A Faster Algorithm for Computing Straight Skeletons

Siu-Wing Cheng* Liam Mencel† Antoine Vigneron‡

Abstract

We present a new algorithm for computing the straight skeleton of a polygon. For a polygon with n vertices, among which r are reflex vertices, we give a deterministic algorithm that reduces the straight skeleton computation to a motorcycle graph computation in $O(n(\log n)\log r)$ time. It improves on the previously best known algorithm for this reduction, which is randomized, and runs in expected $O(n\sqrt{h+1}\log^2 n)$ time for a polygon with h holes. Using known motorcycle graph algorithms, our result yields improved time bounds for computing straight skeletons. In particular, we can compute the straight skeleton of a non-degenerate polygon in $O(n(\log n)\log r + r^{4/3+\varepsilon})$ time for any $\varepsilon > 0$. On degenerate input, our time bound increases to $O(n(\log n)\log r + r^{17/11+\varepsilon})$.

1 Introduction

The straight skeleton $\mathcal{S}$ of a polygon $\mathcal{P}$ is defined as the trace of the vertices when the polygon shrinks, each edge moving at the same speed inwards in a perpendicular direction to its orientation. (See Figure 1.) It differs from the medial axis [10] in that it is a straight line graph embedded in the original polygon, while the medial axis may have parabolic edges. Aichholzer et al. introduced the straight skeleton in 1995, and gave the first algorithm for computing it [2]. However, the concept has been recognized as early as 1877 by von Peschka, in the author's interpretation as projection of roof surfaces [25].

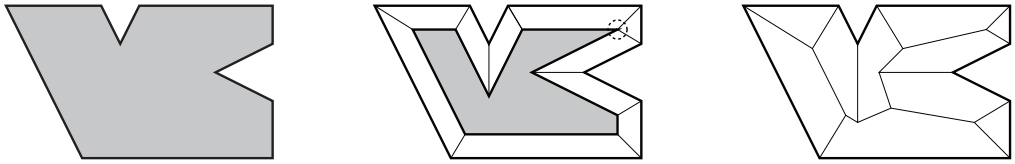
The straight skeleton has numerous applications in computer graphics. It allows one to compute offset polygons [17], which is a standard operation in CAD. Other applications include architectural modelling [23], biomedical image processing [11], city model reconstruction [13], computational origami [14, 15, 16] and polyhedral surface reconstruction [3, 12, 18]. Improving the efficiency of straight skeleton algorithms increases the speed of related tools in geometric computing.

The first algorithm runs in $O(n^2 \log n)$ time, and simulates the shrinking process discretely [2]. Eppstein and Erickson presented the fist sub-quadratic algorithm, which runs in $O(n^{17/11+\varepsilon})$ time [17]. In their work, the authors proposed motorcycle graphs as a means of encapsulating the main difficulty in computing straight skeletons. This notion was expanded on by reducing the straight skeleton problem in non-degenerate cases to a motorcycle graph computation and a lower envelope computation [9]. This reduction was later extended to degenerate cases [21]. Cheng and Vigneron gave an algorithm for the lower envelope computation on a non-degenerate polygon with h holes, which runs in $O(n\sqrt{h+1}\log^2 n)$ expected time. They also provided a method for solving the motorcycle graph problem in $O(n\sqrt{n}\log n)$ time. Putting the two together gives an algorithm which solves the straight skeleton problem in $O(n\sqrt{h+1}\log^2 n + r\sqrt{r}\log r)$ expected time, where r is the number of reflex vertices.

*Department of Computer Science and Engineering, HKUST, Hong Kong. Supported by Research Grants Council, Hong Kong, China (project no. 611711).

†Visual Computing Center, King Abdullah University of Science and Technology. Supported by KAUST base funding.

‡Visual Computing Center, King Abdullah University of Science and Technology.



(a) The input polygon $\mathcal{P}$. (b) An offset of $\mathcal{P}$. (c) Straight skeleton $\mathcal{S}$.

Figure 1: The straight skeleton is obtained by shrinking the input polygon.

	Previously best known	This paper
Arbitrary polygon	$O(n^{8/11+\varepsilon}r^{9/11+\varepsilon})$ [17]	$O(n(\log n)\log r + r^{17/11+\varepsilon})$
Non-degenerate pol.	$O^*(n\sqrt{r}\log^2 n)$ [9]	$O(n(\log n)\log r + r^{4/3+\varepsilon})$
Simple pol. arbitrary	$O^*(n\log^2 n + r^{17/11+\varepsilon})$ [9] [17]	$O(n(\log n)\log r + r^{17/11+\varepsilon})$
Simple pol. $O(\log n)$ bits	$O^*(n\log^2 n)$ [9] [24]	$O(n\log^2 n)$

Table 1: Summary of previously best known results, compared with those of our new algorithm.

Comparison with previous work Recently, Vigneron and Yan described a faster, $O(n^{4/3+\varepsilon})$-time algorithm for computing motorcycle graphs [24]. It thus removed one bottleneck in straight skeleton computation. In this paper we remove the second bottleneck: We give a faster reduction to the motorcycle graph problem. Our algorithm performs this reduction in deterministic $O(n(\log n)\log r)$ time, improving on the previously best known algorithm, which is randomized and runs in expected $O(n\sqrt{h+1}\log^2 n)$ time [9]. Using a different approach, Bowers recently claimed an $O(n\log n)$-time, deterministic algorithm to perform this reduction in the case of simple polygons, and an $O(n(\log c)\log r)$-time reduction for arbitrary polygon, where c is the number of connected components in the motorcycle graph [6]. The latter result appeared after our result was made available.

Using known algorithms for computing motorcycle graphs, our reduction yields faster algorithms for computing the straight skeleton. In particular, using the motorcycle graph algorithm by Vigneron and Yan, we can compute the straight skeleton of a non-degenerate polygon in $O(n(\log n)\log r + r^{4/3+\varepsilon})$ time for any $\varepsilon > 0$. On degenerate input, we use Eppstein and Erickson's algorithm, and our time bound increases to $O(n(\log n)\log r + r^{17/11+\varepsilon})$. For simple polygons whose coordinates are $O(\log n)$-bit rational numbers, we can compute the straight skeleton in $O(n\log^2 n)$ time using the motorcycle graph algorithm by Vigneron and Yan (even in degenerate cases). Table 1 summarizes the previously known results and compares with our new algorithm. O^* denotes the expected time bound of a randomized algorithm, and O is for deterministic algorithms. To make the comparison easier, we replaced the number of holes h with r, as $h = O(r)$. The conference version of this paper appeared in the proceedings of the European Symposium on Algorithms [8].

Our approach We use the known reduction to a lower envelope of slabs in 3D [9, 21]: First a motorcycle graph induced by the input polygon is computed, and then this graph is used to define a set of slabs in 3D. The lower envelope of these slabs is a terrain, whose edges vertically

project to the straight skeleton on the horizontal plane. (See Section 2.)

The difficulty is that these slabs may cross, and in general their lower envelope is a non-convex terrain, so known algorithms for computing lower envelopes of triangles would be too slow for our purpose. Our approach is thus to remove non-convex features: We compute a subdivision of the input polygon into convex cells such that, above each cell of this subdivision, the terrain is convex. Then the portion of the terrain above each cell can be computed efficiently, as it reduces to computing a lower envelope of planes in 3D. The subdivision is computed recursively, using a divide and conquer approach, in two stages.

During the first stage (Section 3), we partition the polygon using vertical lines, that is, lines parallel to the y-axis. At each step, we pick the vertical line ℓ through the motorcycle vertex in the current cell with median x-coordinate. We first cut the cell using ℓ, and we compute the restriction of the terrain to the space above ℓ, which forms a polyline. It can be computed in near-linear time, as it reduces to computing a lower envelope of line segments in the vertical plane through ℓ. Then we cut the cell using the steepest descent paths along the terrain, which begin from the vertices of this polyline. (See Figure 5–8.) We recurse until the current cell does not contain any motorcycle vertex in its interior. (See Figure 8b.)

The first step ensures that the cells of the subdivision are convex. However, valleys (non-convex edges) may still enter the interior of the cells. So our second stage (Section 4) recursively partitions cells using lines that split the set of valleys of the current cell, instead of vertical lines. (See Figure 9.) As the first stage results in a partition where the restriction of the motorcycle graph to any cell is outerplanar, we can perform this subdivision efficiently by divide and conquer.

Each time we partition a cell, we know which slabs contribute to the child cells, as we know the terrain along the vertical plane through the cutting line. In addition, we will argue via careful analysis that our divide and conquer approach avoids slabs being used in too many iterations, and hence the algorithm completes in $O(n(\log n)\log r)$ time.

We state here the main result of this work:

Theorem 1.1 *Given a polygon $\mathcal{P}$ with n vertices, r of which being reflex vertices, and given the motorcycle graph induced by $\mathcal{P}$, we can compute the straight skeleton of $\mathcal{P}$ in $O(n(\log n)\log r)$ time.*

Our algorithm does not handle weighted straight skeleton as defined by Eppstein and Erickson where edges move at different speeds during the shrinking process [17]. In fact, there are several ambiguities in obtaining a valid definition of weighted straight skeletons [5]. Also, it is unknown how to reduce the weighted straight skeleton construction to a lower envelope computation. Therefore, our algorithm does not apply to weighted straight skeleton.

