) $s,t \in D_0 \cup D_1$ and $D_0 \cap D_1 \neq \emptyset$.

The setup with only two jammers appears very restricted still. But we have made a crucial

Observation 1. We can safely go from one disk to another if the two disks intersect.

This observation also holds for more than two jammers/disks. But for more than two disks, the only-if direction does not hold in general: We can move from a disk D_0 to a disk D_1 that intersects D_0, and from there to a third disk D_2 that intersects D_1 but not D_0 (Figure 4).

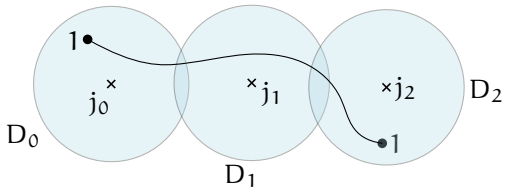

Figure 4: A path through three disks.

One way or another, the principle is: We can move from D_0 to D_1 because they intersect, and we can then move from D_1 to D_2 because the disks intersect. This reasoning strongly suggests to use the so called *intersection graph* of the disks as a model to contemplate about the problem. The vertices of the graph are the disks (of radius $\sqrt{\omega/4}$) around the jammers, and two vertices are connected by an edge, if the corresponding disks intersect. As the graph depends on ω, let us denote it by G_ω. Note that the relation between the disks (do they intersect?) is symmetric. Therefore, we consider G_ω as an undirected graph.

Observation 1 tells us that moving along an edge of G_ω corresponds to the existence of a safe path between any two points within the corresponding disks. Therefore, so does following a sequence of edges, that is, a path in G_ω. This leads to the following

Observation 2. A mission from s to t can be executed at power consumption ω, if and only if there are disks D_s and D_t in the same component of G_ω such that $s \in D_s$ and $t \in D_t$.

Now we have a suitable model that leaves us with a few concrete algorithmic questions.

Question 1. How do we efficiently determine the components of G_ω, for a fixed ω?

Question 2. How do we find suitable disks D_s and D_t?

Question 3. How do we check whether two candidate disks D_s and D_t are in the same component of G_ω?

If we can answer these questions, then we have enough to solve the first group of test sets, where the values a and b are given as part of the input.

In order to solve the remaining groups, we also have to compute a and b, that is, we have to figure out have much power is needed to execute a certain collection of missions. By increasing the power consumption ω we make the disks larger and therefore create more edges in G_ω. Conversely, by decreasing the power consumption ω we make the disks smaller and therefore remove edges from G_ω. The goal is to find the "correct" ω that is just barely good enough for what we need.

Again we face the challenge that conceptually there is an infinite number of choices for ω, if we consider ω to be just any real number. But there is a straightforward discretization because in Observation 2 we only care about the graph G_ω. Consider the evolution of the graph G_ω, when ω runs from 0 to ∞. Initially, at $\omega = 0$, no two disks intersect and so G_ω consists of isolated vertices only. At the end, for ω sufficiently large, all disks intersect and G_ω is a complete graph. In between, there are a number of "interesting" values for ω: whenever a new edge is added to the graph that connects two different components. It follows that other than zero there are at most $n-1$ different "interesting" values for ω.

But how do we find these "interesting" values? Well, they all correspond to an edge being added to the graph. As we have seen in (1), that happens exactly when ω is equal to the squared distance of two jammers. These distances provide us with a finite (quadratic in n) set of candidate values. Let us add to our list of algorithmic questions:

Question 4. How can we efficiently find the interesting values for ω?

3 Algorithm Design

Let us start with the first and central Question 1: How do we efficiently determine the components of G_ω, for a fixed ω?

Computing the components of a graph is a standard problem discussed in every basic algorithms course and you have also seen it in the Algorithms Lab already. The classic approach is to apply BFS or DFS, which takes time linear in the number of vertices and edges. The number of vertices is n, the number of jammers. But what is the number of edges in G_ω? Well, as discussed in the modeling section, it depends on ω. If ω is sufficiently large, then G_ω becomes a complete graph with $\binom{n}{2} = \Theta(n^2)$ edges. We can build G_ω in $O(n^2)$ time by testing all pairs of disks for intersection. Then, computing the components with, say, DFS can be done in $O(n^2)$ as well. For the second group of test sets, where $n \leqslant 1,000$, that should be fast enough. Without this restriction, however, when n is in the tenthousands and n^2 in the hundred millions, we are in trouble. In particular, if we do this computation several times, for several different values of ω. So, let us put this question on our agenda for later:

Question 5. Can we determine the components of G_ω, for a fixed ω, in subquadratic time?

