Discrete Comput Geom (2014) 52:492–514
DOI 10.1007/s00454-014-9625-2

A Faster Algorithm for Computing Motorcycle Graphs

Antoine Vigneron · Lie Yan

Received: 10 September 2013 / Revised: 27 March 2014 / Accepted: 8 July 2014 /
Published online: 29 August 2014

Abstract We present a new algorithm for computing motorcycle graphs that runs in $O(n^{4/3+\varepsilon})$ time for any $\varepsilon > 0$, improving on all previously known algorithms. The main application of this result is to computing the straight skeleton of a polygon. It allows us to compute the straight skeleton of a non-degenerate polygon with h holes in $O(n\sqrt{h+1}\log^2 n + n^{4/3+\varepsilon})$ expected time. If all input coordinates are $O(\log n)$-bit rational numbers, we can compute the straight skeleton of a (possibly degenerate) polygon with h holes in $O(n\sqrt{h+1}\log^3 n)$ expected time. In particular, it means that we can compute the straight skeleton of a simple polygon in $O(n\log^3 n)$ expected time if all input coordinates are $O(\log n)$-bit rationals, while all previously known algorithms have worst-case running time $\omega(n^{3/2})$.

Keywords Algorithms design and analysis · Motorcycle graph · Straight skeleton · Medial axis · Polygon

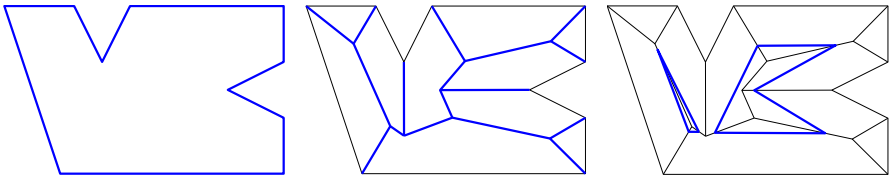
Mathematics Subject Classification 68U05 · 65D18 · 68Q25

1 Introduction

The straight skeleton of a polygon P is a straight line graph embedded in P, formed by the traces of the vertices of P when it is shrunk, each edge moving at the same

A. Vigneron
Visual Computing Center, King Abdullah University of Science
and Technology (KAUST), Thuwal, 23955-6900, Saudi Arabia
e-mail: antoine.vigneron@kaust.edu.sa

L. Yan
Department of Computer Science and Engineering, The Hong Kong University of Science
and Technology (HKUST), Clear Water Bay, Kowloon, Hong Kong
e-mail: l.yan@ust.hk



(a) Input polygon **(b)** Straight skeleton **(c)** Offset polygon

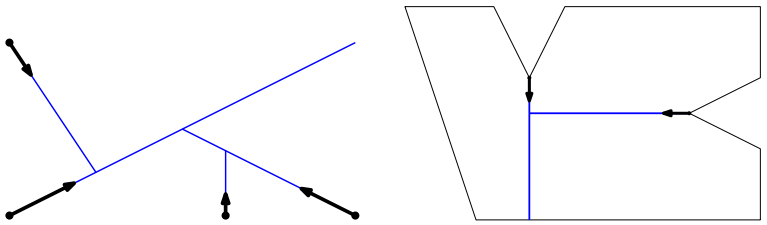
Fig. 1 The straight skeleton of a polygon, and a corresponding offset polygon **a** input polygon, **b** straight skeleton, **c** offset polygon

speed and remaining parallel to its original position (see Fig. 1). It has been known since at least the 19th century; for instance, figures representing the straight skeleton can be found in the book by von Peschka [34]. Aichholzer et al. [5] gave the first efficient algorithms for computing the straight skeleton, and presented it as an alternative to the medial axis having only straight-line edges. The straight skeleton has found numerous applications in computer science, for instance to city model reconstruction [29], architectural modeling [28], polyhedral surface reconstruction [7,21,31], biomedical image processing [16]. It also has a direct application to CAD, as it allows to compute offset polygons [18]. The straight skeleton has become a standard tool in geometric computing, and thus fast and robust software has been developed to compute it [10,27,32].

The complexity of straight skeleton computation, however, is still very much open. The previously best known algorithm is the $O(n^{17/11+\varepsilon})$-time algorithm by Eppstein and Erickson [18], and for non-degenerate polygons, the $O(n^{3/2}\log^2 n)$-time randomized algorithm by Cheng and Vigneron [15]. The only known lower bound is $\Omega(n\log n)$, by a reduction from sorting [20,25]. In this paper, we give new subquadratic algorithms for computing straight skeletons. In particular, if all input coordinates are $O(\log n)$-bit rational numbers, we give an $O(n\sqrt{h+1}\log^3 n)$-time randomized algorithm for computing the straight skeleton of a polygon with h holes. It is the first near-linear time algorithm for computing the straight skeleton of a simple polygon.

Eppstein and Erickson [18] introduced *motorcycle graphs* so as to model the main difficulty of straight skeleton computation. We are given a set of n motorcycles, each motorcycle having a starting point and a velocity. Each motorcycle moves at constant velocity until it reaches the track left by another motorcycle, in which case it crashes. The resulting graph is called a motorcycle graph (see Fig. 2a). The motorcycle graph is a special case of the straight skeleton, where each motorcycle is modeled by a small and thin triangle. Conversely, a polygon induces a motorcycle graph, where each motorcycle starts at a reflex vertex and moves with the same velocity as this vertex moves during the shrinking process (see Fig. 2b). Cheng and Vigneron [15] showed that computing the straight skeleton of a non-degenerate polygon reduces to computing this induced motorcycle graph, and a lower envelope computation; Huber and Held extended this proof to degenerate cases, including degree-1 vertices in the input polygon [27]. The lower envelope computation can be done in $O(n\sqrt{h+1}\log^2 n)$ expected time if P has h holes.

Previously, the bottleneck of straight skeleton computation was the induced motorcycle graph computation. This is our main motivation for designing a faster motorcycle



(a) A motorcycle graph **(b)** Induced motorcycle graph

Fig. 2 The motorcycle graph of a set of four motorcycles (**a**), and the motorcycle graph induced by the polygon from Fig. 1, **a** a motorcycle graph, **b** induced motorcycle graph

graph algorithm. In this paper, we give an algorithm for computing a motorcycle graph that runs in $O(n^{4/3+\varepsilon})$ time, for any $\varepsilon > 0$, improving on all previously known algorithms. Here is a brief description of our algorithm. For each motorcycle, we maintain a tentative track, which may be longer than its actual track in the motorcycle graph. We also maintain a set of target points, which contains the endpoints of the tentative tracks that have been created earlier for this motorcycle, and that it has not reached yet. Initially, the tentative tracks are empty, and then we try to extend them one by one, all the way to the destination point. If two tentative tracks cross, we retract them, by roughly halving the number of possible crossing points on each of them. After performing this halving, the tentative tracks do not intersect, and we can safely move the motorcycle that reaches the end of its tentative track first. Then we try to extend the tentative track of this motorcycle to its next target point, and repeat the process. An example is given in Appendix 2.

Apart from obtaining better time bounds for straight skeleton computation, there are at least two other reasons for studying motorcycle graphs. First, Huber and Held [27] used the idea of computing the straight skeleton from its induced motorcycle graph to design and implement a practical straight skeleton algorithm. So it is important, even in practice, to get a better understanding of motorcycle graph computation. Another motivation for studying motorcycle graphs is a direct application to computer graphics, for quad mesh partitioning [19].

Some of our results make no particular assumptions on the input, but we also present a few results where we assume that the input coordinates are $O(\log n)$-bit rational numbers. We believe that this assumption is sufficient for most applications. For instance, in the applications mentioned above, it is hard to imagine that the input polygons would have features smaller than 1nm, and size larger than 1,000 km, so 64-bit integers should be more than sufficient.

1.1 Summary of Our Results and Comparison with Previous Work

The main novelty in this paper is our algorithm for computing motorcycle graphs (Sect. 2.1). This algorithm is essentially different from the two previous algorithms [15, 18] that both simulate the construction in chronological order. Our algorithm, on the other hand, does not construct the motorcycle tracks in chronological order: It may move some motorcycle to its position at time t, and then later during the execution of the algorithm, move another motorcycle to its position at an earlier time $t' < t$.

(We give one such example in Appendix 2.) This answers an open question by Eppstein and Erickson [18, end of Section 5], who asked whether the running time can be improved by relaxing the chronology of the events.

Our algorithm uses two auxiliary data structures, one for ray shooting, and another for halving queries. Given a query segment on the supporting line of a motorcycle, a halving query returns a splitting point on this segment such that there are roughly the same number of intersections with other supporting lines on both sides (see Sect. 1.2). The implementation of these data structures in different settings, which will be described later, lead to different time bounds.

For all our results, we use the standard real-RAM model [33], that allows to perform arithmetic operations exactly on arbitrary real numbers. But for some of our results, we make the assumption that all input coordinates are $O(\log n)$-bit rational numbers. It has two advantages: it yields better time bounds, and allows us to handle the straight skeleton of degenerate polygons. This improvement comes from the fact that, for bounded precision input, two distinct crossing points between the supporting lines of two pairs of motorcycles are at distance $2^{-O(\log n)}$ from each other. It allows us to use a simpler halving scheme: Instead of halving a segment according to the number of intersection points, we use the midpoint according to the Euclidean distance (see Sects. 3.3, 4.1).

Our motorcycle graph algorithm is simple and implementable, assuming that the auxiliary data structures are available. In addition to these data structures, we only use one priority queue for each motorcycle.