The previous algorithm by Cheng and Vigneron also partitions the polygon into cells and then computes the portion of the straight skeleton within a cell by a lower envelope computation [9]. The challenge is to prevent a slab from contributing to too many cells, as this would entail processing the same slab too many times and result in a high running time. In the case of a simple polygon, the straight skeleton is a tree, so it is natural to divide the tree into subtrees by cutting at the "median vertex" of the tree. In fact, the tree is cut at all intersection points between the straight skeleton edges and a vertical line ℓ through the "median vertex". The polygon is cut by the steepest descending paths that start from these intersection points. This produces subtrees of sizes at least a constant factor less the original size, which leads to a divide-and-conquer recursive strategy. A polygon with holes, say h of them, is harder because its straight skeleton is not a tree any more. So the polygon is turned into a simple polygon by cutting along a spanning tree of low crossing number $O(\sqrt{h})$ that connects the hole and outer boundaries. The complication is that there are now $O(\sqrt{h})$ artificial polygon edges. Indeed, the intersections between the slabs and these artificial polygon edges lead to

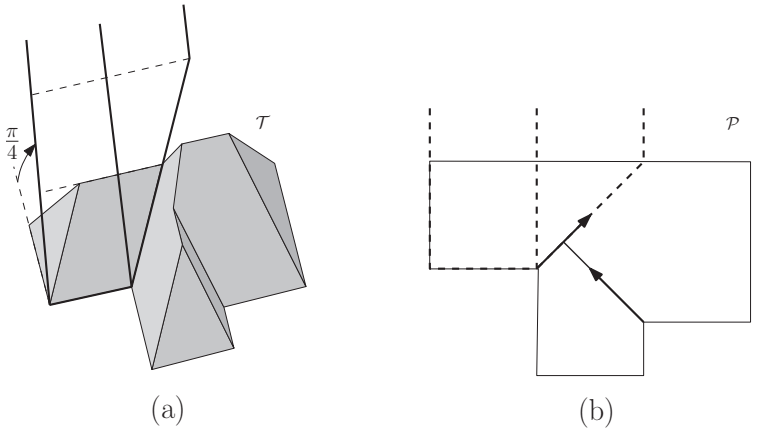


Figure 2: Illustration of the two different types of slabs. (a) The terrain $\mathcal{T}$, an edge slab and motorcycle slab. This terrain has two valleys, adjacent to the two reflex vertices of the polygon. (b) The motorcycle graph associated with $\mathcal{P}$ and the boundaries of the edge slab and the motorcycle slab viewed from above.

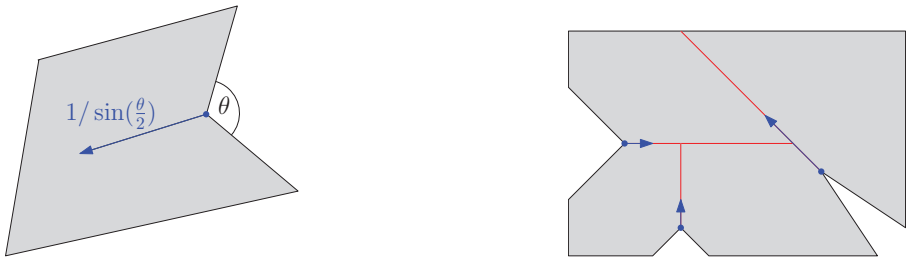
an $O(n\sqrt{h+1} \cdot \text{polylog}(n))$ running time [9]. The main tool introduced in this paper is a new decomposition of the polygon so that each slab does not contribute to too many cells, and hence we avoid this extra factor $\sqrt{h+1}$ is the running time.

2 Notation and preliminaries

A *reflex vertex* of a polygon is a vertex at which the internal angle is more than π. The input polygon is denoted by $\mathcal{P}$. It has n vertices, among which r are reflex vertices. We work in $\mathbb{R}^3$ with $\mathcal{P}$ lying flat in the xy-plane. The z-axis becomes analogous to the time dimension. We say that a line, or a line segment, is *vertical*, if it is parallel to the y-axis, and we say that a plane is vertical if it is orthogonal to the xy-plane. The boundary of a set A is denote by ∂A. The cardinality of a set A is denoted by $|A|$. We denote by $\overline{pq}$ the line segment with endpoints p, q.

Terrain At any time, the horizontal plane $z = t$ contains a snapshot of $\mathcal{P}$ after shrinking for t units of time. While the shrinking polygon moves vertically at unit speed, faces are formed as the trace of the edges, and these faces make an angle $\pi/4$ with the xy-plane. The surface formed by the traces of the edges is the *terrain* $\mathcal{T}$. (See Figure 2a.) The traces of the vertices of $\mathcal{P}$ form the set of edges of $\mathcal{T}$. The boundary edges of $\mathcal{T}$ are the edges of $\mathcal{P}$. For every non-boundary edge e of $\mathcal{T}$, there are exactly two faces of $\mathcal{T}$, say f and f', that are incident to the interior of e. We say that e is *convex* if the dihedral angle between f and f' above $\mathcal{T}$ is greater than π. The edges of $\mathcal{T}$ corresponding to the traces of the reflex vertices will be referred to as *valleys*. Valleys are the only non-convex edges on $\mathcal{T}$ ([2], Lemma 2). The other edges, which are convex, are called *ridges*. The *straight skeleton* $\mathcal{S}$ is the graph obtained by projecting the edges and vertices of $\mathcal{T}$ orthogonally onto the xy-plane. We also call valleys and ridges the edges of $\mathcal{S}$ that are obtained by projecting valleys and ridges of $\mathcal{T}$ onto the xy-plane.

Motorcycle graph Our algorithm for computing the straight skeleton assumes that a motorcycle graph induced by $\mathcal{P}$ is precomputed [9]. This graph is defined as follows. A motorcycle is a point moving at a fixed velocity. We place a motorcycle at each reflex vertex of $\mathcal{P}$. The velocity of a motorcycle is the same as the velocity of the corresponding reflex vertex when $\mathcal{P}$ is



(a) (b)

Figure 3: (a) The motorcycle (blue) associated with the reflex vertex of the shaded polygon. (b) The induced motorcycle graph (red).

shrunk, so its direction is the bisector of the interior angle, and its speed is $1/\sin(\theta/2)$, where θ is the exterior angle at the reflex vertex. (See Figure 3a.)

The motorcycles begin moving simultaneously. They each leave behind a track as they move. When a motorcycle collides with either another motorcycle's track or the boundary of $\mathcal{P}$, the colliding motorcycle halts permanently. (In degenerate cases, a motorcycle may also collide head-on with another motorcycle, but for now we rule out this case.) After all motorcycles stop, the tracks form a planar graph called the *motorcycle graph induced by* $\mathcal{P}$, denoted by $\mathcal{G}$. (See Figure 3b.)

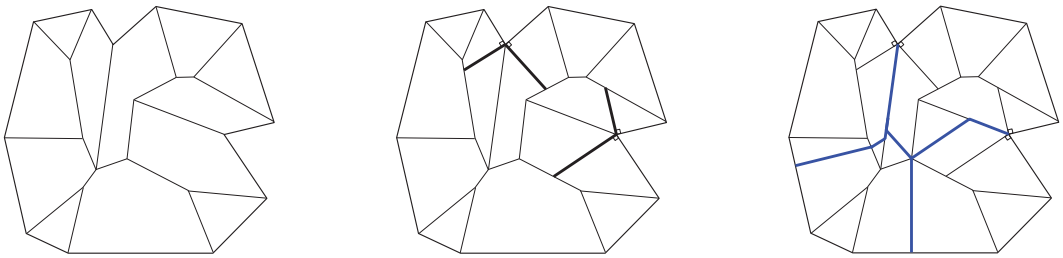
General position assumptions To simplify the description and the analysis of our algorithm, we assume that the polygon is in general position. No edge of $\mathcal{P}$ or $\mathcal{S}$ is vertical. No two motorcycles collide with each other in the motorcycle graph, and thus each valley is adjacent to some reflex vertex. Each vertex in the straight skeleton graph has degree 1 or 3. Our results, however, generalize to degenerate polygons, as explained in Section 5.

Lifting map The *lifted* version $\hat{p}$ of a point $p \in \mathcal{P}$ is the point on $\mathcal{T}$ that is vertically above p. In other words, $\hat{p}$ is the point of $\mathcal{T}$ that projects orthogonally to p on the xy-plane. We may also apply this transformation to a line segment s in the xy-plane, then $\hat{s}$ is a polyline in $\mathcal{T}$. We will abuse notation and denote by $\hat{\mathcal{G}}$ a lifted version of $\mathcal{G}$ where the height of a point is the time at which the corresponding motorcycle reaches it. Then the lifted version $\hat{e}$ of an edge e of $\mathcal{G}$ does not lie entirely on $\mathcal{T}$, but it contains the corresponding valley, and the remaining part of $\hat{e}$ lies above $\mathcal{T}$ [9]. (See Figure 2a.)