First it seems a good idea to look at the other ingredients needed, so that we get (i) a complete solution for the second group of testsets and (ii) a general overview of where the bottleneck is in our approach, so as to avoid premature optimizations. So, let us move on to Question 2: How do we find suitable disks D_s and D_t?

Let us consider D_s, the disk D_t can be handled analogously. The goal is to find a disk (among the given disks of radius $\sqrt{\omega/4}$ around the jammers) that contains the point s. There may be many such disks, but finding any one of them is enough because all disks that contain s intersect (at s) and so they are in the same component of G_ω. Therefore, why do we not simply

take a disk whose center is closest to s (in Euclidean distance)? If such a disk does not contain s, then none of the disks contains s. Now this is a problem we know very well because we have already encountered it in this course under the name "Bistro".

The centers of the disks are exactly the jammers that we received as input. We can find the jammer closest to s brute force in $O(n)$ time. Over the course of the algorithm we will do this for every mission endpoint, which takes $O(mn)$ time overall. For the first three groups of testsets that should be enough because either $n, m \leqslant 1,000$ or $m \leqslant 80$. For the last two groups of testsets, however, we should come up with a better solution.

By computing the Delaunay triangulation of the jammers in $O(n \log n)$ time, we can find a closest jammer in $O(\log n)$ time. Therefore, we can determine suitable disks D_s and D_t for all missions in $O((n+m)\log n)$ time overall. Given that $n, m \leqslant 30'000$, that should be fast enough.

Now that we have found D_s and D_t, let us move on to Question 3: How do we check whether two candidate disks D_s and D_t are in the same component of G_ω?

Having determined the components of G_ω, we store an identifier for the component (for instance, an integer i for the i-th component) at every jammer. Then once we found the closest jammers for s and t, we compare their corresponding identifiers. This takes constant time because all the work has already been done in computing the components and finding the closest jammers.

Actually, for the first couple of testset groups we could also afford to explore the graph G_ω for every mission, that is, do the BFS/DFS each time to find out whether there is a path from D_s to D_t. This takes $O(mn)$ time overall, which is too slow for the last two groups of testsets.

Then how about Question 4: How can we efficiently find the interesting values for ω?

As observed in the modeling section, we have a list of $\Theta(n^2)$ candidate values for ω: the squared distances $\|j_i - j_k\|^2$. So, a rather brute force way to compute the target values a and b would be to go over all these $O(n^2)$ candidates, construct the corresponding G_ω and its components in $O(n^2)$ time, and check whether all (or, for b, the desired subset of) missions can be executed in $O(mn)$ time, or $O((m+n)\log n)$ time with Delaunay. This yields an overall runtime of $O(n^4)$, which is certainly not good enough for anything beyond the first group of testsets.

However, there is a lot of unnecessary computation in this brute force approach. Clearly, increasing ω can only help the missions because $G_{\omega'} \supseteq G_\omega$, for $\omega' \geqslant \omega$. A natural reflex should be to use this monotonicity and replace the linear search for ω by a binary search. Then one of the n^2 factors is replaced by a factor of $\log n^2 = \Theta(\log n)$, and the overall runtime goes down to something like $O((m+n)n \log n)$. This should be enough for the second group of testsets. But before blindly going for this approach, let us consider possible alternatives.

For once, is it really necessary to recompute G_ω from scratch for every ω? For similar values ω and ω', the graphs G_ω and $G_{\omega'}$ should not differ by much. So why do we not sort the list of candidate values and then consider them in increasing order? Then at every step, we only have to add a single edge to the graph, which can be done in constant time. In fact, there is no need to explicitly store the whole graph G_ω to begin with, as we only need to know the component structure. So, rather than storing every single edge, let us just check whether the edge connects two components. If not, well, nothing changes and we can ignore the new edge. Otherwise, we combine the two components joined by the new edge into a single new component and proceed.

That leaves us with a new question:

Question 6. How can we efficiently maintain the components of G_ω, under increasing ω, that is, addition of edges?