Arbitrary precision input For our first set of results, the input coordinates are arbitrary real numbers, on which we can perform exact arithmetic operations. In this case, our new algorithm computes a motorcycle graph in $O(n^{4/3+\varepsilon})$ time (Theorem 2). This improves on the two subquadratic algorithms that were known before: the $O(n^{17/11+\varepsilon})$-time algorithm by Eppstein and Erickson [18], which was first published in 1998, and the $O(n^{3/2}\log n)$-time algorithm by Cheng and Vigneron [15], which first appeared in 2002.

We also give, in Sect. 3.2, an $O(Cn\log^2(n)\min(C,\log n))$-time algorithm for the case of C-oriented motorcycles, where the velocities take only C different directions. This improves on the algorithm by Eppstein and Erickson [18], which runs in $O(n^{4/3+\varepsilon})$ time when $C = O(1)$.

Our last result with arbitrary precision input is an $O(n^{4/3+\varepsilon} + n\sqrt{h+1}\log^2 n)$ expected time algorithm for computing the straight skeleton of a polygon with n vertices and h holes. (This result does not hold for a degenerate polygon where two reflex vertices may collide during the shrinking process, as in Fig. 3). It improves on the algorithm by Cheng and Vigneron [15] which runs in $O(n^{3/2}\log(n)+n\sqrt{h+1}\log^2 n)$ expected time. It also improves on the $O(n^{17/11+\varepsilon})$ time bound of the algorithm by Eppstein and Erickson [18], but their algorithm is deterministic and applies to degenerate cases.

Bounded precision input The following results hold when all input coordinates are $O(\log n)$-bit rational numbers. There has been recent interest in studying computational geometry problems under a bounded precision model (the word RAM), for

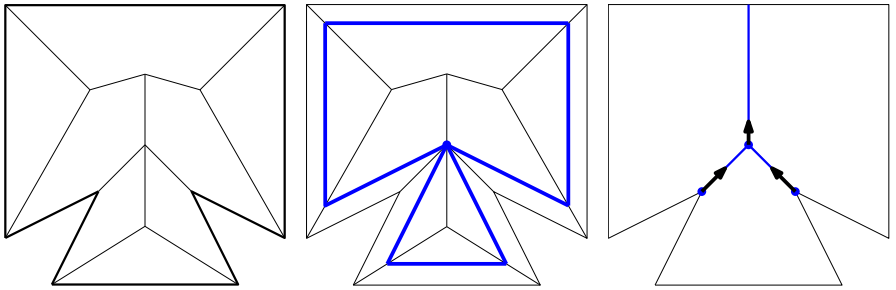


Fig. 3 A degenerate polygon and its straight skeleton (*left*). Two reflex vertices collide during the shrinking process, and a new reflex vertex appears (*middle*). The induced motorcycle graph, where a new motorcycle appears when the two other crash (*right*)

instance the computation of Delaunay triangulations, convex hulls, polygon triangulation and line segment intersections [8,12].

We first show in Sect. 3.3 that a motorcycle graph can be computed in $O(n \log^3 n)$ time if the motorcycles move within a simple polygon, starting from its boundary. The only other non-trivial cases where we know how to compute a motorcycle graph in near-linear time seem to be the case where all velocities have positive x-coordinate, which can be solved in $O(n \log n)$ time by plane sweep, the case of a constant number of different velocity vectors [18], or a constant number of directions (Sect. 3.2), and the part of the motorcycle graph outside the convex hull of the starting points [26].

Then in Sect. 4.2, we show that the straight skeleton of a polygon with n vertices and h holes can be computed in $O(n\sqrt{h+1} \log^3 n)$ expected time. This result still holds in degenerate cases. So with bounded-precision input, and if we allow randomization, it improves on the $O(n^{17/11+\varepsilon})$-time algorithm by Eppstein and Erickson [18]. When $h = o(n/\log^2 n)$, it also improves on the $O(n^{3/2} \log^2 n)$-time algorithm by Cheng and Vigneron [15], which cannot handle all degenerate cases.

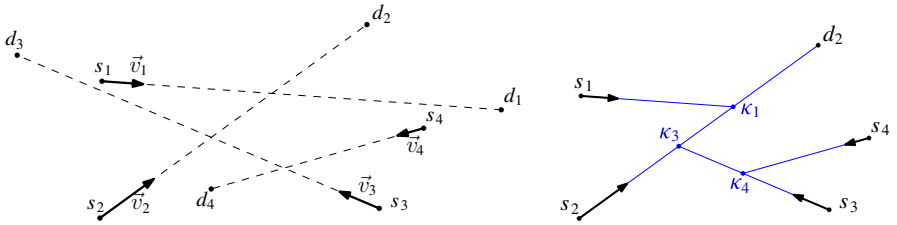
In particular, our algorithm runs in expected $O(n \log^3 n)$ time when $h = 0$, so it is the first near-linear time algorithm for computing the straight skeleton of a simple polygon. The previously best known algorithms run in $\omega(n^{3/2})$ time in the worst case [15,18].

1.2 Notation and Preliminaries

For any two points p, q, we denote by $\overline{pq}$ the line segment between p and q. Unless specified otherwise, $\overline{pq}$ is a closed segment. The *relative interior* of $\overline{pq}$ is the open segment $\overline{pq} \setminus \{p, q\}$. We say that two segments *cross* if their relative interiors intersect.

The motorcycles are numbered from 1 to n. Each motorcycle i has a *starting point* s_i, moves with constant velocity $\mathbf{v}_i$, and has a *destination point* d_i that lies in the ray $(s_i, \mathbf{v}_i)$ (see Fig. 4a). When $p \in \overline{s_i d_i}$, we denote by $\tau(i, p)$ the time when motorcycle i reaches p, so $p = s_i + \tau(i, p)\mathbf{v_i}$. The *supporting line* ℓ_i of motorcycle i is the line through s_i with direction $\mathbf{v}_i$.

Each motorcycle i starts at s_i at time 0, and moves at velocity $\mathbf{v}_i$ until it meets the track left by another motorcycle and crashes, or it reaches d_i and stops. So motorcycle



(a) Input **(b)** Motorcycle graph

Fig. 4 The input to the motorcycle graph problem (**a**), and the resulting motorcycle graph (**b**). **a** Input, **b** motorcycle graph

i crashes if it reaches a point p such that $\tau(i, p) \geqslant \tau(j, p)$, for some motorcycle j that has not crashed or stopped earlier than $\tau(j, p)$. If motorcycle i crashes, we denote by κ_i the point where it crashes, called the *crashing point* (see Fig. 4b). Otherwise, i reaches d_i, and we set $\kappa_i = d_i$. The *trajectory* of i is the segment $\overline{s_i \kappa_i}$; in other words it is the track of i in the motorcycle graph.

The motorcycle graph problem is to compute all the crashing points, given the starting points s_i, the velocities v_i, and the destination points d_i.

In the original motorcycle graph problem, the destination point d_i is at infinity in direction $\mathbf{v}_i$. We can handle this case by computing a bounding box that includes all the vertices of the arrangement of the supporting lines ℓ_i, $i = 1, \ldots, n$, and choosing as destination points the intersections of the rays $(s_i, \mathbf{v}_i)$ with the bounding box. The bounding box can be computed in $O(n \log n)$ time as any extreme vertex in the arrangement is the intersection of two lines with consecutive slopes. The part of the original motorcycle graph that lies outside the bounding box is trivial, as motorcycle tracks can only meet inside the box.

Unless specified otherwise, we make the following general position assumptions. No two motorcycles share the same supporting line, or have parallel supporting lines. No three supporting lines are concurrent. No point s_i, d_i lies on ℓ_j if $j \neq i$. No two motorcycles reach the same point at the same time. (We make these assumptions so as to simplify the description of the algorithm and the proofs, but our results still hold in degenerate cases).

The *crossing point* χ_{ij} is the intersection between ℓ_i and ℓ_j, and thus $\chi_{ij} = \chi_{ji} = \ell_i \cap \ell_j$. The *size* $|pq|$ of a segment $\overline{pq}$ is the number of crossing points χ_{ij} that lie in $\overline{pq}$ (see Fig. 5). We will need a data structure to answer *halving queries*: given a query (i, p, q) where p, q are points on the supporting line ℓ_i, find a point $h = h(p, q) \in \overline{pq}$ such that $|ph| \leqslant \lceil \rho |pq| \rceil$ and $|hq| \leqslant \lceil \rho |pq| \rceil$, for a constant $\rho < 1$. In addition, we require that h is not a crossing point, and that both $|ph|$ and $|hq|$ are strictly smaller than $|pq|$ if $|pq| \geqslant 2$.

2 Algorithm for Computing Motorcycle Graphs

In this section, we present our algorithm for computing motorcycle graphs, as well as its proof of correctness and analysis. An example of the execution of this algorithm on a set of four motorcycles is given in Appendix 2.