Given a point $\hat{p}$ that lies in the interior of a face f of $\mathcal{T}$, there is a unique steepest descent path from $\hat{p}$ to the boundary of $\mathcal{P}$. This path consists either of a straight line segment orthogonal to the base edge e of f, or it consists of a segment going straight to a valley, and then follows this valley [2, Theorem 7]. (In degenerate cases, the path may follow several valleys consecutively.) If $\hat{p}$ is on a ridge, then two such descent paths from p exist, and if $\hat{p}$ is a convex vertex, then there are three such paths. (See Figure 4c.)

Reduction to a lower envelope Following previous work [17, 9, 21], we use a construction of the straight skeleton based on the lower envelope of a set of three-dimensional slabs. Each edge e of $\mathcal{P}$ defines an *edge slab*, which is a 2-dimensional half-strip at an angle of $\pi/4$ to the xy-plane, bounded below by e and along the sides by rays perpendicular to e. (See Figure 2.) We say that e is the *source* of this edge slab.

For each reflex vertex $v = e \cap e'$, where e and e' are edges of $\mathcal{P}$, we define two *motorcycle slabs* making angles of $\pi/4$ to the xy-plane. One motorcycle slab is bounded below by the edge of $\hat{\mathcal{G}}$ incident to v and is bounded on the sides by two rays from each end of this edge in the



(a) The skeleton $\mathcal{S}$.

(b) The skeleton $\mathcal{S}'$. The bold segments are the flat edges.

(c) The bold polylines are the descent paths.

Figure 4: The polygon $\mathcal{P}$, its skeletons and descent paths.

ascent direction of e. The other motorcycle slab is defined similarly with e replaced by e'. The *source* of a motorcycle slab is the corresponding edge of $\hat{\mathcal{G}}$. The following result was proved in non-degenerate cases by Cheng and Vigneron, and extended to degenerate cases by Huber and Held [9, 20]:

Theorem 2.1 *The terrain $\mathcal{T}$ is the restriction of the lower envelope of the edge slabs and the motorcycle slabs to the space vertically above the polygon.*

Our algorithm constructs a graph $\mathcal{S}'$, which is obtained from $\mathcal{S}$ by adding two edges at each reflex vertex v of $\mathcal{P}$ going inwards and orthogonally to each edge of $\mathcal{P}$ incident to v. (See Figure 4b.) These extra edges are called *flat edges*. We also include the edges of $\mathcal{P}$ into $\mathcal{S}'$. It means that each face f of $\mathcal{S}'$ corresponds to exactly one slab. More precisely, a face is the vertical projection of $\mathcal{T} \cap \sigma$ to the xy-plane for some slab σ. By contrast, in the original straight skeleton $\mathcal{S}$, a face incident to a reflex vertex corresponds to one edge slab and one motorcycle slab.

3 Computing the vertical subdivision

In this section, we describe and we analyze the first stage of our algorithm, where the input polygon $\mathcal{P}$ is recursively partitioned using vertical cuts and cuts along steepest descent paths. The corresponding procedure is called Divide-Vertical, and its pseudocode can be found in Algorithm 1. It results in a subdivision of $\mathcal{P}$, such that any cell of this subdivision has the following property: It does not contain any vertex of $\mathcal{G}$ in its interior, or it is contained in the union of two faces of $\mathcal{S}'$. The second stage of our algorithm is presented in Section 4.

3.1 Subdivision induced by a vertical cut

At any step of the algorithm, we maintain a planar subdivision $\mathcal{K}(\mathcal{P})$, which is a partition of the input polygon $\mathcal{P}$ into polygonal cells. Each cell is a polygon, hence it is connected. A cell $\mathcal{C}$ in the current subdivision $\mathcal{K}(\mathcal{P})$ is subdivided as follows.

Let ℓ be a vertical line through a vertex of $\mathcal{G}$. (Remember that a vertical line is a line parallel to the y-axis. We always use vertical lines through some vertices of $\mathcal{G}$ to induce a partition of $\mathcal{P}$.) We assume that ℓ intersects the interior of $\mathcal{C}$, and hence $\mathcal{C} \cap \ell$ consists of several line segments $s_1, \ldots, s_q$. These line segments are introduced as new edges in $\mathcal{K}(\mathcal{P})$; they are called the *vertical edges* of $\mathcal{K}(\mathcal{P})$. They may be further subdivided during the course of the algorithm, and we still call the resulting edges vertical edges.

We then insert non-vertical edges along steepest descent paths, as follows. Note that we are able to efficiently compute the intersection $\mathcal{S}' \cap \ell$ without knowing $\mathcal{S}'$ (this is explained in the

ALGORITHM 1: DIVIDE-VERTICAL

Input: Cell $\mathcal{C}_i$
Output: Skeleton $\mathcal{S}'_i$
Select a median vertex in V_i, and draw the vertical line ℓ through it;
Construct the vertical edges $s_1, \ldots, s_q$ of $\ell \cap \mathcal{C}_i$;
Compute the lower envelope of the slabs along the vertical plane through ℓ;
Construct the lifted version $\hat{s}_1, \ldots, \hat{s}_q$ of the vertical boundary segments;
Trace within $\mathcal{C}_i$ the steepest descent paths from each vertex of $\hat{s}_1, \ldots, \hat{s}_q$;
Update $\mathcal{K}(\mathcal{P})$ using $s_1, \ldots, s_q$ and the descent paths as new boundaries;
for *each child cell* $\mathcal{C}_j$ *of* $\mathcal{C}_i$ **do**
    Construct the data structure for $\mathcal{C}_j$;
    if $\mathcal{C}_j$ *is a wedge or is empty* **then**
        Compute $\mathcal{S}'_j$ by brute force.
    end
    else
        if $V_j = \emptyset$ **then**
            Call DIVIDE-VALLEY($\mathcal{C}_i$)
        end
        else
            Call DIVIDE-VERTICAL($\mathcal{C}_j$).
        end
    end
end

detailed description of the algorithm). Each intersection point $p \in s_j \cap \mathcal{S}'$ has a lifted version $\hat{p}$ on $\mathcal{T}$. There are at most three steepest descent paths to $\partial\mathcal{C}$ from $\hat{p}$, which follows from our general position assumption that each vertex in $\mathcal{S}$ has degree at most 3. The vertical projections of these paths onto $\mathcal{C}$ are also inserted as new edges in $\mathcal{K}(\mathcal{P})$. The resulting partition of $\mathcal{C}$ is the *subdivision induced by* ℓ. (See Figure 6b.)

We denote by $\mathcal{C}_1, \mathcal{C}_2, \ldots$ the cells of $\mathcal{K}(\mathcal{P})$ that are constructed during the course of the algorithm. Let ℓ_i^- and ℓ_i^+ denote the vertical lines through the leftmost and rightmost point of $\mathcal{C}_i$, respectively. When we perform one step of the subdivision, each new cell lies entirely to the left or to the right of the splitting line, and thus by induction, any vertical edge of a cell $\mathcal{C}_i$ either lies in ℓ_i^- or ℓ_i^+. We now study the geometry of these cells.

Lemma 3.1 *Let p be a reflex vertex of a cell $\mathcal{C}_i$. Then p is a reflex vertex of $\mathcal{P}$ such that $\partial\mathcal{C}_i$ and $\partial\mathcal{P}$ coincide in a neighborhood of p, or p is a point where a descent path bounding $\mathcal{C}_i$ reaches a valley.*

Proof. We prove it by induction on the successive refinement of $\mathcal{K}(\mathcal{P})$ induced by vertical lines. The initial cell is $\mathcal{C}_1 = \mathcal{P}$, and hence the property holds. When we perform a subdivision of a cell $\mathcal{C}_i$ along a line ℓ, we cannot introduce reflex vertices along ℓ, as we insert the segments $\mathcal{C}_i \cap \ell$ as new cell boundaries. So new reflex vertices may only appear along descent paths. They cannot appear at the lower endpoint of a descent path, as a descent path can only meet a reflex vertex along its exterior angle bisector. So a reflex vertex may only appear in the interior of a descent path, and a descent path only bends when it reaches a valley. (This case occurs in the rightmost cell in Figure 6b.) □

The lemma above shows that non-convexity may only be introduced when a descent path bounding a cell reaches a valley. The lemma below implies that, at any point in time, this can occur only once per valley (within the cell containing the segment $\overline{bq}$ described below).