Once again, this is a classical question that is discussed in every course about algorithm design under the name of *union-find* or *disjoint-set* data structure. For a collection of disjoint sets on n elements, one can implement the operations

- $\mathit{Find}(x)$ that returns the set that contains x, and
- $\mathit{Union}(x, y)$ that joins the set that contains x with the one that contains y,

so that a single operation takes $O(\alpha(n))$ amortized time (over a sequence of n operations), where α is the extremely slowly growing inverse Ackermann function. For all practical purposes we may regard $\alpha(n)$ as a small constant. The runtime analysis is a bit involved, but the data structure is rather easy to implement. This data structure you should know, both from the theory point of view and also how to implement it. So, if you are not so firm about the details, please refer to your favorite algorithm textbook to brush up.

In our context, the disjoint sets are the components of the graph. Initially, every vertex (jammer) is alone in its own component. Whenever an edge $j_i j_k$ is added to G_ω, we use the find operation on both to determine whether they are in the same component. And if they are not, then we join the two components with the union operation. The overall time needed to maintain the components of G_ω is this way is $O(n^2\alpha(n))$.

Very conveniently we can use the same disjoint-set data structure to figure out which missions can be executed with respect to the current ω: test whether or not the endpoints are in the same component. The $\alpha(n)$ factor for finding the components is swallowed by the $O(n)$ or $O(\log n)$ factor used to determine the closest jammers.

This leaves us with the last question, Question 5: Can we determine the components of G_ω, for a fixed ω, in subquadratic time?

As discussed in the modeling section, there are no more than n interesting values for ω: 0 plus the (at most) $n-1$ values that correspond to (the length of) edges in G_ω that join two different components. What are these edges that join different components? Given that we consider the edges by increasing length, we actually run Kruskal's algorithm, which at every step adds a shortest edge that connects two components. In other words, the interesting edges in our setting form a Euclidean minimum spanning tree. As discussed in the tutorial about "Proximity Structures", the Delaunay triangulation contains all EMSTs. In the same way we obtain the following

Lemma 3. For every ω, the Delaunay triangulation of $J = \{j_0, \ldots, j_{n-1}\}$ contains all Euclidean minimum spanning forests of G_ω.

Proof. Every Euclidean minimum spanning forest F of G_ω can be augmented to an EMST $T \supseteq F$ for J by adding edges. As shown in the tutorial, every EMST is part of the Delaunay triangulation. In particular, $F \subseteq T$ is contained in the Delaunay triangulation of J. □

Using Lemma 3 we see that all interesting edges appear in the Delaunay triangulation for J. Or rather, all interesting values for ω appear as a squared distance of some edge in the Delaunay triangulation for J. As a triangulation has only $O(n)$ edges, this reduces the number of candidate

values to consider from $\Theta(n^2)$, for all pairs of jammers, to $O(n)$. For n in the range of 30'000 that is a considerable factor to get rid of and reason enough to celebrate!

Algorithm overview. After having collected so many ideas about how to address various subproblems, it is time to step back and consider the big picture. How does the overall algorithm look like?

(1) Compute the Delaunay triangulation D of the jammers.

(2) Extract a vector E of all edges of D and sort E by (squared) length.

(3) Using E, compute a disjoint-set data structure U_p for the components of G_p, where p is the given initial power consumption.

(4) Set $a := 0$ and $b := 0$. Initialize disjoint-set data structures U_a and U_b for J, corresponding to G_a and G_b, respectively.

(5) For every mission (s_i, t_i), with $i = 0, \ldots, m-1$:

  (i) Using D find a closest jammer j_σ for s_i and a closest jammer j_τ for t_i. Let $d_s = 4\|s_i - j_\sigma\|^2$ and $d_t = 4\|t_i - j_\tau\|^2$.

  (ii) Using U_p determine whether j_σ and j_τ are in the same component of G_p. If so and $d_s \leqslant p$ and $d_t \leqslant p$, then output "y"; otherwise, output "n".

  (iii) If (s_i, t_i) can be executed in G_p, then ensure that it can also be executed in G_b:

  - let $b := \max\{b, d_s, d_t\}$ and
  - go over E and add all edges of length $\leqslant b$ to U_b and then continue to add edges (increasing b accordingly) until j_σ and j_τ are in the same component of G_b.

  (iv) Ensure that (s_i, t_i) can also be executed in G_a:

  - let $a := \max\{a, d_s, d_t\}$ and
  - go over E and add all edges of length $\leqslant a$ to U_a and then continue to add edges (increasing a accordingly) until j_σ and j_τ are in the same component of G_a.

(6) Output a and b.