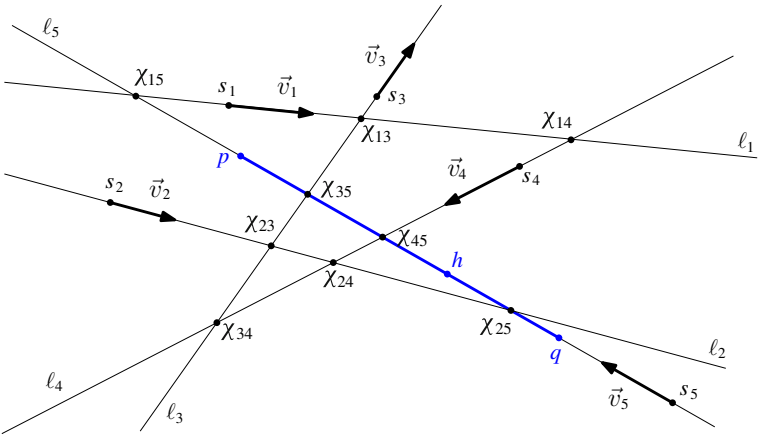


Fig. 5 The size of $\overline{pq}$ is $|pq| = 3$. Point h is a possible result $h(p, q)$ of a halving query $(5, p, q)$ with $\rho = 1/2$

2.1 Algorithm Description

Our algorithm maintains, for each motorcycle i, a *confirmed track* $\overline{s_i c_i}$, and a *tentative track* $\overline{s_i t_i}$, such that $c_i \in \overline{s_i d_i}$ and $t_i \in \overline{c_i d_i}$. So the tentative track is at least as long as the confirmed track. As we will show in the next section, the confirmed track is a subset of the trajectory, so we have $c_i \in \overline{s_i \kappa_i}$ at any time during the execution of the algorithm. The tentative track, however, may go beyond κ_i (see Appendix 2).

Our algorithm will ensure that no two tentative tracks cross. We keep all the tentative tracks in a ray shooting data structure, so that we can enforce this invariant by checking for intersection each time we try to extend a tentative track. This data structure returns the first tentative track hit by a query ray $(p, \mathbf{v})$, if any. We also build a data structure to answer halving queries, which will be used to shorten tentative tracks and keep them disjoint.

Our algorithm builds the motorcycle graph by extending the confirmed tracks until they form the whole motorcycle graph. We may also update the tentative track of a motorcycle when we extend its confirmed track. A set of *target points* is associated with each motorcycle i. In particular, we maintain in a stack S_i the set of target points that lie beyond the confirmed track of motorcycle i, thus $S_i \subset \overline{c_i d_i} \setminus \{c_i\}$. In other words, S_i records the target points that motorcycle i has not reached yet (see Fig. 6). The stack S_i is ordered from c_i to d_i. We denote by $\mathrm{Top}(S_i)$ its first element, so $\mathrm{Top}(S_i)$ is the target point in S_i that is closest to c_i. At the beginning, we set $S_i = \{s_i, d_i\}$ for all i. New target points will be created in Case (3b) of our algorithm, as described below.

If motorcycle i has neither crashed nor stopped, then its tentative track ends at the first target point in S_i, so $t_i = \mathrm{Top}(S_i)$. Otherwise, the tentative track and the confirmed track are the same, thus $t_i = c_i$. So after a motorcycle has crashed or stopped, the ray shooting data structure records its confirmed track.

An *event* (i, p) happens when a motorcycle i reaches a target point p. We process events one by one, and while an event is being processed, new events may be generated. After an event has been processed, we process the earliest available event.

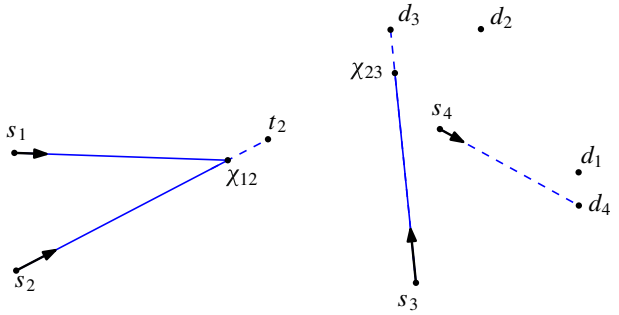


Fig. 6 This is the same example as Appendix 2, Fig. (m). The confirmed tracks are *solid*, and the tentative tracks are *dashed*. For motorcycle 1, the confirmed track and the tentative track go to $\chi_{12} = c_1 = t_1 = \kappa_1$. The stack S_1 only records d_1. For motorcycle 2, the confirmed track ends at $c_2 = \chi_{12}$, the tentative track ends at t_2, and $S_2 = (t_2, \chi_{23}, d_2)$. For motorcycle 3, we have $c_3 = \chi_{23}$, $t_3 = d_3$, and $S_3 = (d_3)$. For motorcycle 4, we have $c_4 = s_4$, $t_4 = d_4$, and $S_4 = (d_4)$

As $t_i = \text{Top}(S_i)$ is the closest target point to i in S_i, it means that we always process the event (i, t_i) such that $\tau(i, t_i)$ is smallest. Note that it does not imply that our simulation is done in chronological order: When we process an event (i, t_i), we may create a new event (j, p) such that $\tau(j, p) < \tau(i, t_i)$ (see Appendix 2).

We record in a priority queue $\mathcal{Q}$ the event (i, t_i) for each motorcycle i that has not crashed or stopped. An event with earlier time $\tau(i, t_i)$ has higher priority. As $t_i = \text{Top}(S_i)$, we can update the event queue $\mathcal{Q}$ in $O(\log n)$ time each time a stack S_i is updated. So we can extract the next available event in $O(\log n)$ time. The first n events are the events (i, s_i), $i = 1, \ldots, n$, and occur at time $t = 0$. We process these n events in an arbitrary order.

We now explain how to process an event (i, t_i). To avoid confusion, for any motorcycle j, we use the notation c_j, t_j to denote the endpoints of its confirmed and tentative track just before processing this event, and we use the notation c'_j, t'_j for their position just after processing this event. We first extend the confirmed track of motorcycle i to t_i, thus $c'_i = t_i$. We also delete t_i from S_i. We are now in one of the following cases:

(1) If $t_i = d_i$, then motorcycle i stops. In order to avoid processing irrelevant events in the future, we remove S_i from $\mathcal{Q}$.
(2) If t_i is a crossing point χ_{ij} that lies in the confirmed track of j (that is, $t_i \in \overline{s_j c_j}$), then i crashes at t_i. So we remove S_i from $\mathcal{Q}$.
(3) Otherwise, we try to extend the tentative track to the next target point $q = \text{Top}(S_i)$. So we perform a ray shooting query with ray $(t_i, \mathbf{v}_i)$, which gives us the first track intersected by $\overline{t_i q}$, if any.
   (3a) If $\overline{t_i q}$ does not cross any track, then $t'_i = q$, and we do not need to do anything else to handle this event.
   (3b) Otherwise, let j be the result of the ray-shooting query, so $\overline{s_j t_j}$ is the first track hit by segment $\overline{t_i q}$, starting from t_i. We shorten the tentative track of i, which means that we insert the new target point χ_{ij} into S_i, as well as the point $t'_i = h(t_i, \chi_{ij})$ obtained by a halving query on $\overline{t_i \chi_{ij}}$. If the crossing point χ_{ij} does not lie in the confirmed track of j, that is, if $\chi_{ij} \in \overline{c_j t_j} \setminus \{c_j\}$,

then we also shorten the tentative track of j, so we insert χ_{ij} into S_j, and we insert $t'_j = h(c_j, \chi_{ij})$ into S_j.

After applying the rules above, we update the ray shooting data structure (if needed), and we move to the next available event.

2.2 Proof of Correctness

Initially, we create the target points s_i, d_i for $i = 1, \ldots, n$. After this, we create new target points only in case (3b) of our algorithm. There are two types of such target points: the crossing points χ_{ij} obtained by ray-shooting, and the points obtained by halving queries. We call χ-targets the first type of target points, and h-targets the latter. By our assumption that the result of a halving query is not a crossing point, a target point cannot be both a h-target and a χ-target.

We need the following lemma. Remember that we say that two segments cross if their relative interiors intersect.

Lemma 1 *During the course of the algorithm, no two tentative tracks cross.*

Proof For sake of contradiction, assume that two tentative tracks cross during the course of the algorithm. Let (i, t_i) be the first event that generates such a crossing, so just before processing this event, the tentative tracks $\overline{s_j t_j}$, $j = 1, \ldots, n$ do not cross, and there is a crossing among the tracks $\overline{s_j t'_j}$. We must be in case (3), because we do not extend any tentative track in cases (1) and (2). Besides, we only extend the track of motorcycle i in case (3). So there must be another motorcycle $k \neq i$ such that $\overline{s_i t'_i}$ crosses $\overline{s_k t_k}$.

In case (3a), the segment $\overline{t_i q}$ obtained by ray shooting does not cross any tentative track $\overline{s_j t_j}$, $j \neq i$, and since $t'_i = q$, then the new portion $\overline{t_i t'_i}$ of the track does not cross any other tentative track. The same is true in case (3b), because t'_i is in $\overline{t_i \chi_{ij}}$, where track j is the first track hit by $\overline{t_i q}$. So we just proved that, in any case, the new portion $\overline{t_i t'_i}$ of the track does not cross any track $\overline{s_j t_j}$, $j \neq i$, and in particular, $\overline{t_i t'_i}$ does not cross $\overline{s_k t_k}$.