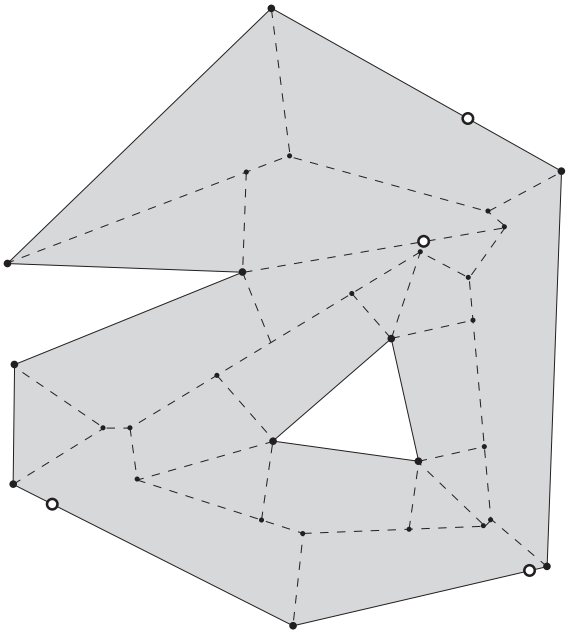
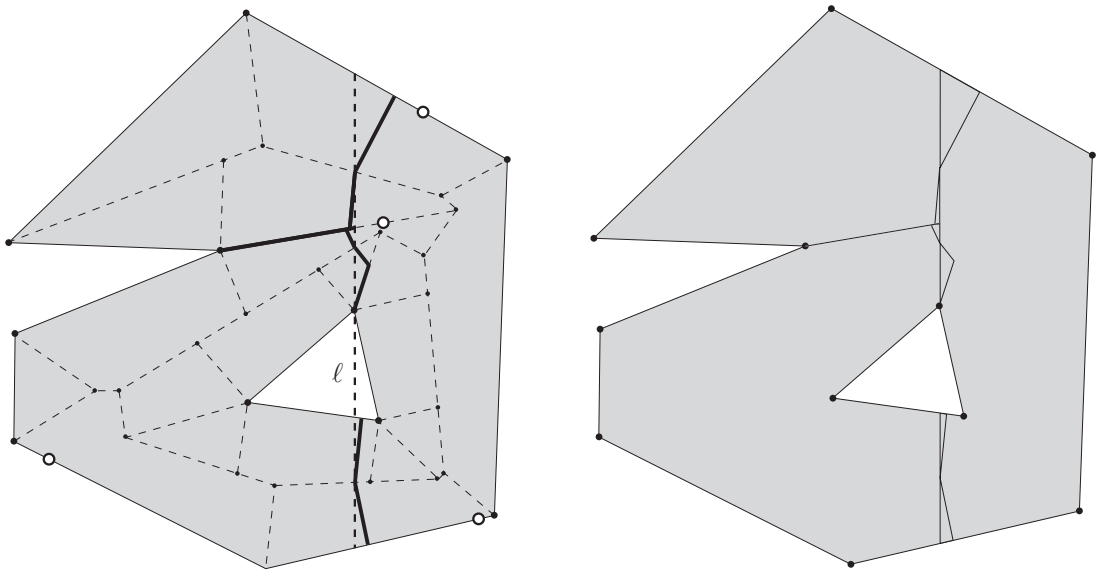
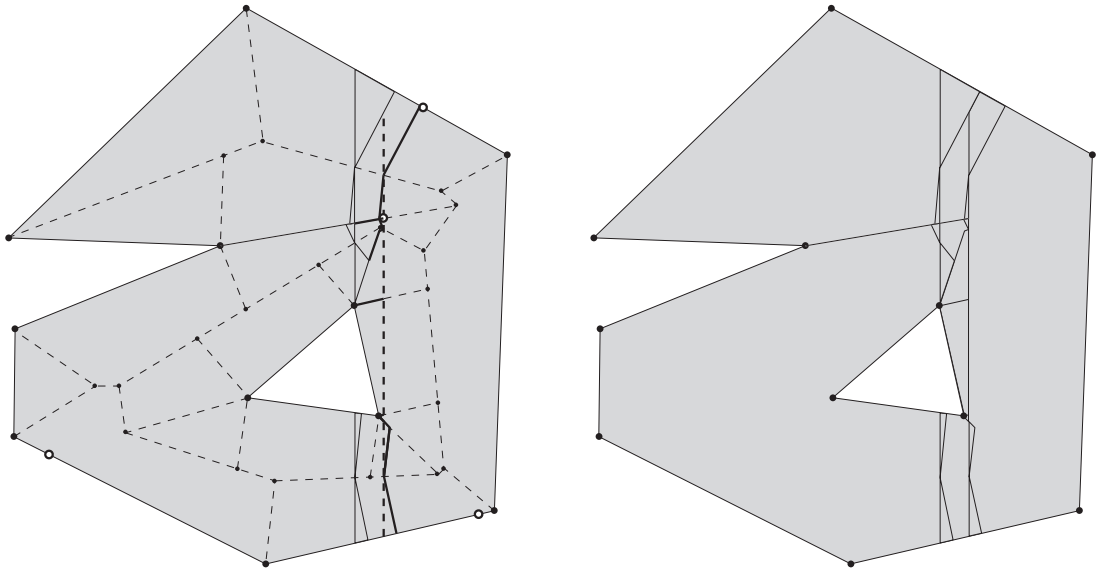


Figure 5: Input polygon $\mathcal{P}$ and the graph $\mathcal{S}'$. The white dots are the vertices of the motorcycle graph induced by the reflex vertices of the polygon. The white dots on $\partial\mathcal{P}$ are in fact terminal vertices as the corresponding motorcycles run into $\partial\mathcal{P}$.



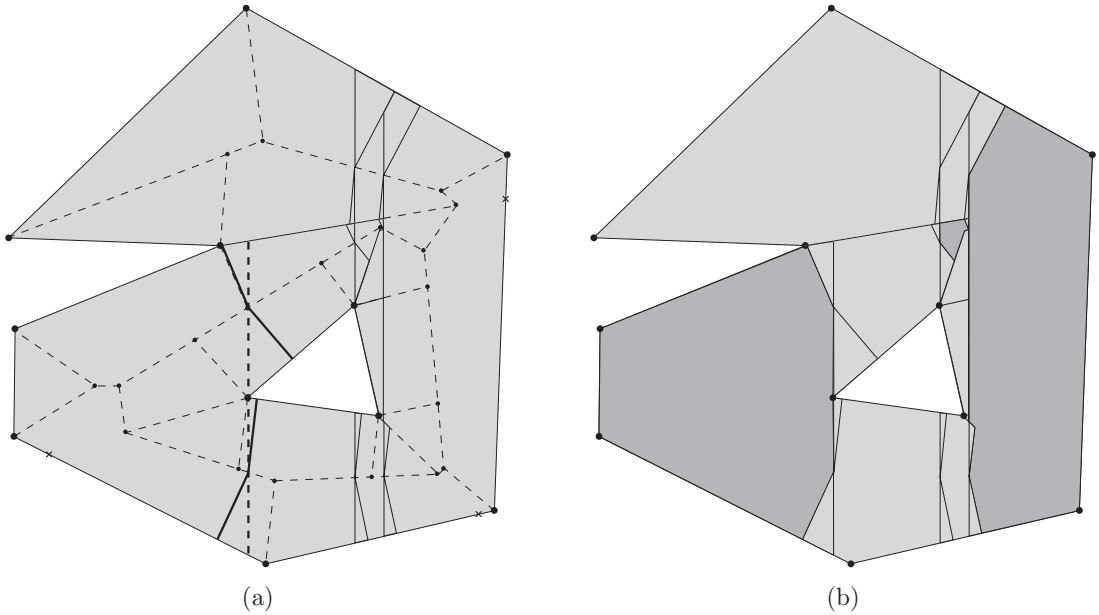
(a) (b)

Figure 6: (a) First vertical cut using the vertical bold dashed line ℓ. The bold polylines are the descent paths that start at the intersections $\mathcal{S}' \cap \ell$. (b) Subdivision of $\mathcal{P}$ into cells induced by the first vertical cut.



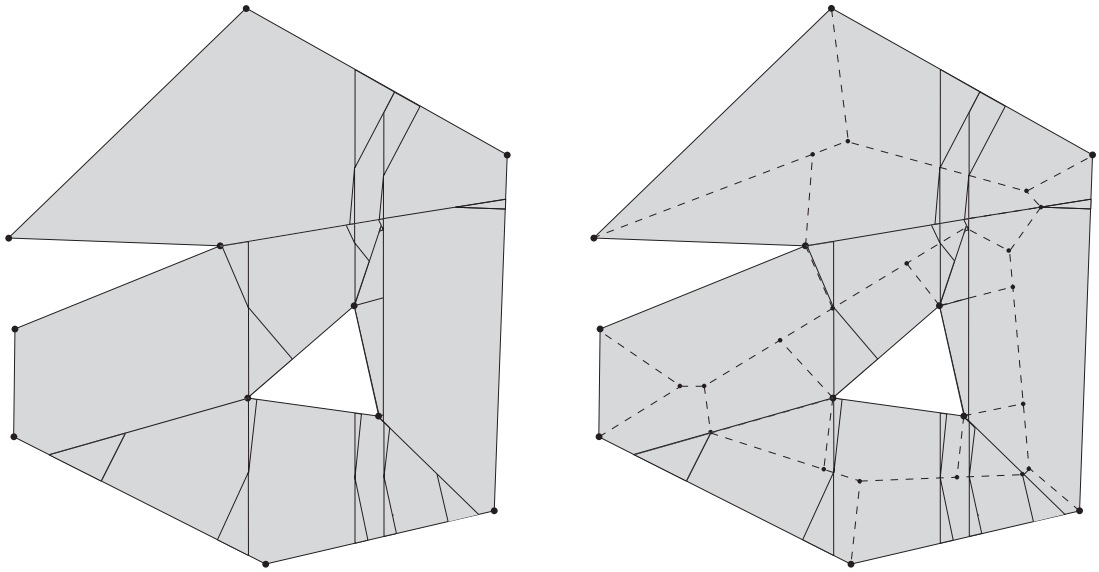
(a) (b)

Figure 7: (a) Second vertical cut using the vertical bold dashed line. The bold polylines are the descent paths that start the intersections between $\mathcal{S}'$ and the vertical line. (b) Subdivision induced by the second vertical cut.