Let us analyze the complexity of the algorithm. Step (1) takes $O(n \log n)$ time. In Step (2) we sort a vector of $O(n)$ edges, which also takes $O(n \log n)$ time. In Step (3) we iterate over a vector of $O(n)$ edges and maintain a disjoint-set data structure, which takes $O(n\alpha(n))$ time. Step (4) takes $O(n)$ time. Step (5)(i) takes $O(\log n)$ time per mission, so $O(m \log n)$ time overall. Step (5)(ii) takes $O(m\alpha(n))$ time overall. In Step (5)(iii) and (iv), the overall costs for going over E and updating G_a and G_b are bounded by $O(n\alpha(n))$. Therefore, the overall time complexity of this algorithm is $O((m+n)\log n)$, and the space complexity is $O(n)$. As $m, n \leqslant 30'000$, we have $\log n \leqslant 15$. Given that our asymptotic notation does not hide large constant factors, we expect to solve all groups of test sets in this way.

Alternative solutions. Another $O((n+m)\log n)$ time solution can be obtained by combining `boost::connected_components` with binary search. Coding this is a bit more work, and the actual runtimes are noticeably slower due to larger constants hidden in the asymptotic term:

If the boost algorithm is treated as a black-box, then the graph has to be rebuilt at every step of the binary search, and the components are always computed from scratch. Nevertheless, if implemented properly such a solution should also give 100 points.

A solution using binary search over the whole value range of `double`, that is, incurring a $\log p$ rather than a $\log n$ factor, should give 75 points. If implemented efficiently it may also give 100 points. But such as solution is suboptimal: From a theoretical point of view, an algorithm whose complexity can be bounded in n and m independently from p is much better than an algorithm that slows down with increasing p. From a practical perspective, a factor of $\log p \approx 50$ is rather large, even compared to $\log n \approx 15$.

Another way to arrive at 75 points is to spend an additional $\log n$ factor in runtime, for instance, by using `std::map` to store indices or by recomputing nearest neighbors at every step of the binary search. Or one can compute components using BFS/DFS in $O(n(m + \log n))$ time, which should still be fine for the third group of test sets with $m \leqslant 80$.

Without using Delaunay triangulations one can spend $O((n + m)n \log n)$ time to compute G_ω from the complete graph. If implemented properly, then 50 points are the reward.

For the first group of test sets there is no need to compute a and b. It can, for instance, be solved by computing for every mission the minimum power needed using BFS/DFS on the complete graph in $O(mn^2 \log n)$ time.

4 Implementation

Having designed the algorithm properly, an implementation should be straightforward. Let us discuss a few relevent details nevertheless.

For once, note that the input specification for the initial power consumption p says $p < 2^{53}$. This exhausts the mantissa width of `double`, so we should be very careful when doing further calculations with this value. If you follow the general advice to avoid roots and reformulate the relevant expressions as discussed in Section 2 (see Equation (1)), there is no need for calculations and the value can be used as-is.

For the coordinates of jammers and mission endpoints, the input specification tells us $< 2^{24}$. So computing and comparing squared distances using `double` is fine.

To build the Delaunay triangulation, we need predicates only.

After these three observations it should be clear that no exact constructions are needed, and so the EPIC-kernel[2] is the kernel of choice.

As for the disjoint-set data structure, you can either implement it yourself, or use the available implementation `boost::disjoint_sets`. Regardless, make sure that you know how to implement this data structure. A Master of Computer Science should know this... See Section 6 for an example implementation.

In order to connect the vertices of the Delaunay triangulation to the disjoint-set data structure, it is convenient to use indices. That is, we associate an index (`int` from 0 to $n - 1$) to every vertex and use these indices also as entities in the disjoint-set data structure.

[2] `CGAL::Exact_predicates_inexact_constructions_kernel`

5 A Complete Solution

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_2.h>
#include <CGAL/Triangulation_vertex_base_with_info_2.h>
#include <CGAL/Triangulation_face_base_2.h>
#include <vector>
#include <boost/pending/disjoint_sets.hpp>

typedef CGAL::Exact_predicates_inexact_constructions_kernel  K;
typedef CGAL::Triangulation_vertex_base_with_info_2<int,K>   Vb;
typedef CGAL::Triangulation_face_base_2<K>                   Fb;
typedef CGAL::Triangulation_data_structure_2<Vb,Fb>          Tds;
typedef CGAL::Delaunay_triangulation_2<K,Tds>                Delaunay;
typedef Delaunay::Finite_edges_iterator                      EI;
typedef std::pair<K::Point_2,int>                            IPoint;
typedef boost::disjoint_sets_with_storage<>                  Uf;

struct Edge {
  Edge(int u_, int v_, K::FT sql_) : u(u_), v(v_), sql(sql_) {}
  int u, v;  // endpoints
  K::FT sql; // squared length
};

inline bool operator<(const Edge& e, const Edge& f) { return e.sql < f.sql; }

void handle_missions() {
  std::size_t n, m;
  double p;
  std::cin >> n >> m >> p;