By our assumption, we also know that $\overline{s_k t_k}$ cannot cross $\overline{s_i t_i}$. So the only remaining possibility is that $\overline{s_k t_k}$ crosses $\overline{s_i t'_i}$ at t_i. Then t_i is the crossing point χ_{ik}. This point $t_i = \chi_{ik}$ cannot be in the confirmed track $\overline{s_k c_k}$, because that would be case (2) of our algorithm, and we showed that we are in case (3). Since χ_{ik} is a χ-target of i, and it does not lie in the confirmed track of k, then it must have been inserted at the same time in S_i and S_k while processing a previous event. Since χ_{ik} is not on the confirmed track of k, then it must still be in S_k. So the tentative track $\overline{s_k t_k}$ cannot contain χ_{ik} in its relative interior, a contradiction. □

We want to argue that our algorithm computes the motorcycle graph correctly. So assume it is not the case. As our algorithm moves motorcycles forward until they either reach their destination point or crash, it could only fail if during the execution of our algorithm, the confirmed track of at least one motorcycle i goes beyond the point κ_i where it is supposed to crash in the motorcycle graph. Let us consider the event (i, t_i) that is first processed by our algorithm, such that motorcycle i goes beyond κ_i. So

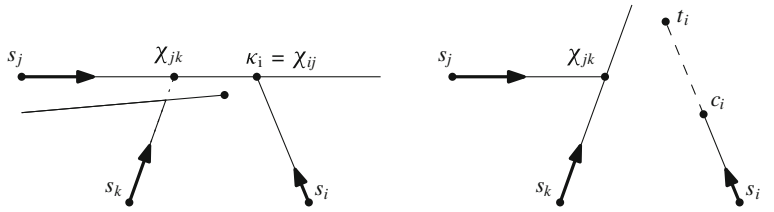


Fig. 7 The motorcycle graph (*left*) and an incorrect computation (*right*)

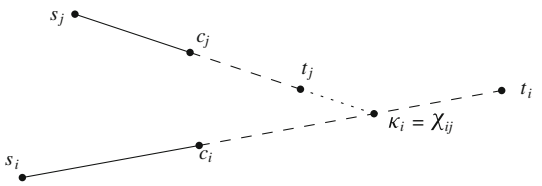


Fig. 8 Proof of correctness, remaining case

κ_i is in the segment $\overline{c_i t_i} \setminus \{t_i\}$. Let j denote the motorcycle that i crashes into, in the (correct) motorcycle graph, so $\kappa_i = \chi_{ij}$.

When we process (i, t_i), in the current graph constructed by our algorithm, motorcycle j cannot have reached d_j, because it would mean that the tentative tracks $\overline{s_i t_i}$ and $\overline{s_j d_j}$ are crossing at $\kappa_i = \chi_{ij}$, which is impossible by Lemma 1.

We now rule out the case where, when our algorithm processes (i, t_i), motorcycle j has already crashed into some motorcycle k in the graph constructed by the algorithm (see Fig. 7). For sake of contradiction, assume it did happen.

- If we had $i = k$, then χ_{ij} would have been created as a χ-target for j earlier. At this point, i had not gone past χ_{ij}, because (i, t_i) is the first such event. As $\tau(j, \chi_{ij}) < \tau(i, \chi_{ij})$, the algorithm would have moved j to χ_{ij} before i moves further, and thus j would not crash at χ_{ij}, a contradiction.
- Thus we must have $i \neq k$. As t_i is beyond $\kappa_i = \chi_{ij}$, and tentative tracks cannot cross, we must have $c_j \in \overline{s_j \chi_{ij}}$. So j crashed into k at $\chi_{jk} \in \overline{s_j \chi_{ij}}$. As in the correct motorcycle graph, j does not crash into k, it means that the algorithm has already moved k past its (correct) crashing point, which contradicts our assumption that (i, t_i) was the first such event.

We just proved that j has not stopped or crashed when the algorithm processes event (i, t_i), so at this point there should be an event (j, t_j) in the queue. By Lemma 1, the tracks $\overline{s_i t_i}$ and $\overline{s_j t_j}$ cannot cross, so we must have $t_j \in \overline{c_j \kappa_i}$ (see Fig. 8). It implies that $\tau(j, t_j) \leqslant \tau(j, \kappa_i)$. But since i crashes into j in the (correct) motorcycle graph, we must have $\tau(j, \kappa_i) < \tau(i, \kappa_i)$, thus $\tau(j, t_j) < \tau(i, \kappa_i)$. As $\kappa_i \in \overline{c_i t_i}$, we have $\tau(i, \kappa_i) \leqslant \tau(i, t_i)$, thus $\tau(j, t_j) < \tau(i, t_i)$. But this is impossible, because our algorithm always processes the earliest available event, so it would have processed (j, t_j) rather than (i, t_i).

2.3 Analysis

Our algorithm uses two auxiliary data structures: for answering halving queries, and for ray shooting. The running time of our algorithm depends on their preprocessing

time and query time. Let $P(n)$ denote an upper bound on the preprocessing time of these two data structures, and let $Q(n)$ denote an upper bound on the time needed for a query or update—so we can answer a ray-shooting query or a halving query in time $Q(n)$, and we can update the ray shooting data structure in time $Q(n)$. Let $M(n)$ denote the space requirement of these two data structures. We now prove the following result:

Theorem 1 *We can compute a motorcycle graph of size n in time $O(P(n) + n(Q(n) + \log n)\log n)$, using $O(M(n) + n\log n)$ space.*

Each time we handle an event, we perform at most two halving queries, one ray-shooting query, and we may update two tentative tracks in the ray-shooting data structure. We also pay an $O(\log n)$ time overhead to update the priority queue $\mathcal{Q}$. So after preprocessing, the running time will be at most the number of events times $Q(n)+\log n$. The space we need, in addition to the space taken by the auxiliary data structures, is proportional to the number of events, as we only need to maintain a priority queue over these events. Thus we only need to argue that our algorithm processes a total of $O(n\log n)$ events. In fact, at each event we process, a motorcycle reaches a target point, so we only need to show that $O(n\log n)$ target points are created during the course of the algorithm.

Initially, we create $O(n)$ target points, which are s_i, d_i for $i = 1,\ldots,n$. After this, we only create new target points in case (3b) of the algorithm. In this case, we create one χ-target, and at most two h-targets obtained by halving. Thus we only need to bound the number of χ-targets. At the end of the algorithm, some of these χ-targets χ_{ij} correspond to an actual crash, with motorcycle i crashing into j, or j crashing into i. In any case, there are at most n such χ-targets. We need to consider the other χ-targets, that do not correspond to an actual crash. In this case, either motorcycle i or j does not reach χ_{ij}, so at the end of the computation, χ_{ij} must appear in the stack S_i or S_j of target points that have not been reached by motorcycle i or j, respectively. Thus, in order to complete the proof of Theorem 1, we only need the following lemma.

Lemma 2 *At the end of the execution of our algorithm, for any motorcycle i, the number of χ-targets in S_i is $O(\log n)$.*

Proof In this proof, we only consider the status of the stack S_i at the end of the algorithm, and we assume that it contains more than one χ-target. We denote by $\chi_1,\ldots,\chi_m$ the χ-targets in S_i, in reverse order, so $\chi_m\ldots\chi_2\chi_1$ is a subsequence of S_i, where χ_m is closest to κ_i and χ_1 is closest to d_i.

Each target χ_j was created in case (3b) of our algorithm. At the same time, an h-target $h_j = h(g_j,\chi_j)$ was created by a halving query using another target point g_j. As the points χ_j, $j = 1,\ldots,m$ are in S_i, motorcycle i never reaches these points during the course of the algorithm, so χ_1 and h_1 must have been created first, then χ_2 and h_2 …and finally χ_m and h_m.

For any $2 \leqslant j \leqslant m$, as χ_j is created after χ_{j-1}, and these two points are created when motorcycle i reaches g_j and g_{j-1}, respectively, it implies that g_{j-1} is in $\overline{s_i g_j}$. We also know that χ_{j-1} lies in $\overline{\chi_j d_i}$, because χ_{j-1} appears after χ_j in S_i. So $\overline{g_j\chi_j}$, $j = 1,\ldots,m$ is a sequence of nested segments, that is, we have $\overline{g_j\chi_j} \subset \overline{g_{j-1}\chi_{j-1}}$ for all $2 \leqslant j \leqslant m$. More precisely:

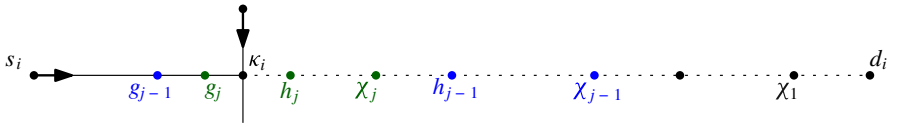


Fig. 9 Proof of Lemma 2, first case

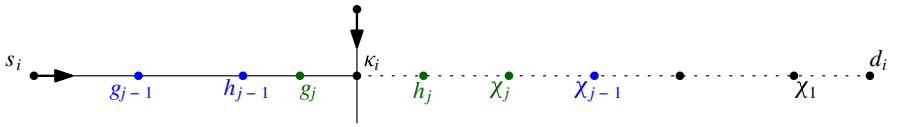


Fig. 10 Proof of Lemma 2, second case

– If h_{j-1} is in S_i, then $\overline{g_j\chi_j} \subset \overline{g_{j-1}h_{j-1}}$, because χ_j is created after h_{j-1}, and motorcycle i never reaches h_{j-1} (see Fig. 9).
– If h_{j-1} is not in S_i, then $\overline{g_j\chi_j} \subset \overline{h_{j-1}\chi_{j-1}}$ (see Fig. 10). It can be proved as follows. As h_{j-1} is created at the same time as χ_{j-1}, then χ_j is created after h_{j-1}. So χ_j must have been created after motorcycle i reaches h_{j-1}, otherwise we would have $\chi_j \in \overline{s_i h_{j-1}}$, and since motorcycle i reaches h_{j-1} later, χ_j would not be in S_i. As χ_j is created after motorcycle i reaches h_{j-1}, we must have $g_j \in \overline{h_{j-1}\chi_{j-1}}$.