(a) (b)

Figure 8: (a) Third vertical cut using the vertical bold dashed line. (b) Vertical subdivision computed by Divide-Vertical on the polygon in Figure 5. The darker cells are in conflict with one or more valleys, hence a non-trivial application of Divide-Valley is required.



(a) (b)

Figure 9: (a) Final subdivision computed by DIVIDE-VALLEY and (b) its overlay with $\mathcal{S}'$.

Lemma 3.2 *Let $e = \overline{pq}$ be a valley or a flat edge of $\mathcal{S}'$, with p being a reflex vertex of $\mathcal{P}$ and q being the other endpoint of e. At any time during the course of the algorithm, there is a point b along e such that $\overline{pb}$ is contained in the union of the boundaries of the cells of $\mathcal{K}(\mathcal{P})$, and the interior of $\overline{bq}$ is contained in the interior of a cell $\mathcal{C}_i$.*

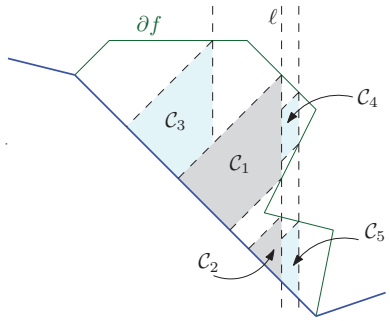
Proof. Intuitively, this lemma holds because any two descent paths that lead to the same valley must merge. We now prove it by induction on the number of steps executed in the algorithm. So we assume that at the current point of the execution of the algorithm, there is a point b on e such that $\overline{pb}$ is contained in the union of the edges of $\mathcal{K}(\mathcal{P})$, and the interior of $\overline{bq}$ is contained in the interior of a cell $\mathcal{C}_j$. At the start of the algorithm, we have $b = p$. Edge e can only intersect the interior of a new cell if this cell is obtained by subdividing $\mathcal{C}_j$. When performing this subdivision, at most two descent paths and one vertical cut can intersect $\overline{bq}$, and then the descent paths from these intersection points to b are added as cell boundaries. After that, we are again in the situation where e is split into two segments $\overline{pb'}$ and $\overline{b'q}$, with $\overline{pb'}$ being covered by edges of $\mathcal{K}(\mathcal{P})$ and $\overline{b'q}$ being in the interior of a cell. □

A ridge, on the other hand, can cross the interior of several cells. But its intersection with any given cell is a single line segment:

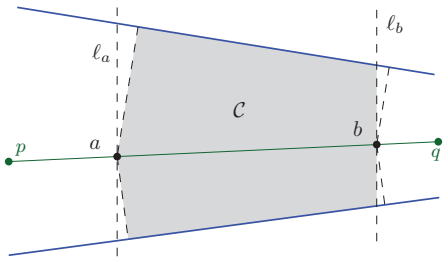
Lemma 3.3 *For any ridge e and any cell $\mathcal{C}_i$, the intersection $e \cap \mathcal{C}_i$ is a single line segment, and $e \cap \partial\mathcal{C}_i$ consists of at most two points.*

Proof. As e is a convex edge, the only descent paths that can meet e are descent paths that start from e. So e can only be partitioned by a vertical line cut through its interior. When we perform one such subdivision along a segment of e, it is split into two segments, one on each side of the cutting line, and these segments now belong to two different cells. When we repeat the process, it remains true that $e \cap \mathcal{C}_i$ is a segment, and that it can only meet $\partial\mathcal{C}_i$ at its endpoints. □

An *empty cell* is a cell of $\mathcal{K}(\mathcal{P})$ whose interior does not overlap with $\mathcal{S}'$. (See Figure 10a.) Thus an empty cell is entirely contained in a face of $\mathcal{S}'$. Another type of cell, called a *wedge*,



(a) The cells $\mathcal{C}_1, \ldots, \mathcal{C}_5$ are empty. The first cut is performed along ℓ.



(b) The wedge $\mathcal{C}$ corresponding to $\overline{ab}$.

Figure 10: Empty cells and a wedge. Blue edges belong to ∂P.

will play an important role in the analysis of our algorithm. Let $\overline{pq}$ be a ridge of $\mathcal{S}'$, and let a, b be two points in the interior of $\overline{pq}$. Let ℓ_a and ℓ_b be the vertical lines through a and b, respectively. Consider the subdivision of $\mathcal{P}$ obtained by inserting vertical boundaries along ℓ_a and ℓ_b, and the four descent paths from a and b. (See Figure 10b.) The cell of this subdivision containing $\overline{ab}$ is called the *wedge* corresponding to $\overline{ab}$. The lemma below shows that wedges are the only cells that can overlap the interior of a ridge, without containing any of the ridge's endpoints in the interior of the cell.

Lemma 3.4 *Let $\mathcal{C}_i$ be a cell overlapping a ridge, but not its endpoints. Then $\mathcal{C}_i$ is a wedge.*

Proof. Let a and b be the points on $\partial\mathcal{C}_i$ which are farthest along the ridge in either direction. A ridge can only intersect descent paths that start from it, so a and b must each lie on a vertical cut, ℓ_a and ℓ_b. Right after both vertical cuts ℓ_a and ℓ_b have been made (at different times), $\overline{ab}$ lies in the interior of the wedge $\mathcal{C}$ corresponding to $\overline{ab}$. Other vertical cuts may have been made before arriving at the current subdivision $\mathcal{K}(\mathcal{P})$ that contains $\mathcal{C}_i$. However, no vertical cut can be made between a and b, otherwise a and b could not be in the same cell. So there is no vertical cut in the interior of $\mathcal{C}$, and thus no descent path has been traced inside $\mathcal{C}$. It follows that $\mathcal{C}_i$ is the wedge $\mathcal{C}$. ◻

3.2 Data structure

During the course of the algorithm, we maintain the polygon $\mathcal{P}$ and its subdivision $\mathcal{K}(\mathcal{P})$ in a doubly-connected edge list [4]. So each cell $\mathcal{C}_i$ is represented by a circular list of edges, or several if it has holes. In the following, we show how we augment these chains so that they record incidences between the boundary of $\mathcal{C}_i$ and the faces of $\mathcal{S}'$.

For each cell $\mathcal{C}_i$, let $\mathcal{S}'_i$ be the subdivision of $\mathcal{C}_i$ induced by $\mathcal{S}'$. (So the two-dimensional faces of $\mathcal{S}'_i$ are the non-empty intersections of two-dimensional faces of $\mathcal{S}'$ with $\mathcal{C}_i$.) Let Q denote a circular list of edges that form one component of $\partial\mathcal{C}_i$. We subdivide each vertical edge of Q at each intersection point with an edge of $\mathcal{S}'$. Now each edge e of Q bounds exactly one face f_j of $\mathcal{S}'_i$. We store a pointer from e to the slab σ_j corresponding to f_j. In addition, for each vertex of Q which is a reflex vertex of $\mathcal{P}$, we store pointers to the two corresponding motorcycle slabs. We call this data structure a *face list*. So we store one face list for each connected component of $\partial\mathcal{C}_i$. (See Figure 11.)

Lemma 3.5 makes an observation that will be used in subsequent lemmas.

