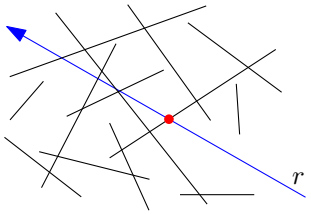
Solution — First hit

1 The problem in a nutshell

Given a ray r and a collection S of line segments in $\mathbb{R}^2$, where does r hit a first segment from S?



2 Modeling

Clearly the problem is of geometric nature. Therefore CGAL is a natural tool to use. Given that an actual hit (that is, intersection) point is required in the output, exact geometric constructions are needed in general. If a ray does not hit any segment, then predicates should suffice (as in the problem "Hit"). Maybe there is some balance to strike between the use of predicates and constructions?

The numbers in the input can be rather large (up to $\approx 2^{51}$). Therefore, it is unlikely that naive use of limited precision arithmetic will work. In addition, there are a few properties of the input and some special cases to keep in mind:

- The segments in S may intersect or overlap. There may even be several copies of the same segment in S.
- The ray r may hit no segment from S ($\Rightarrow$ Output no.)
- The ray r may hit several segments at the same point.
- The source of r may lie on a segment from S, effectively ending the ray "before it starts".
- The ray r may be collinear with one or several segments from S.

None of these issues should really pose a problem. But it is good to be aware of them, so as to consciously dismiss them if irrelevant, rather than ignore them for lack of consideration.

Overall, there is not all that much to model here. For the time being, it seems natural to simply model S as a linear sequence of line segments that we can iterate over from front to back (using the C++ concept/named requirement `ForwardIterator`).

3 Algorithm Design

There is a simple and obvious linear time algorithm: Check every segment s: Does s intersect r? If so compute $s \cap r$. Among all such intersection points compute the one closest to the source

of r. A proper implementation of this algorithm actually yields 99 of 100 points and this is what everybody should be able to get easily. So what can we possibly improve in a linear time algorithm?

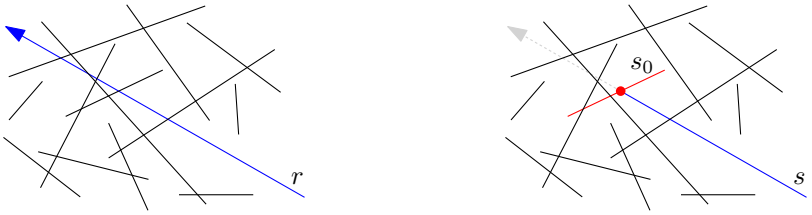
One of the basic recommendations for geometric algorithms is to *avoid constructions* if possible. Constructions tend to increase the size of the numbers/coordinates to deal with, and therefore, slow down the computation. In this problem we basically use two nontrivial geometric operations:

(i) testing a ray and a segment for intersection, and

(ii) constructing the intersection of a ray and a segment.

Clearly, (i) is a predicate whereas (ii) is a construction. So we aim to minimize the number of type (ii) operations executed by our algorithm. In the current version, both (i) and (ii) are executed for every segment, that is, $n = |S|$ times. The following observations turn out helpful to reduce the number of constructions.

Clipping. First note that we do not construct anything for those segments that do not intersect the ray r. Unfortunately we have no control over how many segments in the input intersect r. In the worst case all of them do. However, as soon as we found one segment s_0 that intersects r, we are not interested anymore in segments that intersect r *after* s_0 (when running through r from its source to infinity). Therefore, we can effectively discard the portion of r after the intersection with s_0. In geometric terms, we can clip r to become a line segment s.



This modification should decrease the number of intersection constructions, but it is not clear by how much. In the worst case there is no difference at all, namely if we encounter the intersection points in decreasing distance from the source of r. However, that would be a rather unlucky coincidence. In fact, we can most likely avoid such a worst case behaviour by treating the segments of S in a (uniformly) *random* order rather than in the specific order they are given to us (which may be bad). This modification works because the order in which we treat the segments of S is irrelevant for the correctness of the algorithm. The runtime, however, *is* affected by the order. Now our algorithm is a randomized algorithm and we would like to analyze its *expected* runtime to learn how much—if anything—we gain compared to the original deterministic algorithm.

In order to be able to randomly permute S we change our model so that S is a linear sequence with random access to individual elements (using the C++ concept/named requirement `RandomAccessIterator`).