Thus $\overline{g_j\chi_j}$ is contained in either $\overline{g_{j-1}h_{j-1}}$ or $\overline{h_{j-1}\chi_{j-1}}$, and since $h_{j-1} = h(g_{j-1}, \chi_{j-1})$, it follows that the size $|g_j\chi_j|$ decreases exponentially when j increases from 1 to m. As $\overline{g_{m-1}\chi_{m-1}}$ contains χ_m and χ_{m-1}, we have $|g_{m-1}\chi_{m-1}| \geqslant 2$. In addition, $|g_1\chi_1| \leqslant n$, so we must have $m = O(\log n)$. □

3 Auxiliary Data Structures

Our algorithm, presented in Sect. 2.1, requires two auxiliary structures. The first one is simply a ray-shooting data structure. As ray shooting is a standard operation in computational geometry, we will be able to directly use known data structures. The second data structure we need is for answering halving queries. We show below how to construct efficient data structures for this type of queries, and the corresponding time bounds for our motorcycle graph algorithm.

3.1 General Case

In this section, we present the auxiliary data structures for the most general case, as presented in Sect. 1.2. So motorcycles have arbitrary starting position, destination point and velocity.

For ray shooting, we can directly use a data structure by Agarwal and Matoušek [2], which requires preprocessing time $O(n^{4/3+\varepsilon})$, with update and query time $O(n^{1/3+\varepsilon})$, for any $\varepsilon > 0$.

For halving queries, we use known range searching data structures and parametric search, as in the work of Agarwal and Matoušek on ray shooting: Our problem is an optimization version of range counting in an arrangement of lines, so we obtain the same bounds [2, Section 3.1].

Lemma 3 *Given the n supporting lines $\ell_1, \ldots, \ell_n$, we can construct a data structure with $O(n^{4/3+\varepsilon})$ preprocessing time and space requirement, and $O(n^{1/3+\varepsilon})$ query time, that answers the following queries (i, p, q). Assume there are k crossing points χ_{ij} on $\overline{pq}$. Then we return the median crossing point and the next: the $\lceil k/2 \rceil$th and the $(\lceil k/2 \rceil + 1)$th such crossing point, in the ordering from p to q along $\overline{pq}$.*

With the two auxiliary data structures above, Theorem 1 yields the following result.

Theorem 2 *A motorcycle graph can be computed in $O(n^{4/3+\varepsilon})$ time, using $O(n^{4/3+\varepsilon})$ space, for any $\varepsilon > 0$.*

It should be possible to replace the n^{ε} factor in the bounds of Lemma 3 with a polylogarithm using known range searching techniques [11,30], because we only need a static data structure for halving queries, but in any case we need a dynamic data structure for ray shooting queries, so it would not improve our overall time bounds.

3.2 C-Oriented Motorcycle Graphs

We consider the special case where motorcycles can only take C different directions $\mathbf{d_1}, \ldots, \mathbf{d_C}$. Eppstein and Erickson gave an $O(n^{4/3+\varepsilon})$-time algorithm when $C = O(1)$. We show that with appropriate auxiliary data structures, we can solve this case in time $O(n \log^2 n)$. In the following, we do not assume that $C = O(1)$, so our time bounds will also have a dependency on C.

Proposition 1 *We can compute a C-oriented motorcycle graph in $O(Cn \log^2(n) \min(C, \log n))$ time.*

We use the following data structures, and then the result follows from Theorem 1.

Ray shooting data structures A first approach to answer our ray shooting queries is to use C instances of a data structure for vertical ray shooting in a planar subdivision. Several data structures are known for this problem [6], we use a data structure by Cheng and Janardan [14] that takes $O(\log^2 n)$ time per update and $O(\log n)$ time per query. So overall, we get $Q(n) = O(C \log^2 n)$ with the terminology of Theorem 1.

Alternatively, we can use $C(C-1)$ instances of a data structure for vertical ray shooting among horizontal segments. Each data structure is used to answer ray shooting queries with a given direction, into segments with another direction: We just need to change the two coordinate axes to these two directions. Using a recent result by Giyora and Kaplan [22], we obtain $Q(n) = O(C^2 \log n)$.

Halving queries Our data structure for halving queries simply consists of a sorted list of motorcycles for each direction. So for each $k \in 1, \ldots, C$, we have an array $\mathcal{A}_k$ of the motorcycles with direction $\mathbf{d}_k$, sorted according to the intercept of their supporting lines with a line orthogonal to $\mathbf{d}_k$. We now explain how to answer a halving query $\overline{pq} \subset \ell_i$.

Without loss of generality, assume ℓ_i has direction $\mathbf{d}_1$. For each direction $\mathbf{d}_k$, $k = 2, \ldots, C$, the subset of motorcycles whose supporting lines cross $\overline{pq}$ appear

in consecutive positions in $\mathcal{A}_k$. We can find the first and the last index of these lines in $O(\log n)$ time by binary search. So we obtain all the arrangement vertices in $\overline{pq}$ in $C-1$ sorted subarrays. We then compute the median m_k of each such subarray $\mathcal{A}_k \cap \overline{pq}$, and the median of these points m_k weighted by the number of points $|\mathcal{A}_k \cap \overline{pq}|$ in the corresponding subarray. This gives a halving point $h(p,q)$ with $\rho = 3/4$, because at most half of the total weight is on each side of this point, and thus at most one fourth of the points must be on each side.

The median of each subarray can be found in $O(1)$ time, and their weighted median in $O(C)$ time [17], so the query time is dominated by the C binary searches. Thus, we can answer halving queries in $O(C \log n)$ time.

3.3 Bounded Precision Input

The data structure for answering halving queries in Sect. 3.1 is quite involved. In practice, one would rather implement halving queries by simply halving the Euclidean length $\|pq\|$ instead of approximately halving the number of crossing points. Unfortunately, in the infinite precision model that is commonly used in computational geometry, this would cause the analysis of our algorithm in Lemma 2 to break down, because a stack of target points S_i may have size $\Omega(n)$ at the end of the algorithm.

Such a counterexample would require the distance between consecutive target points in S_i to become exponentially small near the crashing point, which does not seem likely to happen in practice. To formalize this idea, we make the assumption that all input numbers (the coordinates of the starting points, the destination points, and the velocities) are rational numbers, whose numerator and denominator are in $\{-2^{w-1}, \ldots, 2^{w-1}-1\}$ for some integer w. In other words, the input numbers are w-bit signed integers. We still assume that arithmetic operations between two numbers can be performed in constant time.

In this model, the coordinates of a crossing point χ_{ij} are rational numbers obtained by solving a 2×2 linear system whose coefficients are w-bit rational numbers. So the denominators of the coordinates of the crossing points are $O(w)$-bit integers and thus, any two distinct crossing points are at distance at least $2^{-O(w)}$ from each other.

Assume that we replace our halving operation, as defined in Sect. 1.2, with halving the Euclidean length. So $h(p,q)$ is the midpoint of $\overline{pq}$, which can be computed in constant time. Then any nested sequence of segments obtained by successive halving, as in the proof of Lemma 2, consists of $O(w)$ such nested segments, because we only subdivide a segment $\overline{g_j \chi_j}$ if it contains more than one crossing point, in which case it must have length $2^{-O(w)}$. So the bound on the size of S_i becomes $O(w)$, and we get the following result.

Theorem 3 *If the input coordinates are w-bit rational numbers, we can compute a motorcycle graph in time $O(nw(Q'(n) + \log n))$, where $Q'(n)$ is the time needed for updates or queries in the ray-shooting data structure. This algorithm uses $O(nw + M'(n))$ space, where $M'(n)$ is the space usage of the ray-shooting data structure.*

For bounded-precision input, the bottleneck of our algorithm has thus become the ray shooting data structure, whose update and query time bound is $O(n^{1/3+\varepsilon})$ in the

most general case. Therefore, we obtain a faster motorcycle graph algorithm if we are in a special case where faster ray-shooting data structures are known. One such case is ray-shooting in a connected planar subdivision, which can be done in $O(\log^2 n)$-time per update and query using a data structure by Goodrich and Tamassia [23]. We can use this data structure if, for instance, all motorcycles move inside a simple polygon P, starting from its boundary. (So for all i, $\overline{s_i d_i} \subset P$, and s_i is on the boundary of P.) Then we perform ray shooting in the union of the tentative tracks and the edges of P, which form a connected subdivision. It yields the following time bound.

Corollary 1 *We can compute a motorcycle graph in $O(nw \log^2 n)$ time, using $O(nw)$ space, for n motorcycles moving inside a simple polygon with $O(n)$ vertices, starting on its boundary, and if the input has w-bit rational coordinates.*

4 Application to Straight Skeleton Computation

In this section, we give new results on straight skeleton computation, using our new motorcycle graph algorithm.

4.1 Preliminaries and Non-degenerate Cases

As we mentioned in the introduction, the straight skeleton problem and the motorcycle graph problem are closely related. We now explain it in more details.

Consider the reflex (non-convex) vertices of a polygon P. When we construct the straight skeleton of P, these vertices move inward and may collide into edges, or other vertices. These events, called split events and vertex events, are the difficult part of straight skeleton computation, because they affect the topology of the shrinking polygon by splitting it, and because they are non-local: A reflex vertex may affect a chain of edges on the other side of the polygon. The other type of events, called edge events, where an edge shrinks to a point, are easily handled with a priority queue. So the interaction between reflex vertices is a crucial part in straight skeleton computation, and the motorcycle graph presented below helps to determine these interactions.