Lemma 3.5 *A hole of a cell is necessarily a hole of $\mathcal{P}$.*

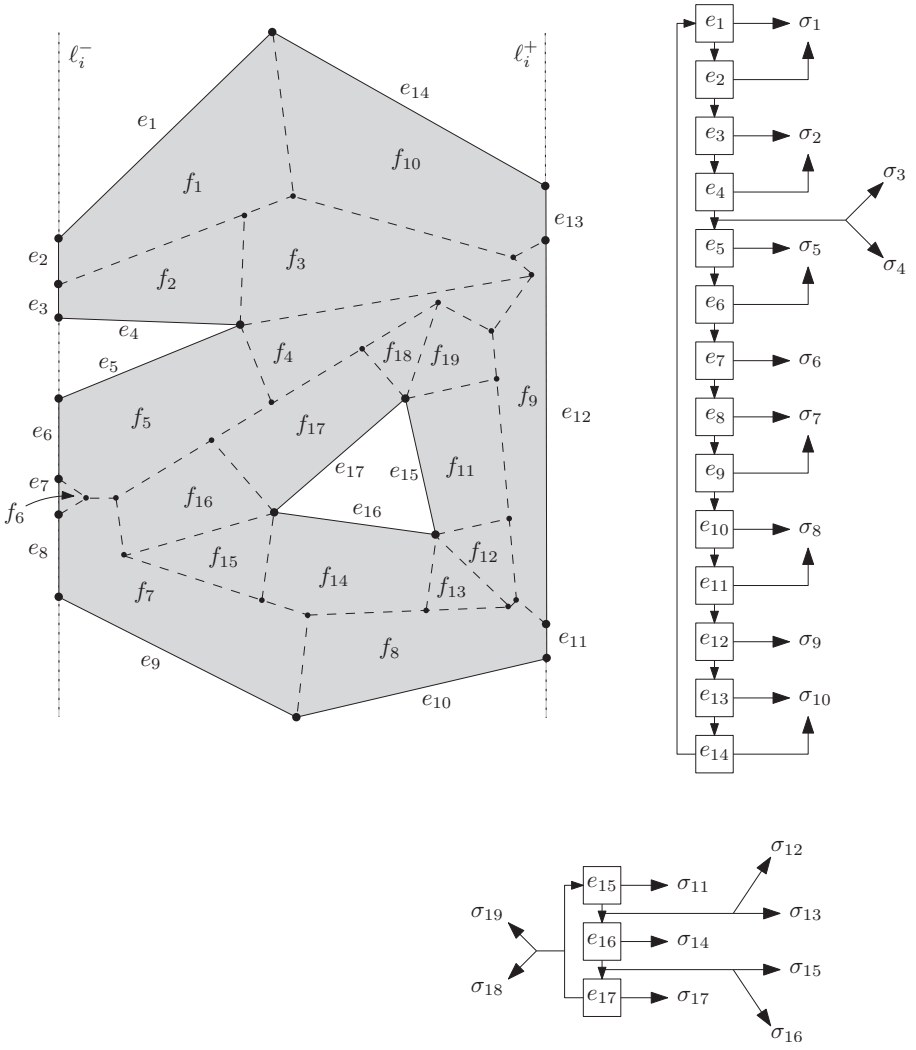


Figure 11: The face lists for the cell $\mathcal{C}_i$ bounded by the vertical line cuts ℓ_i^- and ℓ_i^+. The faces are denoted by $f_1, \ldots, f_{19}$ and the corresponding slabs are $\sigma_1, \ldots, \sigma_{19}$. The face lists point to these slabs, as the exact shape of the faces of $\mathcal{S}'$ is not known.

Proof. When we subdivide a cell $\mathcal{C}_i$, each newly added edge either connects directly to $\partial\mathcal{C}_i$, or connects to $\partial\mathcal{C}_i$ via a descent path. Therefore, every new boundary edge created in subdividing $\mathcal{C}_i$ is connected to $\partial\mathcal{C}_i$. If there was a hole, it would imply that there is a sequence of new boundary edges that is not connected to $\partial\mathcal{C}_i$, an impossibility. It follows that no new holes are created by our algorithm. The initial cell $\mathcal{C}_1$ contains holes which are precisely the holes of $\mathcal{P}$. □

We say that a vertex v of the motorcycle graph $\mathcal{G}$ *conflicts* with a cell $\mathcal{C}_i$ of $\mathcal{K}(\mathcal{P})$ if either v lies in the interior of $\mathcal{C}_i$, or v is a reflex vertex of $\partial\mathcal{C}_i$. We also store the list of all the vertices conflicting with each cell $\mathcal{C}_i$. This list V_i is called the *vertex conflict list* of $\mathcal{C}_i$. In summary, our data structure consists of a doubly-connected edge list storing $\mathcal{K}(\mathcal{P})$, and the face lists and the vertex conflict list V_i of each cell $\mathcal{C}_i$.

We say that an edge e of $\mathcal{S}'$ conflicts with the cell $\mathcal{C}_i$ if it intersects the interior of $\mathcal{C}_i$. We denote by c_i the number of edges of $\mathcal{S}'$ conflicting with $\mathcal{C}_i$. During the course of the algorithm, we do not necessarily know all the edges conflicting with a cell $\mathcal{C}_i$, and we don't even know c_i, but this quantity will be useful for analyzing the running time. In particular, it allows us to bound the size of the data structure for $\mathcal{C}_i$.

Lemma 3.6 *If $\mathcal{C}_i$ is non-empty, then the total size of the face lists of $\mathcal{C}_i$ is $O(c_i)$. In particular, it implies that $\partial\mathcal{C}_i$ has $O(c_i)$ edges, and $\mathcal{C}_i$ overlaps $O(c_i)$ faces of $\mathcal{S}'$. On the other hand, if $\mathcal{C}_i$ is empty, then the total size is $O(1)$, and thus $\partial\mathcal{C}_i$ has $O(1)$ edges.*

Proof. Let Q denote the outer boundary of $\mathcal{C}_i$, and let $|Q|$ denote its number of edges. By Lemma 3.1, each reflex vertex p of Q is in a valley, and the two edges of Q incident to p bound the two faces of $\mathcal{S}_i$ incident to this valley. So any subchain Q' of Q that bounds only one face f of $\mathcal{S}'_i$ must be convex. The edges of Q' can take only 3 directions: vertical, parallel to the base edge of f, or the steepest descent direction. So Q' can have at most 5 edges: two vertical edges, two edges parallel to the steepest descent direction, and one edge along the base edge of f.

Thus, Q can be partitioned into at least $|Q|/5$ subchains, such that two consecutive subchains bound different faces of $\mathcal{S}'_i$. Any vertex of Q at which two consecutive subchains meet must be incident to an edge e of $\mathcal{S}'_i$ that conflicts with $\mathcal{C}_i$. By Lemma 3.2 and 3.3, this edge can meet $\partial\mathcal{C}_i$ at most twice. So in total, Q has at most $10(c_i + 1)$ edges.

Now consider the holes of $\mathcal{C}_i$, if any. Such a hole must be a hole of $\mathcal{P}$ according to Lemma 3.5, so each vertex along its boundary is the endpoint of at least one edge that conflicts with $\mathcal{C}_i$. Each conflicting edge is adjacent to at most one hole vertex, so there are $O(c_i)$ such vertices in $\mathcal{C}_i$. In addition, each edge of a hole bounds only one face, and for each reflex vertex, another two faces corresponding to motorcycle slabs are added. So in total, the face lists for holes have size $O(c_i)$.

We just proved that the total size of the face lists is $O(c_i + 1)$. If c_i is non-empty, we have $c_i \geq 1$, and thus the bound can be written $O(c_i)$. Otherwise, if $\mathcal{C}_i$ is empty, then it does not conflict with any edge, so $c_i = 0$. Hence, the data structure has size $O(1)$. □

3.3 Algorithm

Our algorithm partitions $\mathcal{P}$ recursively, using vertical cuts, as in Sect. 3.1. In this section, we show how to perform a step of this subdivision in near-linear time. A cell $\mathcal{C}_i$ is subdivided along a vertical cut through its median conflicting vertex, so the vertex conflict lists of the new cells will be at most half the size of the conflict lists of $\mathcal{C}_i$. When the vertex conflict list of $\mathcal{C}_i$ is

empty, we call the procedure DIVIDE-VALLEY presented in Section 4. If $\mathcal{C}_i$ is empty[1] or is a wedge, then we stop subdividing $\mathcal{C}_i$, and it becomes a *leaf cell*.

We now describe in more details how we perform a vertical cut efficiently. We assume that the cell $\mathcal{C}_i$ conflicts with at least one vertex, and that $\mathcal{C}_i$ is given with the corresponding data structure as described in Sect. 3.2. We first find the conflicting vertex with median x-coordinate in $O(|V_i|)$ time. We compute the list of vertical boundary segments $s_1, \ldots, s_q$ created by the cut along the vertical line ℓ through the median vertex. This list is sorted along ℓ, and it can be constructed in time proportional to the number of edges bounding $\mathcal{C}_i$, which is $O(c_i)$ by Lemma 3.6.

Each segment s_i can be lifted vertically to a polyline $\hat{s}_i$ on $\mathcal{T}$. We compute $\hat{s}_1, \ldots, \hat{s}_q$ as follows. Let H denote the vertical plane through ℓ. We first find the list of slabs corresponding to the faces of $\mathcal{S}'_i$. We obtain this list as the union of the slabs that appear in the face lists of $\mathcal{C}_i$. We compute the intersection of each such slab with H. This gives us a set of $O(c_i)$ segments in H, of which we compute the lower envelope. It can be done in $O(c_i \log c_i)$ time [19]. Then we obtain $\hat{s}_1, \ldots, \hat{s}_q$ by scanning through this lower envelope and the list $s_1, \ldots, s_q$. Overall it takes time $O(c_i \log c_i)$ to compute this lower envelope, and it has $O(c_i)$ edges, as each edge of $\mathcal{S}'_i$ or $\mathcal{C}_i$ creates at most one vertex along this chain.

The partition induced by ℓ is obtained by tracing steepest descent paths from $s_1, \ldots, s_q$. For a vertical edge s_j, any vertex of $\hat{s}_j$, when projected onto the horizontal plane, corresponds precisely to a point where s_j intersects an edge e of $\mathcal{S}'_i$. At each of these points, we do the following without actually knowing $\mathcal{S}'_i$. There are at most three steepest descent paths from $a = \hat{e} \cap \hat{s}_j$, one for each slab through a. Each such descent path consists of one line segment along the slab, followed possibly by another line segment along a valley in the case where the slab is a motorcycle slab. Let γ denote one of these descent paths. As we know the slab and the starting point of γ, we can construct γ in constant time. This path γ goes all the way to $\partial\mathcal{P}$, so if necessary, we clip it at ℓ or $\partial\mathcal{C}_i$.