Analysis. Intuitively we expect to start with a segment somewhere in the middle (with respect to the order of intersections along r) and therefore discard a constant fraction of intersection

points from consideration. In fact we expect this to happen *every time* we discover a new intersection with the segment s (the clipped ray r), which suggests a logarithmic number of intersection constructions in expectation. Let us formally argue to confirm our expectations.

We can model our problem one-dimensionally as follows. A segment s_i that intersects r is represented by the point in $s_i \cap r$ closest to the source of r (recall that this intersection may not be a single point). If $s_i \cap r = \emptyset$, then we simply ignore s_i because it is irrelevant for our problem and we will never (regardless of the order for S) use an intersection construction for s_i. In this way we regard the segments as numbers $x_0, \ldots, x_n$ in $\mathbb{R}^+ \simeq r$. Note that several segments may intersect r in the same point and so in general some of the x_i may be equal. We will get back to this, but for the time being let us suppose that this does not happen, that is, the x_i are pairwise distinct.

When our algorithm handles s_i, a new intersection is constructed if and only if s_i intersects r before *all* other segments $s_0, \ldots, s_{i-1}$. In our one-dimensional model this corresponds to $x_i = \min\{x_0, \ldots, x_i\}$. Denote $m_i = \min\{x_0, \ldots, x_i\}$, for $i \in \{0, \ldots, n-1\}$. Then the expected number of intersection constructions done by the algorithm corresponds to the expected number of changes in the sequence $m_0, \ldots, m_{n-1}$.

Lemma 1. Let $x_0, \ldots, x_{n-1}$ be a (uniformly) random permutation of n pairwise distinct real numbers, and let $m_i = \min\{x_0, \ldots, x_i\}$. Then the expected number of changes in the sequence $m_0, \ldots, m_{n-1}$ is $H_n = \Theta(\log n)$ (where we count the appearance of the first element m_0 as a change).

Proof. Let $X_i = 1$, if $m_i \neq m_{i-1}$, and $X_i = 0$, otherwise. Note that $m_i = m_{i-1}$, unless x_i is the minimum of the first $i+1$ elements, and this minimum is unique because the x_i are pairwise distinct. There are $i!$ permutations of $i+1$ pairwise distinct numbers where the unique minimum appears last. Therefore, we have

$$E[X_i] = p[X_i = 1] = \frac{i!}{(i+1)!} = \frac{1}{i+1}.$$

The quantity we are interested in is

$$E\left[\sum_{i=0}^{n-1} X_i\right] = \sum_{i=0}^{n-1} E[X_i] = \sum_{i=0}^{n-1} \frac{1}{i+1} = \sum_{i=1}^{n} \frac{1}{i} = H_n = \Theta(\log n),$$

where the first equality is by linearity of expectation and where H_n denotes the n-th Harmonic number. □

The analysis confirms our intuition that clipping reduces the number of intersection constructions (type (ii) operations) from linear to expected $O(\log n)$. The number of intersection tests (type (i) operations) stays at n, but the tests are noticeably cheaper than the constructions. Of course, the overall asymptotic complexity is linear regardless. But when measuring absolute runtimes we should not ignore multiplicative constants completely.

Discussion. But is there not one detail that we forgot? Lemma 1 assumes that the values x_i are pairwise distinct, which may not hold for our segments in general (because many segments *may* intersect r in the same point). There are two possible answers:

(1) *So what?* The algorithm is correct, as it was without the clipping. The goal was to improve the performance, which we achieved for many instances, in particular generic ones. Many intersections falling together is a degeneracy, that is addressed correctly, albeit without speedup.

(2) *Alright, ...* then why do we not realize the intersection test so that it disregards an intersection at one of the endpoints of the clipped ray (the endpoint that is not the source of r)?

Both answers have their merits and our test sets intentionally do not punish you for choosing the first. If you tend towards the second answer, there is an easy way to address this issue: Modify the intersection test so that it disregards a segment s_i that intersects only the target point t of the clipped ray s. Feel encouraged to work out the details, but we will not further discuss this here.

The technique applied here is not specific to the problem at hand, but applies to a larger class of optimization problems where updating an objective value is more expensive than testing for violation of optimality. See the paper by Chan [1] for more examples.

4 Implementation

The implementation is straightforward. A few remarks are in order, though.