The motorcycle graph *induced* by a polygon P is such that each motorcycle starts at a reflex vertex, moves as the same velocity as the corresponding reflex vertex when we shrink P, and stops if it reaches the boundary ∂P of P (see Fig. 2).

If P is degenerate, then two reflex vertices may collide and create a new reflex vertex. In this case we need to create a new motorcycle after the collision (see Fig. 3). So when two motorcycles collide in the induced motorcycle graph, we may have to create a new motorcycle [27]. Our motorcycle graph algorithm, as described above, does not apply directly to this case, because the proof of Lemma 2 breaks down. (The reason is that S_i may hold a linear number of target points at the end of the execution of the algorithm, due to the newly created motorcycles, see Fig. 11). In Sect. 4.2, we will explain how to compute these generalized motorcycle graphs efficiently on bounded-precision input. But the following theorem still holds in degenerate cases.

Theorem 4 (Cheng and Vigneron [15], Huber and Held [27]) *The straight skeleton of a polygon P with n vertices and h holes can be computed in expected time $O(n\sqrt{h+1}\log^2 n)$ if we know the motorcycle graph induced by the vertices of P.*

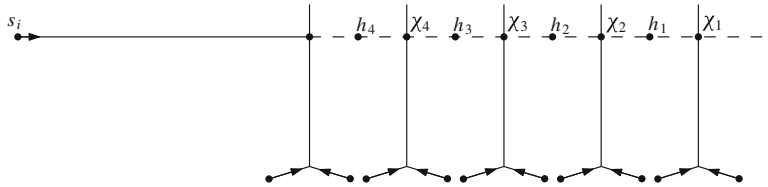


Fig. 11 An example where S_i holds a linear number of target points at the end of execution of the algorithm. The speed of the motorcycle at the bottom are adjusted so that χ_1 is created first, then $\chi_2 \ldots$

The algorithm above uses $O(n\sqrt{h+1}\log n)$ space. Thus, using our motorcycle graph algorithm from Theorem 2, we obtain the following result.

Corollary 2 *We can compute the straight skeleton of a non-degenerate polygon with n vertices and h holes in $O(n^{4/3+\varepsilon} + n\sqrt{h+1}\log^2 n)$ time, and using $O(n^{4/3+\varepsilon} + n\sqrt{h+1}\log n)$ space, for any $\varepsilon > 0$.*

4.2 Bounded Precision Input

We use the same bounded precision assumptions as in Sect. 3.3, where the input coordinates are w-bit integers or, equivalently, w-bit rational numbers. Similarly, to simplify the presentation, we use the integer model in the proofs, but we state the results in the rational model.

Thus, the coordinates of the vertices of the input polygon P are w-bit integers. The supporting lines ℓ_i of the motorcycles are angle bisectors between two edges of the input polygon. In order to apply the same halving scheme as in Sect. 3.3, where the Euclidean length is used instead of the number of arrangement vertices, we need to argue that the separation between two vertices in this arrangement of bisectors cannot be too small. This distance can be shown to be at least 2^{-W}, where $W = 64(80w + 105) + 1 = O(w)$, by applying the separation bound by Burnikel et al. [9]. So we obtain a result for induced motorcycle graphs that is analogous to Theorem 3.

Lemma 4 *Given a polygon P whose input coordinates are w-bit rational numbers, we can compute the motorcycle graph induced by P in time $O(nw(Q'(n)+\log n))$, where $Q'(n)$ is the time needed for updates or queries in the ray-shooting data structure. The space requirement is $O(nw + M'(n))$ space, where $M'(n)$ is the space usage of the ray-shooting data structure.*

In the lemma statement above, we do not exclude degenerate cases. This is another advantage of this bounded precision model. As the argument in our analysis only relies on the separation bound between two distinct crossing points, and not on the number of motorcycles crossing a given segment, a newly created motorcycle does not affect our analysis as it still obeys the same separation bound: A newly created motorcycle still lies on the bisector of two input edges, though these two edges are not adjacent in the input polygon [27] (see Fig. 3).

We still need to describe an efficient ray-shooting data structure. As our input polygon has h holes, the boundary ∂P of P together with the tentative tracks form a collection of $h+1$ disjoint simple polygons. We could directly use known ray-shooting

data structures [4,24], which can be made dynamic at the expense of an extra n^{ε} factor in the running time [1]. In the following, we give a different approach, which leads to a better time bound when used as a subroutine of our algorithm. This approach takes advantage of the fact that the holes of P are fixed (only the tentative tracks are dynamic). We use a spanning tree with low crossing number, which is not a new idea in ray-shooting data structures [13,24].

We pick one point on the boundary of each hole of P, and on the boundary of P. We connect these $h+1$ points using a spanning tree $\mathcal{T}$ with low stabbing number [3], that is, a spanning tree such that any line crosses at most $O(\sqrt{h})$ edges of T. This tree can be computed in $O(n^{1+\varepsilon})$ time [3, Section 8]. We maintain a polygonal subdivision which is the overlay of P with $\mathcal{T}$ and the tentative tracks. So at each intersection between an edge of $\mathcal{T}$ and an edge of P or a tentative track, we split the corresponding edges and tracks at the intersection point. This subdivision $\mathcal{S}$ is connected and has $O(n\sqrt{h})$ edges, and we maintain it in the ray shooting data structure by Goodrich and Tamassia [23], which has $O(\log^2 n)$ update and query time.

Each time a tentative track is extended or retracted, as a tentative track intersects $O(\sqrt{h})$ edge of $\mathcal{T}$, we can update the subdivision and the data structure by making $O(\sqrt{h})$ updates in the ray shooting data structure. Similarly, when our motorcycle graph algorithm tries to extend a tentative track, we can find the first tentative track being hit by a query ray in $O(\sqrt{h}\log^2 n)$ time: We first perform a ray shooting query in $\mathcal{S}$, which takes $O(\log^2 n)$ time. If we hit an edge of $\mathcal{T}$, we make another ray shooting query starting at the hitting point of the previous query, and in the same direction. We repeat this process as long as the result of the query is an edge of $\mathcal{T}$, and by the low-stabbing number property, it may only happen $O(\sqrt{h})$ times.

Overall, our ray shooting data structure has update and query time $O(\sqrt{h}\log^2 n)$. So by Theorem 4 and Lemma 4, we obtain the following result. Note that it still holds for degenerate input.

Theorem 5 *The straight skeleton of a polygon with n vertices and h holes, whose coordinates are w-bit rational numbers, can be computed in $O(nw\sqrt{h+1}\log^2 n)$ expected time. The space requirement is $O(n(w+\sqrt{h+1}\log n))$.*

Acknowledgments Lie Yan was supported by KAUST base funding. We thank the anonymous referees for their helpful comments.

Appendix 1. Pseudocode

In this section, we give the pseudocode of our algorithm. It is more detailed than the description in Sect. 2.1, and it can handle degenerate cases.

To deal with the degenerate cases where some supporting lines are concurrent, or two or more motorcycles reach a point at the same time, we record all the target points created so far in a dictionary data structure $\mathcal{D}$. We can implement $\mathcal{D}$ as a balanced binary search tree, sorted in lexicographical order of the coordinates (x, y), which allows to retrieve a point in $O(\log n)$ time. We associate two fields with each point p stored in $\mathcal{D}$:

– The set $M(p)$ of the motorcycles i such that $p \in S_i$. So $M(p)$ records all the motorcycles i that could possibly reach p, at a given point of the execution of our algorithm. The set $M(p)$ itself is stored in a balanced binary search tree, ordered according to $(\tau(i, p), i)$ in lexicographic order. So we can decide in $O(\log n)$ time whether a motorcycle i is in $M(p)$, and we can find another motorcycle reaches p at the same time as i if there is one.
– A flag Blocked(p) which is set to FALSE initially, and is set to TRUE as soon as a confirmed track has reached p, implying that any other motorcycle that reaches p must crash.

After the initialization stage, our algorithm handles repeatedly the earliest available event, according to the four cases (1), (2), (3a) and (3b) from Sect. 2.1.

Lines 11 and 12 deal with Cases (1) and (2). The condition $p = d_i$ corresponds to Case (1). The other two conditions check whether we are in Case (2). In particular, condition Blocked(p) = TRUE means that at least one other motorcycle has reached p, thus motorcycle i crashes. With degenerate input, it is possible that another (or several other) motorcycle reaches p at the same time as i, in which case Blocked(p) = FALSE if (i, p) is the first event involving p that has been processed. The condition at Line 12 checks whether we are in this situation. If so, i must crash. Our data structure for storing $M(p)$ allows to check this condition in $O(\log n)$ time.

At Line 18 we perform a ray shooting query from c_i in direction $\mathbf{v_i}$. If the ray does not hit any track, this query returns a point at infinity and thus the test at Line 19 is positive.

Case (3a) corresponds to a positive answer to the test at Line 19. The condition $d(s_i, p') > d(s_i, \text{Top}(S_i))$ detects whether the track of i to the next target points hits any other track. The other condition $p' = \text{Top}(S_i)$ and $j \in M(p')$, checks for a boundary case, where $\text{Top}(S_i)$ falls on another tentative track. The test is positive if p' has already been identified as a target point of j before. In this case we only extend the tentative track of i, without doing any unnecessary halving.

Line 21 branches to Case (3b). Similar to Line 19, we do not perform an unnecessary halving operation when $i \in M(p')$ or $j \in M(p')$.