These descent paths cannot cross, and by construction they do not cross the vertical boundary edges. Each edge of $\mathcal{S}'_i$ may create at most three such descent paths, so we create $O(c_i)$ such new descent paths. There are also $O(c_i)$ new vertical edges, so we can update the doubly-connected edge list in time $O(c_i \log c_i)$ by plane sweep. Using an additional $O(|V_i| \log c_i)$ time, we can update the vertex conflict lists during this plane sweep. The face lists can be updated in overall $O(c_i)$ time by splitting the face lists of $\mathcal{C}_i$ along the lower endpoints of the new descent paths, and inserting new subchains along each vertical edge s_j, which we obtain directly from $\hat{s}_j$ in linear time. So we just proved the following:

Lemma 3.7 *We can compute the subdivision of a non-empty cell $\mathcal{C}_i$ induced by a line through its median conflicting vertex, and update our data structure accordingly, in $O((c_i + |V_i|) \log c_i)$ time.*

3.4 Analysis

In the previous section, we saw that the vertical subdivision of each cell $\mathcal{C}_i$ can be obtained in time near-linear in the size of the data structure for $\mathcal{C}_i$. We now bound the overall running time of the algorithm, so we need to bound the sum $\sum_i c_i + |V_i|$ over all cells created by DIVIDE-VERTICAL.

We use the *recursion tree* associated with Algorithm 1. Each node ν of this tree represents a cell $\mathcal{C}_i$, and the child cells of $\mathcal{C}_i$ are stored at the descendants of ν in the recursion tree. In particular, the cells stored at the descendants of ν form a partition of the cell stored at ν. Each time we subdivide a cell $\mathcal{C}_i$, the vertex conflict list of each new cell has at most half the size of

[1] Recall that a cell is empty if its interior does not intersect $\mathcal{S}'$.

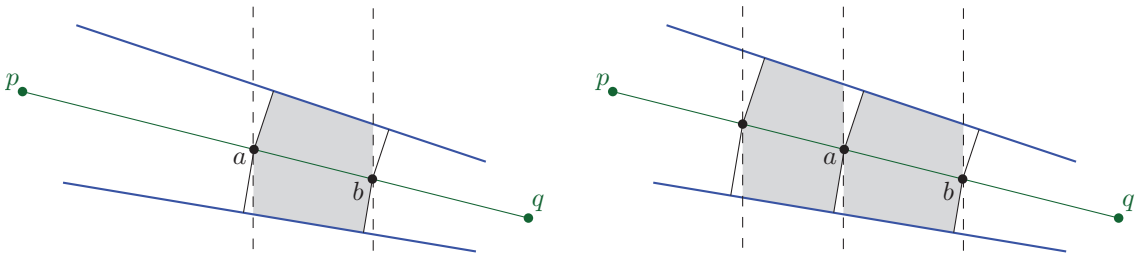


Figure 12: A first wedge is created (left), and an adjacent wedges is created afterwards (right). The cell containing p has been split simultaneously. Blue edges belong to ∂P.

the vertex conflict list of $\mathcal{C}_i$. As there are at most $2r$ vertices in $\mathcal{G}$, it follows that:

Lemma 3.8 *The recursion tree of* Divide-Vertical *has depth* $O(\log r)$.

The degree of any vertex in $\mathcal{K}(\mathcal{P})$ is at most 5, because there can be at most three descent paths through any point, as well as two vertical edges. It implies that any point of $\mathcal{P}$ is contained in at most 5 cells at each level of the recursion tree. It follows that:

Lemma 3.9 *Any point in* $\mathcal{P}$ *is contained in* $O(\log r)$ *cells of* $\mathcal{K}(\mathcal{P})$ *throughout the algorithm.*

In particular, if we apply this result to each of the $2r$ vertices of $\mathcal{G}$, we obtain:

Lemma 3.10 *Throughout the algorithm, the sum* $\sum_i |V_i|$ *of the sizes of the vertex conflict lists is* $O(r \log r)$.

We now bound the total number of conflicts between edges of $\mathcal{S}'$ and cells of $\mathcal{K}(\mathcal{P})$.

Lemma 3.11 *Throughout the algorithm, each edge* e *of* $\mathcal{S}'$ *conflicts with* $O(\log r)$ *cells. It follows that* $\sum_i c_i = O(n \log r)$.

Proof. Let p, q denote the endpoints of e. First we assume that e is a ridge. By Lemma 3.9, there are at most $O(\log r)$ cells containing p or q, so it remains to bound the number of cells that overlap e but not $\{p, q\}$. By Lemma 3.4, these must be wedges. A wedge overlapping with e is created only if at least two vertical cuts through e have been made. When the second such cut is made, the wedge associated with a segment $\overline{ab} \subset e$ is created. Assume without loss of generality that a is between p and b. Any wedge is a leaf cell. So the wedge associated with $\overline{ab}$ will not be split by vertical cuts again. In order to create a new wedge along e, one must cut with a vertical line through $\overline{pa}$ or $\overline{bq}$. (See Figure 12.) It creates a new wedge adjacent to the first one, and it splits the cell containing p or q, creating a new cell containing p or q. Repeating this process, we can see that for each new wedge created along e, a new cell containing p or q is created. So there can be only $O(\log r)$ wedges along e.

If e is a valley or a flat edge, then by Lemma 3.2, it only conflicts with cells that contain its higher endpoint, so throughout the algorithm, there are $O(\log r)$ such cells by Lemma 3.9. □

We can now state the main result of this section. Its proof follows from Lemmas 3.6, 3.7, Lemma 3.10, and 3.11.

Lemma 3.12 *The vertical subdivision procedure completes in* $O(n(\log n) \log r)$ *time. The cells of the resulting subdivision are either empty cells, wedges, or do not contain any motorcycle vertex in their interior. They are simply connected, and the only reflex vertices on their boundaries lie on valleys.*

Proof. The vertical subdivision procedure subdivides every cell recursively unless it is an empty cell or a wedge, or it does not contain any motorcycle graph vertex. So the final subdivision consists of these three kinds of cells only. When we perform a subdivision, we can identify in constant time each empty child cell, because by Lemma 3.6, these cells have constant size. When we find such a cell, we do not recurse on it, so these cells do not affect the running time of our algorithm. Therefore, by Lemma 3.7, the running time of Algorithm 1 is the $O(\sum_i (c_i + |V_i|) \log c_i)$ over all cells created during the course of the algorithm. By Lemmas 3.10 and 3.11, this quantity is $O(n(\log n) \log r)$.

Lemma 3.1 implies the only reflex vertices on the boundary of a cell lie on valleys.

We prove by contradiction that the cells are simply connected. Suppose that at the end of the vertical subdivision, a cell $\mathcal{C}_i$ has a hole. This hole must be a hole of $\mathcal{P}$ according to Lemma 3.5, hence it has a reflex vertex which conflicts with $\mathcal{C}_i$. As the vertex conflict list of $\mathcal{C}_i$ is non-empty, it must be an empty cell or a wedge, in which case it cannot contain a hole of $\mathcal{P}$. □

4 Cutting between valleys

4.1 Algorithm

In this section, we describe the second stage of the algorithm. The corresponding procedure is called Divide-Valley, and its pseudocode is supplied in Algorithm 2. Let $\mathcal{C}_i$ be a cell of $\mathcal{K}(\mathcal{P})$ constructed by Divide-Vertical on which we call Divide-Valley. This cell $\mathcal{C}_i$ is not empty and is not a wedge, as these are handled via brute force by Divide-Vertical, so by Lemma 3.12 it does not contain any motorcycle graph vertex in its interior. Let R_i denote the set of valleys that conflict with $\mathcal{C}_i$. We call R_i the *valley conflict list*. The *extended valley* e' corresponding to a valley $e \in R_i$ is the segment obtained by extending e until it meets the boundary $\partial\mathcal{C}_i$ of the cell. By Lemma 3.2, the valley e must meet $\partial\mathcal{C}_i$, so we only need to extend it in one direction so as to obtain e'. As $\mathcal{C}_i$ does not contain any motorcycle graph vertex in its interior, it implies that the extended valleys of $\mathcal{C}_i$ do not cross. By Lemma 3.12, the cell $\mathcal{C}_i$ is simply connected, so the extended valleys along with $\partial\mathcal{C}_i$ form an outerplanar graph with outer face $\partial\mathcal{C}_i$. (See Figure 13.)

