# Solution — Hiking Maps

## 1 The problem in a nutshell

Cover a polygonal (hiking) path with triangles (map parts) so that every segment of the path is fully contained in at least one triangle. The triangles are given as a sequence $t_0, \ldots, t_{n-1}$, and we must determine a minimum length subsequence $t_i, \ldots, t_j$ that provides such a cover for the given path. The output is the length $j - i + 1$ of such a minimum subsequence.

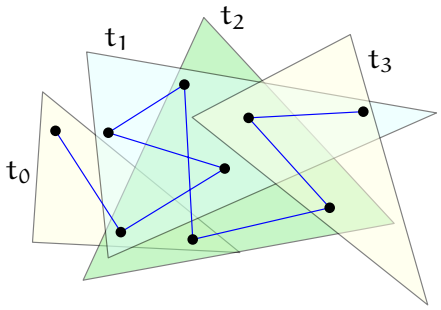


Figure 1: An example where the triangles $t_0, t_1, t_2$ form a minimum covering and so the correct output is 3.

## 2 Modeling

Both the path and the map parts are given geometrically, by coordinates in $\mathbb{R}^2$. Therefore, CGAL is a natural tool to use. On a high level, this problem appears quite similar to the problem "Search Snippets" from the week before, where we must determine a minimum length subsequence of documents that contains a given collection of words. Maybe there is an analogy here where triangles correspond to documents and path segments correspond to words? Answer: Indeed, it is exactly the same problem...

On a lower level, what kind of primitive operations might be useful to solve the problem? Leaving optimality aside for the moment, we need to check the covering condition, that is, is every path segment fully contained in at least one of the selected triangles? Hence it would be helpful to have as a primitive operation a test that determines whether a given triangle contains a given segment of the path. Let us discuss how such a test might look like.

The triangles in the input are provided in a somewhat peculiar fashion: by six points on the boundary, two for each side. Consulting the representation slide from the tutorial reveals that this is not how a `CGAL::Triangle_2` is represented usually; the default representation stores the three vertices of the triangle. Of course, we could simply construct these vertices by building lines from the given point pairs and then intersecting those lines. However, doing so obviously uses geometric constructions and in order to be sure that these constructions are carried out correctly, we would have to use a kernel that supports exact constructions. Maybe

exact constructions are necessary, but before taking that route we should at least consider possible alternatives that may avoid using exact constructions.

Note that a segment $s$ is contained in a triangle $t$ if and only if both endpoints of $s$ are in $t$. So how can we test whether a point $p$ is contained in a triangle $t$? The triangle $t$ has three sides, and in order to be inside, $p$ must be on the "correct" side, for each of these three sides. This type of question should remind you of the sidedness or *orientation* test that was discussed in the tutorial: Given three points $p, q, r$, do they form a leftturn, a rightturn, or are they straight/collinear? So if we have two points $q_0, q_1$ on a side of $t$, then by testing the orientation of the triple $q_0, q_1, p$ we can determine whether or not $p$ is on the correct side of the line through $q_0$ and $q_1$. But which of the two sides is the "correct" side? Well, here it comes handy that all the points $q_0, \ldots, q_5$ lie in the relative interior of the corresponding edge by specification. Therefore, all the other points $q_2, \ldots, q_5$ lie on the correct side of $q_0q_1$ and we can use any of them to determine the "correct" side—or rather order—for the pair $q_0, q_1$.

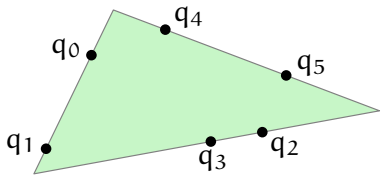


Figure 2: A point $p$ is contained in this triangle, if and only if none of $(q_0, q_1, p)$, $(q_3, q_2, p)$, or $(q_5, q_4, p)$ is a rightturn.

By implementing the point-in-triangle test in such a way, we use the orientation predicate only and avoid geometric constructions altogether. Being able to use the EPIC[1] kernel is always preferable compared to a kernel that supports exact constructions.

## 3 Algorithm Design

There are two orthogonal aspects in this problem, for which we award points. The first aspect is the use of predicates vs. constructions, as discussed in the preceding section. The second aspect is the overall complexity of the combinatorial algorithm.