Our pseudocode does not handle explicitly the case where two motorcycles have same supporting line. These cases can be easily handled by ad-hoc arguments [15]. One way of doing it is to insert additional target points at initialization. For each supporting line shared by several motorcycles, between any two consecutive motorcycles i, j along this line that go toward each other, we insert their potential collision point, that is, we insert into S_i and S_j the point p such that $\tau(i, p) = \tau(j, p)$. For each motorcycle i along this line, if the first starting point s_j in the ray $(s_i, \mathbf{v}_i)$ is in $\overline{s_i d_i}$, we also update d_i to be s_j.

Appendix 2. Example

We give an example of the execution of our algorithm on a set of 4 motorcycles. (Confirmed tracks are solid, and tentative tracks are dotted.) It demonstrates two features of our algorithm that were mentioned above:

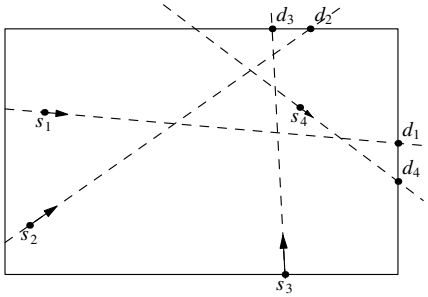
Algorithm 1 motorcycle_graph

Initialize the dictionary $\mathcal{D}$. ▷ Initialization
for $i = 1 \to n$ **do**
  Set $c_i \leftarrow s_i$, $t_i \leftarrow s_i$ and $S_i \leftarrow \{s_i, d_i\}$.
  Insert s_i and d_i into $\mathcal{D}$;
  Insert motorcycle i into $M(s_i)$ and $M(d_i)$.
Initialize the event queue Q and the data structures for ray-shooting and halving.
while Q is not empty **do** ▷ Main loop
  Let (i, p) be the earliest available event. ▷ So $p = t_i$.
  Set $c_i \leftarrow p$.
  Pop Top(S_i) from S_i. ▷ Here Top$(S_i) = p$.
  if Blocked(p) = TRUE, or $p = d_i$, or
    $\exists k \in M(p) \setminus \{i\}$ such that $\tau(k, p) = \tau(i, p)$ **then**
    Motorcycle i crashes.
    Set $t_i \leftarrow p$.
    Remove i from $M(j)$ for all motorcycles j that appear in S_i.
    Remove S_i from Q.
  else
    $(j, p') \leftarrow$ rayshooting$(c_i, \mathbf{v}_i)$.
    if $d(s_i, p') > d(s_i, \text{Top}(S_i))$, or $(p' = \text{Top}(S_i)$ and $j \in M(p'))$ **then**
      Set $t_i \leftarrow$ Top(S_i).
    else
      if $i \in M(p')$ **then**
        Set $t_i \leftarrow$ Top(S_i).
      else
        Push p' into S_i.
        Push $p'' = h(c_i, p')$ into S_i.
        Set $t_i \leftarrow p''$.
        Insert i into $M(p')$ and $M(p'')$.
      if $p' \notin \overline{s_j c_j}$ and $j \notin M(p')$ **then**
        Push p' into S_j.
        Push $p''' = h(c_j, p')$ into S_j.
        Set $t_j \leftarrow p'''$.
        Insert j into $M(p')$ and $M(p''')$.
  Set Blocked$(p) \leftarrow$ TRUE.

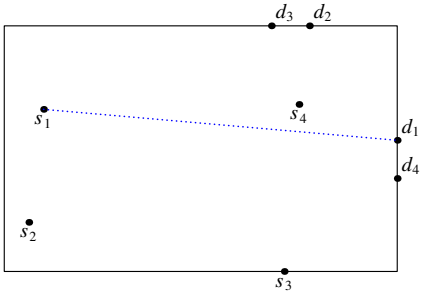
– A tentative track may be longer than the final track in the motorcycle graph. For instance, the tentative track $\overline{s_1 d_1}$ in (b) is longer than the final track $\overline{s_1 \chi_{12}}$ in (q).
– Our algorithm does not construct the motorcycle graph in chronological order. For instance, in (i), motorcycle 2 is moved to χ_{12}, which is its position at time $\tau(2, \chi_{12}) = 2.12072$. Then in (k), motorcycle 3 is moved to p_4, which is its position at time $\tau(3, p_4) = 1.667206$.

The four motorcycles 1, 2, 3 and 4 start at time 0 at initial points $s_1 = (0.8, 3.3)$, $s_2 = (0.5, 1)$, $s_3 = (5.7, 0)$ and $s_4 = (6, 3.4)$. Their velocities are $v_1 = 1.2(\cos -5°, \sin -5°)$, $v_2 = 1.7(\cos 35°, \sin 35°)$, $v_3 = 2(\cos 93°, \sin 93°)$, and $v_4 = 0.8(\cos -37°, \sin -37°)$.

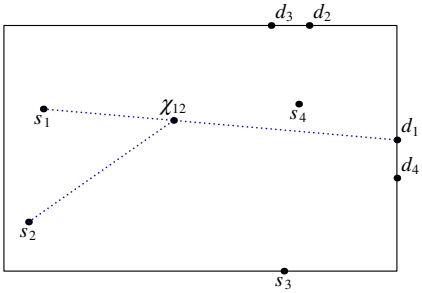
We use the halving scheme as specified in Sect. 1.2 with $\rho = 1/2$. So for instance, we create p_4 in (j) by halving $\overline{s_3 \chi_{23}}$. There are three crossings along this segment: $\chi_{13}, \chi_{23}, \chi_{34}$. Then p_4 is created as a point between χ_{13} and χ_{34}, in this case we just use the midpoint.



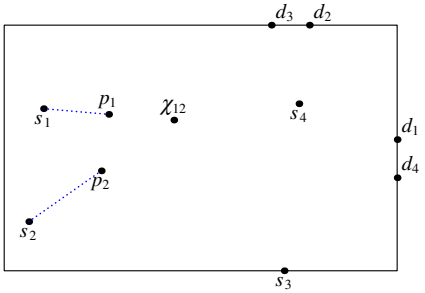
(a)



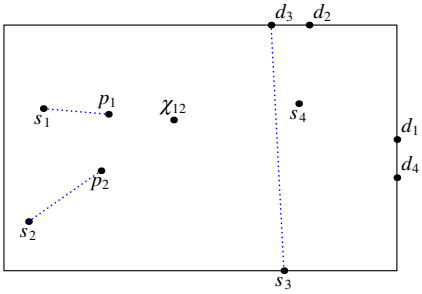
(b) $\tau(1, d_1) = 6.022919$



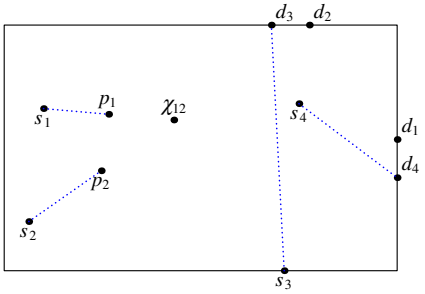
(c) new events: $\tau(1, \chi_{12}) = 2.219469$, and $\tau(2, \chi_{12}) = 2.120721$



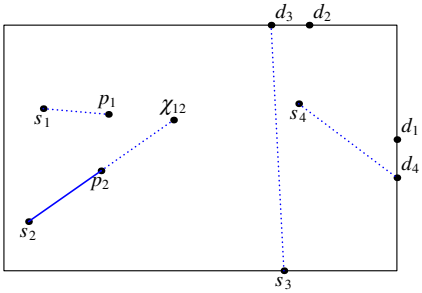
(d) new events: $\tau(1, p_1) = 1.109735$, and $\tau(2, p_2) = 1.060361$



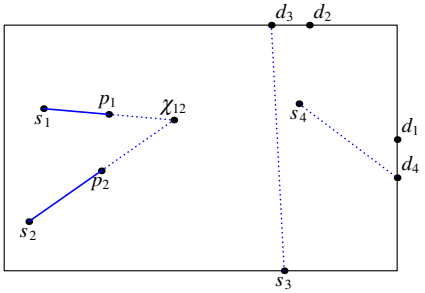
(e) $\tau(3, d_3) = 2.503431$



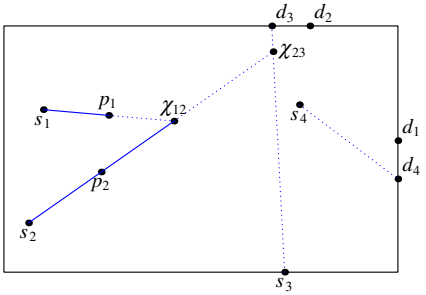
(f) $\tau(4, d_4) = 3.130339$



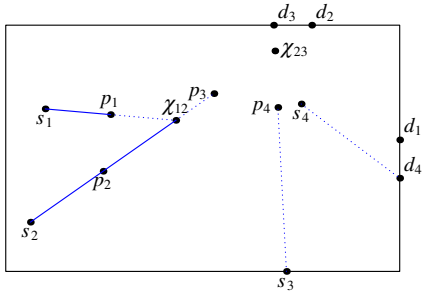
(g) move: $\tau(2, p_2) = 1.060361$



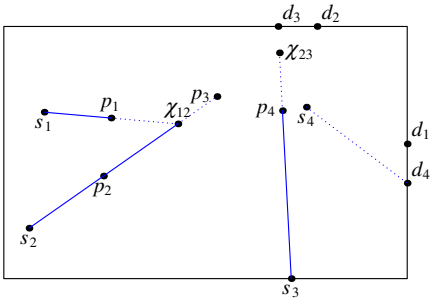
(h) move: $\tau(1, p_1) = 1.109735$



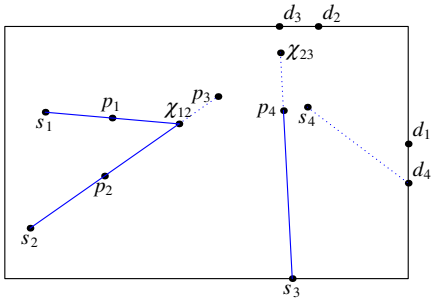
(i) move: $\tau(2, \chi_{12}) = 2.120721$. new events: $\tau(2, \chi_{23}) = 3.565653$, and $\tau(3, \chi_{23}) = 2.24147$