- We do not need roots but we construct intersection points. Therefore, using `CGAL::Exact_predicates_exact_constructions_kernel` is the logical choice of kernel.
- The input numbers are up to 51 bits, so following the tutorial slides we use the type `long` to read them in. For the final output, we use the suggested rounding function from the tutorial.
- A uniformly random permutation of an n element set is easy to generate in $O(n)$ time. The standard library provides a function `std::random_shuffle` that does it for a given range.
- When implementing the clipping, we have to take care of the case that the intersection between the two segments (or the ray and the segment) is a segment. This happens when a segment from S is collinear with r. In such a case, we need to determine the endpoint of the intersection (segment) that is closer to the source of r and clip there. Conveniently, browsing the available predicates on the "2D and 3D Linear Geometry Kernel Reference" page of the CGAL manual, we find a predicate named `CGAL::collinear_are_ordered_along_line` that allows to make that decision. Another straightforward way to make that decision is to compare the (squared) distances of the points.
- Whenever constructing an intersection $s_i \cap r$, it is better to intersect s_i with r rather than with s (the current clipped version of r). Of course, the result is the same. But the coordinates of s (at least for the endpoint that is not the source of r) can be much larger than the coordinates used to represent r, which are part of the original input.

5 A Complete Solution

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <vector>
#include <algorithm>
#include <type_traits>
#include <stdexcept>

typedef CGAL::Exact_predicates_exact_constructions_kernel K;
typedef std::result_of<K::Intersect_2(K::Ray_2,K::Segment_2)>::type IT;

// round down to next double (as defined in the tutorial)
double floor_to_double(const K::FT& x) {
  double a = std::floor(CGAL::to_double(x));
  while (a > x) a -= 1;
  while (a+1 <= x) a += 1;
  return a;
}

// clip/set target of s to o
void shorten_segment(K::Segment_2& s, const IT& o) {
  if (const K::Point_2* p = boost::get<K::Point_2>(&*o))
    s = K::Segment_2(s.source(), *p);
  else if (const K::Segment_2* t = boost::get<K::Segment_2>(&*o))
    // select endpoint of *t closer to s.source()
    if (CGAL::collinear_are_ordered_along_line
        (s.source(), t->source(), t->target()))
      s = K::Segment_2(s.source(), t->source());
    else
      s = K::Segment_2(s.source(), t->target());
  else
    throw std::runtime_error("Strange segment intersection.");
}

void find_hit(std::size_t n) {
  // read input
  long x1, y1, x2, y2;
  std::cin >> x1 >> y1 >> x2 >> y2;
  K::Ray_2 r(K::Point_2(x1, y1), K::Point_2(x2, y2));
  std::vector<K::Segment_2> segs;
  segs.reserve(n);
  for (std::size_t i = 0; i < n; ++i) {
    std::cin >> x1 >> y1 >> x2 >> y2;
    segs.push_back(K::Segment_2(K::Point_2(x1,y1), K::Point_2(x2,y2)));
  }
  std::random_shuffle(segs.begin(), segs.end());

  // clip the ray at each segment hit (cuts down on the number of intersection
  // points to be constructed: for a uniformly random order of segments, the
  // expected number of constructions is logarithmic in the number of segments
  // that intersect the initial ray.)
  K::Segment_2 rc(r.source(), r.point(1));

  // find some segment hit by r
  std::size_t i = 0;
  for (; i < n; ++i)
    if (CGAL::do_intersect(segs[i], r)) {
      shorten_segment(rc, CGAL::intersection(segs[i], r));
      break;
```

```
    }
  if (i == n) { std::cout << "no\n"; return; }
  // check remaining segments against rc
  while (++i < n)
    if (CGAL::do_intersect(segs[i], rc))
      shorten_segment(rc, CGAL::intersection(segs[i], r)); // not rc!

  std::cout << floor_to_double(rc.target().x()) << " "
            << floor_to_double(rc.target().y()) << "\n";
}

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cout << std::setiosflags(std::ios::fixed) << std::setprecision(0);
  for (std::size_t n; std::cin >> n && n > 0;)
    find_hit(n);
}
```

References

[1] Timothy M. Chan, Geometric Applications of a Randomized Optimization Technique. *Discrete Comput. Geom.*, **22**, 4, (1999), 547–567.