**Brute force.** The total number of possible subsequences is $\Theta(n^2)$. For each subsequence we can check in $O(mn)$ time whether or not it is a valid cover. Therefore, a brute-force solution can be implemented in $O(mn^3)$ time. Even with moderate input sizes of $m, n \leqslant 100$ for the first two testsets, we should not count on getting any points for such a solution. However, in practice the algorithm may not reach this worst case bound. This is because some path segments may only be contained in few map triangles and we can stop testing a candidate subsequence as soon as a single segment is not covered.

**Binary search.** A first improvement can be obtained by doing binary search over the length of the sequence. Initially, the length can be anywhere in $[1, n]$. For a fixed length we have $O(n)$ possible sequences, for each of which we can test in $O(mn)$ time whether it provides a suitable covering. If any of them does, we continue to look for sequences of smaller length; otherwise,

[1] shortcut for Exact Predicates Inexact Constructions

we need to increase the length of the sequence. The overall runtime is $O(mn^2 \log n)$, which should be enough for the first two testsets.

In this problem, the first two testsets are designed rather generously. Probably any reasonable correct implementation scores 20 points for the first and 20 more for the second, if either using binary search or geometric constructions are avoided. In general, you should not expect to score points for a brute-force algorithm, unless such an algorithm requires some insights already (as here: binary search or avoiding constructions).

**Scan and update.** For $m$ and $n$ in the thousands, as in the last three testsets, a cubic algorithm is hopeless. In order to improve the runtime, observe that similar sequences cover a similar subset of path segments. For instance, if a sequence $t_i, \ldots, t_j$ of triangles covers a certain subset of path segments, then the sequence $t_i, \ldots, t_{j+1}$ covers all those segments as well, plus possibly additional segments covered by $t_{j+1}$. Rather than testing both sequences independently, we should first test $t_i, \ldots, t_j$. If it covers all segments, there is no need to test $t_i, \ldots, t_{j+1}$ at all because we are looking for a shortest covering sequence only. Instead we should try to see whether $t_{i+1}, \ldots, t_j$ covers all path segments as well. If $t_i, \ldots, t_j$ does not cover all path segments, then we should check for all uncovered segments whether they are covered by $t_{j+1}$.

Let us cast this idea into an algorithm. We maintain for every path segment $s_i := p_i p_{i+1}$ a counter `covered`, that indicates how many triangles from the current sequence $t_b, \ldots, t_{e-1}$ cover $s_i$. The segment $s_i$ is covered by $t_b, \ldots, t_{e-1}$ if and only if `covered[i]` $\geqslant 1$. First we let $b = e = 0$, that is, set $t_b, \ldots, t_{e-1}$ to the empty sequence. Then we gradually increment $e$, until $t_b, \ldots, t_{e-1}$ covers all path segments. Whenever we increment $e$, we test for every segment $s_i$ whether it is covered by $t_{e-1}$ and if so, increment `covered[i]`. In order to easily see whether the current sequence provides a full covering, it is convenient to maintain a counter `uncovered` that indicates how many segments are not covered by the current sequence $t_b, \ldots, t_{e-1}$. Whenever we increment `covered[i]` for some segment $s_i$ from zero to one, the counter `uncovered` is decremented accordingly.

As soon as $t_b, \ldots, t_{e-1}$ forms a covering, we check whether we can increment $b$ and still maintain a (thereby shorter) covering. Whenever incrementing $b$, we test for every segment $s_i$ whether it is covered by $t_b$ and if so, decrement `covered[i]`. If `covered[i]` goes down to zero for some $s_i$, we increment `uncovered` accordingly. Also this is the moment when we realize that incrementing $b$ destroyed the covering property and, therefore, the sequence $t_{b-1}, \ldots, t_{e-1}$ is our next candidate for a minimum length covering. Afterwards, we start to increase $e$ so as to regain a covering.

In this way we iterate once over $t_0, \ldots, t_{n-1}$. At every step, we first add triangles at the end of the current sequence until it forms a covering and then remove triangles from the beginning of the sequence until the covering property is lost. Among the resulting candidate sequences we select a shortest one to give our overall result. The outer loop iterates once over all triangles and the inner loop iterates over all path segments. Therefore the overall runtime is $O(mn)$, which should be enough for $m, n \leqslant 2,000$ to score the remaining 60 points.