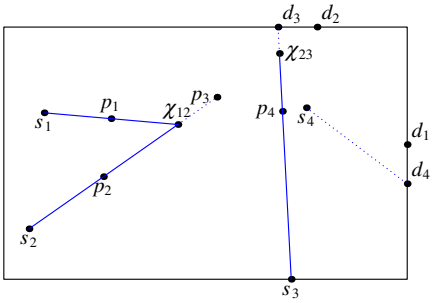
(j) new events: $\tau(2, p_3) = 2.676739$, and $\tau(3, p_4) = 1.667206$



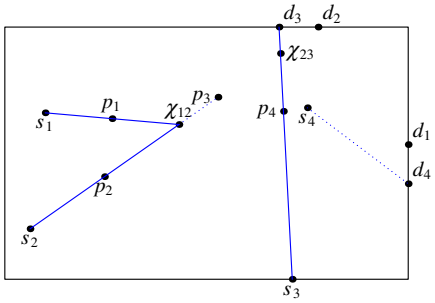
(k) move: $\tau(3, p_4) = 1.667206$



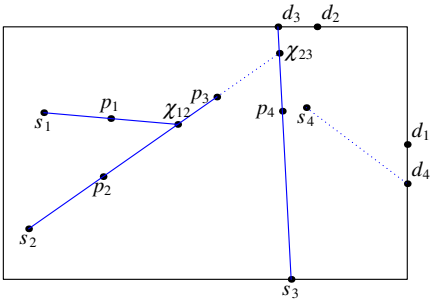
(l) move: $\tau(1, \chi_{12}) = 2.219469$



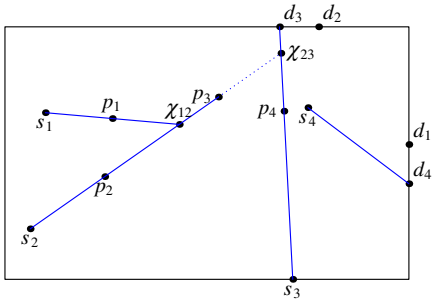
(m) move: $\tau(3, \chi_{23}) = 2.24147$



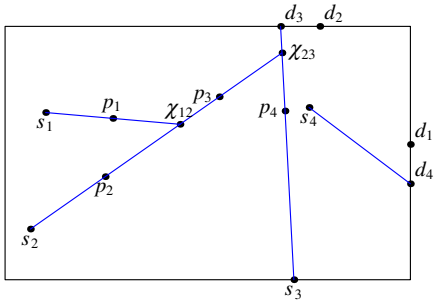
(n) move: $\tau(3, d_3) = 2.503431$



(o) move: $\tau(2, p_3) = 2.676739$



(p) move: $\tau(4, d_4) = 3.130339$



(q) move: $\tau(2, \chi_{23}) = 3.565653$

References

1. Agarwal, P., Erickson, J.: Geometric range searching and its relatives. Advances in Discrete and Computational Geometry, vol. 23, pp. 1–56. American Mathematical Society, Providence, RI (1998)
2. Agarwal, P., Matoušek, J.: Ray shooting and parametric search. SIAM J. Comput. **22**(4), 794–806 (1993)
3. Agarwal, P., Sharir, M.: Applications of a new space-partitioning technique. Discrete Comput. Geom. **9**(1), 11–38 (1993)
4. Agarwal, P., Sharir, M.: Ray shooting amidst convex polygons in 2D. J. Algorithms **21**(3), 508–519 (1996)
5. Aichholzer, O., Aurenhammer, F., Alberts, D., Gärtner, B.: A novel type of skeleton for polygons. J. Univers. Comput. Sci. **1**(12), 752–761 (1995)
6. Arge, L., Brodal, G.S., Georgiadis, L.: Improved dynamic planar point location. In: Proceedings of 47th Symposium on Foundations of Computer Science, pp. 305–314 (2006)
7. Barequet, G., Goodrich, M., Levi-Steiner, A., Steiner, D.: Straight-skeleton based contour interpolation. In: Proceedings of 14th ACM-SIAM Symposium on Discrete Algorithms, pp. 119–127 (2003)
8. Buchin, K., Mulzer, W.: Delaunay triangulations in o(sort(n)) time and more. J. ACM **58**(2), 6 (2011)
9. Burnikel, C., Fleischer, R., Mehlhorn, K., Schirra, S.: A strong and easily computable separation bound for arithmetic expressions involving square roots. In: Proceedings of 8th ACM-SIAM Symposium on Discrete Algorithms pp. 702–709 (1997)
10. Cacciola, F.: A CGAL implementation of the straight skeleton of a simple 2D polygon with holes. In: 2nd CGAL User Workshop (2004). https://www.cgal.org/UserWorkshop/2004straight_skeleton.pdf. Accessed 21 Aug 2014
11. Chan, T.: Optimal partition trees. Discrete Comput. Geom. **47**(4), 661–690 (2012)
12. Chan, T., Patrascu, M.: Transdichotomous results in computational geometry, I: point location in sublogarithmic time. SIAM J. Comput. **39**(2), 703–729 (2009)
13. Chazelle, B., Edelsbrunner, H., Grigni, M., Guibas, L., Hershberger, J., Sharir, M., Snoeyink, J.: Ray shooting in polygons using geodesic triangulations. Algorithmica **12**(1), 54–68 (1994)
14. Cheng, S.-W., Janardan, R.: New results on dynamic planar point location. SIAM J. Comput. **21**(5), 972–999 (1992)
15. Cheng, S.-W., Vigneron, A.: Motorcycle graphs and straight skeletons. Algorithmica **47**(2), 159–182 (2007)
16. Cloppet, F., Oliva, J., Stamon, G.: Angular bisector network, a simplified generalized Voronoi diagram: application to processing complex intersections in biomedical images. IEEE Trans. Pattern Anal. Mach. Intell. **22**(1), 120–128 (2000)
17. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. The MIT Press, Cambridge (2009)
18. Eppstein, D., Erickson, J.: Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. Discrete Comput. Geom. **22**(4), 569–592 (1999)
19. Eppstein, D., Goodrich, M., Kim, E., Tamstorf, R.: Motorcycle graphs: canonical quad mesh partitioning. Comput. Graph. Forum **27**(5), 1477–1486 (2008)

20. Erickson, J.: Crashing motorcycles efficiently. http://web.engr.illinois.edu/~jeffe/open/cycles.html. Accessed 21 Aug 2014 (1998)
21. Felkel, P., Obdržálek, Š.: Straight skeleton implementation. In: Proceedings of 14th Spring Conference on Computer Graphics, pp 210–218 (1998)
22. Giyora, Y., Kaplan, H.: Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. In: Proceedings of 18th ACM-SIAM Symposium on Discrete Algorithms, pp. 19–28 (2007)
23. Goodrich, M., Tamassia, R.: Dynamic ray shooting and shortest paths in planar subdivisions via balanced geodesic triangulations. J. Algorithms **23**(1), 51–73 (1997)
24. Hershberger, J., Suri, S.: A pedestrian approach to ray shooting: shoot a ray, take a walk. J. Algorithms **18**(3), 403–431 (1995)
25. Huber, S.: Computing straight skeletons and motorcycle graphs: theory and practice. PhD Thesis, Universität Salzburg, Austria (2011). http://www.buchhandel.de/detailansicht.aspx?isbn=9783844009385
26. Huber, S., Held, M.: Motorcycle graphs: stochastic properties motivate an efficient yet simple implementation. ACM J. Exp. Algorithmics **16**, Art. No. 1.3 (2011)
27. Huber, S., Held, M.: A fast straight-skeleton algorithm based on generalized motorcycle graphs. Int. J. Comput. Geom. Appl. **22**(5), 471–499 (2012)
28. Kelly, T., Wonka, P.: Interactive architectural modeling with procedural extrusions. ACM Trans. Graph. **30**(2), 14:1–14:15 (2011)
29. Laycock, R., Day, A.: Automatically generating large urban environments based on the footprint data of buildings. In: Proceedings of 8th ACM Symposium on Solid Modeling and Applications, pp. 346–351 (2003)
30. Matoušek, J.: Efficient partition trees. Discrete Comput. Geom. **8**, 315–334 (1992)
31. Oliva, J., Perrin, M., Coquillart, S.: 3D reconstruction of complex polyhedral shapes from contours using a simplified generalized Voronoi diagram. Comput. Graph. Forum **15**(3), 397–408 (1996)
32. Palfrader, P., Held, M., Huber, S.: On computing straight skeletons by means of kinetic triangulations. In: Proceedings of 20th European Symposium on Algorithms, pp. 766–777 (2012)
33. Preparata, F., Shamos, M.: Computational Geometry: An Introduction. Springer, Berlin (1985)
34. von Peschka, G.: Kotirte Ebenen. Buschak & Irrgang, Brünn (1877)