  // read jammers and build Delaunay
  std::vector<IPoint> jammers;
  jammers.reserve(n);
  for (std::size_t i = 0; i < n; ++i) {
    int x, y;
    std::cin >> x >> y;
    jammers.push_back(IPoint(K::Point_2(x,y), i));
  }
  Delaunay t;
  t.insert(jammers.begin(), jammers.end());

  // extract edges and sort by length
  std::vector<Edge> edges;
  edges.reserve(3*n);
  for (EI e = t.finite_edges_begin(); e != t.finite_edges_end(); ++e)
    edges.push_back(Edge(e->first->vertex((e->second+1)%3)->info(),
                         e->first->vertex((e->second+2)%3)->info(),
                         t.segment(e).squared_length()));
  std::sort(edges.begin(), edges.end());

  // compute components with power consumption p
  Uf ufp(n);
  typedef std::vector<Edge>::const_iterator ECI;
  for (ECI e = edges.begin(); e != edges.end() && e->sql <= p; ++e)
    ufp.union_set(e->u, e->v);

  // handle missions
  K::FT a = 0;
```

```
  K::FT b = 0;
  Uf ufa(n);
  Uf ufb(n);
  ECI ai = edges.begin();
  ECI bi = edges.begin();
  for (std::size_t i = 0; i < m; ++i) {
    int x0, y0, x1, y1;
    std::cin >> x0 >> y0 >> x1 >> y1;
    K::Point_2 p0(x0, y0), p1(x1, y1);
    Delaunay::Vertex_handle v0 = t.nearest_vertex(p0);
    Delaunay::Vertex_handle v1 = t.nearest_vertex(p1);
    K::FT d = 4 * std::max(CGAL::squared_distance(p0, v0->point()),
                           CGAL::squared_distance(p1, v1->point()));
    if (d <= p && ufp.find_set(v0->info()) == ufp.find_set(v1->info())) {
      // mission possible with power p -> also with b
      std::cout << "y";
      if (d > b) b = d;
      for (; bi != edges.end() &&
           ufb.find_set(v0->info()) != ufb.find_set(v1->info());
           ++bi)
        ufb.union_set(bi->u, bi->v);
    } else
      std::cout << "n";
    // ensure it is possible at power a
    if (d > a) a = d;
    for (; ai != edges.end() &&
         ufa.find_set(v0->info()) != ufa.find_set(v1->info());
         ++ai)
      ufa.union_set(ai->u, ai->v);
  }
  if (ai != edges.begin() && (ai-1)->sql > a) a = (ai-1)->sql;
  if (bi != edges.begin() && (bi-1)->sql > b) b = (bi-1)->sql;
  std::cout << "\n" << a << "\n" << b << "\n";
}

int main()
{
  std::ios_base::sync_with_stdio(false);
  std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
  std::size_t t;
  for (std::cin >> t; t > 0; --t) handle_missions();
  return 0;
}
```

6 A Union-Find Data Structure

Below is an example for a simple implementation of a disjoint sets data structure that uses union by size and path compression. The amortized complexity of an operation is $O(\alpha(n))$, where α is the inverse Ackermann function, which can be regarded as constant for all practical purposes.

```
struct Uf {
  Uf(std::size_t n) : sz(n,1) {
    comp.reserve(n);
    for (std::size_t i = 0; i < n; ++i) comp.push_back(i);
  }
  void union_set(std::size_t i, std::size_t j) {
    i = find_set(i);
```

```
    j = find_set(j);
    if (i != j) {
      if (sz[i] < sz[j]) std::swap(i,j);
      comp[j] = i;
      sz[i] += sz[j];
    }
  }
  std::size_t find_set(std::size_t i) {
    assert(i < comp.size());
    if (comp[i] == i) return i; else return comp[i] = find_set(comp[i]);
  }
private:
  std::vector<std::size_t> sz, comp;
};
```