## 4 Implementation

After the discussion in the preceding two sections there is not much left to say. Constructions can be avoided, hence the EPIC kernel is the kernel of choice (and you will notice a drop in performance when using an exact constructions kernel instead).

An additional speedup may be obtained by caching the containment test results, that is, precomputing and storing for every triangle which path segments it contains. What could we hope to gain by that? If the scan algorithm is implemented correctly, it tests every combination twice: once when adding the triangle to the current sequence and again when removing it. Hence the gain should be about a factor of two (for the time spent in orientation tests).

But in addition there is a possible gain in not having to iterate over *all* path segments but only over *those* that are actually *contained* in the triangle (to be added to or removed from the current sequence). Obviously the difference between those two groups depends on the input and it may be zero, in case the triangle contains all path segments. But certainly the restriction does not hurt and there may very well be a noticeable gain for realistic instances (where we would expect every map triangle to contain only a small number of path segments). Indeed this change results in a noticeable speedup for the large testsets. But in this problem we opted to not reward this optimization with extra points.

## 5 Expectations

The combinatorial aspect of this problem is very similar (in fact, identical) to the problem "Search Snippets" that you have encountered before. The geometric aspect (predicate vs. constructions) has already been encountered in the problems "Hit" vs. "First Hit". Therefore, it is our expectation that a vast majority of students is able to solve this problem to completion and score all 100 points.

## 6 A Complete Solution

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <vector>
#include <algorithm>
#include <iostream>

typedef CGAL::Exact_predicates_inexact_constructions_kernel K;
typedef std::vector<K::Point_2> Pts;
typedef std::vector<std::size_t> Covered;
typedef std::vector<Covered> Trs;
typedef Covered::const_iterator CCI;

// does triangle t contain point p?
inline bool contains(const Pts& t, const K::Point_2& p) {
  return !CGAL::right_turn(t[0],t[1],p) &&
    !CGAL::right_turn(t[2],t[3],p) &&
    !CGAL::right_turn(t[4],t[5],p);
}

void find_cover()
{
```

```
  // read input
  std::size_t m, n;
  std::cin >> m >> n;

  Pts path;
  path.reserve(m);
  for (std::size_t i = 0; i < m; ++i) {
    int x, y;
    std::cin >> x >> y;
    path.push_back(K::Point_2(x,y));
  }

  // store indices of all legs that are covered by triangle i
  Trs triangles(n);
  for (std::size_t i = 0; i < n; ++i) {
    Pts t;
    for (std::size_t j = 0; j < 6; ++j) {
      int x, y;
      std::cin >> x >> y;
      t.push_back(K::Point_2(x,y));
    }
    // ensure correct (ccw) order for orientation tests
    for (std::size_t j = 0; j < 6; j+=2)
      if (CGAL::right_turn(t[j],t[j+1],t[(j+2)%6])) std::swap(t[j],t[j+1]);
    // store which path segments are covered
    bool prev = contains(t,path[0]);
    for (std::size_t j = 1; j < m; ++j)
      if (contains(t,path[j])) {
        if (prev) triangles[i].push_back(j-1); else prev = true;
      } else
        prev = false;
  }

  // search for the cover by scanning through the sequence of triangles
  Covered covered(m-1,0); // #times i,i+1 is covered
  std::size_t uncovered = m-1; // #uncovered segments (covered[i]==0)
  std::size_t best = n; // size of best range so far
  for (std::size_t tb = 0, te = 0; tb != triangles.size();) {
    // ensure covering
    for (; uncovered > 0 && te != triangles.size(); ++te)
      for (CCI j = triangles[te].begin(); j != triangles[te].end(); ++j)
        if (++covered[*j] == 1) --uncovered;
    if (uncovered != 0) break;
    // can we remove tb?
    do
      for (CCI j = triangles[tb].begin(); j != triangles[tb].end(); ++j)
        if (--covered[*j] == 0) ++uncovered;
    while (++tb != te && uncovered == 0);
    best = std::min(best,te-tb+1);
  }
  std::cout << best << "\n";
}

int main()
{
  std::ios_base::sync_with_stdio(false);
  int t;
  for (std::cin >> t; t > 0; --t) find_cover();
}
```