ALGORITHM 2: Divide-Valley

Input: Cell $\mathcal{C}_i$
Output: Skeleton $\mathcal{S}'_i$
if *no valley conflicts with* $\mathcal{C}_i$ **then**
    Compute $\mathcal{S}' \cap \mathcal{C}_i$ as a lower envelope of planes;
    return
end
Build the list of all valleys conflicting with $\mathcal{C}_i$;
Construct a balanced cut s as in Lemma 4.1;
Construct the vertical slab H through s;
Construct $\hat{s}$ as the lower envelope of the slabs intersecting H;
Trace within $\mathcal{C}_i$ the two or three steepest descent paths from each vertex of $\hat{s}$;
Update the partition $\mathcal{K}(\mathcal{P})$ using s and the descent paths as new boundaries;
for *each child cell* $\mathcal{C}_j$ *of* $\mathcal{C}_i$ **do**
    Construct the data-structure for $\mathcal{C}_j$;
    Call Divide-Valley($\mathcal{C}_j$);
end

At this stage of the algorithm, the cells are simply connected, so we record each cell $\mathcal{C}_i$ using a single face list. We do not need vertex conflict lists, as the cells do not conflict with any

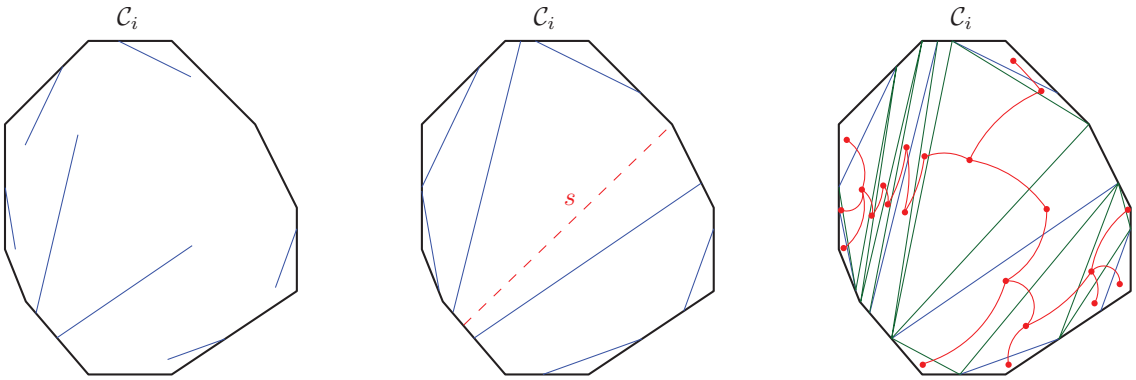


Figure 13: (Left) The cell $\mathcal{C}_i$ and the conflicting valleys. (Middle) The extended valleys, and a balanced cut. (Right) The triangulation and its dual graph.

motorcycle graph vertex. We do not need to store the valley conflict list R_i either, as we can obtain it in linear time from the face list.

If $\mathcal{C}_i$ conflicts with at least one valley, we first construct a *balanced cut*, which is a chord s of $\partial\mathcal{C}_i$ such that there are at most $2|R_i|/3$ extended valleys on each side of s. (See Figure 13, middle.) The existence of s and the procedure to identify s are explained below, in Lemma 4.1, but we first describe the rest of the algorithm. This balanced cut plays exactly the same role as the vertical edges along the cutting line that were used in Divide-Vertical. So we insert s as a new boundary segment, we compute its lifted version $\hat{s}$, and at each crossing between s and $\mathcal{S}'$, inserts the descent paths as new boundary edges.

We repeat this process recursively, and we stop recursing whenever a cell does not conflict with any valley. All the structural results in Section 3 still hold, except that now a cell is sandwiched between two balanced cuts, which can have arbitrary orientation.

So now we assume that we reach a leaf $\mathcal{C}_i$, which does not conflict with any valley. By Lemma 3.1, this cell $\mathcal{C}_i$ must be convex. As valleys are the only non-convex edges of $\mathcal{T}$, its restriction $\hat{\mathcal{C}}_i$ above $\mathcal{C}_i$ is convex. Hence, it is the lower envelope of the supporting planes of its faces. These faces are obtained in $O(c_i)$ time from the face lists, and the lower envelope can be computed in $O(c_i \log c_i)$ time algorithm using any optimal 3D convex hull algorithm. [2] We project $\hat{\mathcal{C}}_i$ onto the xy-plane and we obtain the restriction $\mathcal{S}'_i$ of $\mathcal{S}'$ to $\mathcal{C}_i$.

4.2 Analysis

It remains to analyze this algorithm, and prove the existence of a balanced cut.

Lemma 4.1 *Given a simply connected cell $\mathcal{C}_i$ that does not conflict with any motorcycle vertex, and that conflicts with at least one valley, and given the face list of $\mathcal{C}_i$, we can compute a balanced cut of $\mathcal{C}_i$ in time $O(c_i \log c_i)$.*

Proof. By Lemma 3.6, the cell $\mathcal{C}_i$ has $O(c_i)$ edges. We obtain the list R_i of valleys conflicting with $\mathcal{C}_i$ in $O(c_i)$ time by traversing the face list. Let $e_1, \ldots, e_q$ denote these valleys. We first compute the set of extended valleys $R'_i = \{e'_1, \ldots, e'_q\}$. The set R'_i can be obtained in $O(c_i)$ time by traversing $\partial\mathcal{C}_i$ as follows: We start at an arbitrary vertex of $\mathcal{C}_i$, and each time we encounter an endpoint of a valley, we push the valley into a stack. At each edge u of $\mathcal{C}_i$ that we traverse,

[2] Although it would not improve the overall time bound of our algorithm, we can even compute $\hat{\mathcal{C}}_i$ in $O(c_i)$ time using a linear-time algorithm for the medial axis of a convex polygon [1]: First construct the polygon on the xy-plane that is bounded by the traces of the supporting planes of the faces of $\hat{\mathcal{C}}_i$, then compute its medial axis, and construct its intersection with $\mathcal{C}_i$.

we check whether the extended valley e'_j at the top of the stack meets it, and if so, we draw e'_j, we pop it out of the stack, and we check whether the new edge at the top of the stack meets u.

Now we consider the outerplanar graph obtained by inserting the chords of R'_i along $\partial\mathcal{C}_i$. (See Figure 13, middle.) We triangulate this graph, which can be done in $O(c_i)$ time using Chazelle's linear-time triangulation algorithm [7], or in $O(c_i \log c_i)$ time using simpler algorithms [4]. We construct the dual of this triangulation. We subdivide any edge of the dual corresponding to an extended valley, and we assign weight one to the new node. The other nodes have weight zero. This graph is a tree, with degree at most 3, so we can compute a weighted centroid ω in time $O(c_i)$ [22]. If the centroid is an edge of the tree, then its removal yields connected components each of weight at most $|R_i|/2$. If the centroid is a node of the tree, then its removal yields connected components each of weight at most $2|R_i|/3$. If ω corresponds to an extended valley e'_j, we pick $s = e'_j$ as the balanced cut. Otherwise, ω corresponds to a face of the triangulation. We cut along the edge s of this triangular face corresponding to the subtree with largest weight. □

Lemma 4.1 plays the same role as Lemma 3.7 in the analysis of Divide-Vertical. At each level of recursion, the size of the largest valley conflict list R_i is multiplied by at most $2/3$, so the recursion depth is still $O(\log r)$. A leaf cell $\mathcal{C}_i$ is handled in $O(c_i \log c_i)$ time by computing a lower envelope of planes, as explained above. It follows that we can complete the second step of the subdivision, and compute $\mathcal{S}'$ within each cell, in overall $O(n(\log n)\log r)$ time. Then Theorem 1.1 follows.

Our analysis of this algorithm is tight, as shown by the example in Section 6.

5 Degenerate cases

As discussed in Section 2, the description and analysis of our algorithm was given for polygons in general position. Here we briefly explain why our result generalizes to arbitrary polygons.

Almost all degeneracies can be treated by standard perturbation techniques, replacing high degree nodes with several nodes of degree 3 [17]. The only difficult case is when two or more valleys meet, and generate a new valley. In the induced motorcycle graph, this situation is represented by two or more motorcycles colliding, and generating a new motorcycle. (See Figure 14.) Huber and Held gave the definition of motorcycle graphs in degenerate cases [21].

Let the movement of motorcycle m be defined by two moving edges during the shrinking process. We call the edge to the left of the track the *left arm* of m, and the other edge the *right arm*. From each reflex vertex we launch a motorcycle as described in Section 2, so the two reflex edges form the two arms.

Suppose two or more motorcycles crash simultaneously at a point p, and a new motorcycle is created. Denote by $m_1, ..., m_k$ the motorcycles that crashed at p such that (i) their traces appear counter-clockwise around p and (ii) the traces of m_1 and m_k bound the convex slice of a local disc D around p. Then we start at p a motorcycle m which inherits the left arm of m_1 and the right arm of m_k. This tells us the speed and direction of the new motorcycle. (See Figure 14c.)

So in degenerate cases, we assume that the exact induced motorcycle graph has been computed. It can be done in time $O(r^{17/11+\varepsilon})$ for any $\varepsilon > 0$ [17]. Then the problem becomes one of computing a lower envelope of slabs. For each motorcycle, we create two slabs in the same way as in the non-degenerate case: For each arm of a motorcycle vertex, the corresponding slab makes an angle $\pi/4$ with the horizontal, and is bounded by rays orthogonal to the arm and going through the endpoints of the motorcycle edge. (See Figure 14d.)

The only difference with the non-degenerate case is that now, instead of having each valley

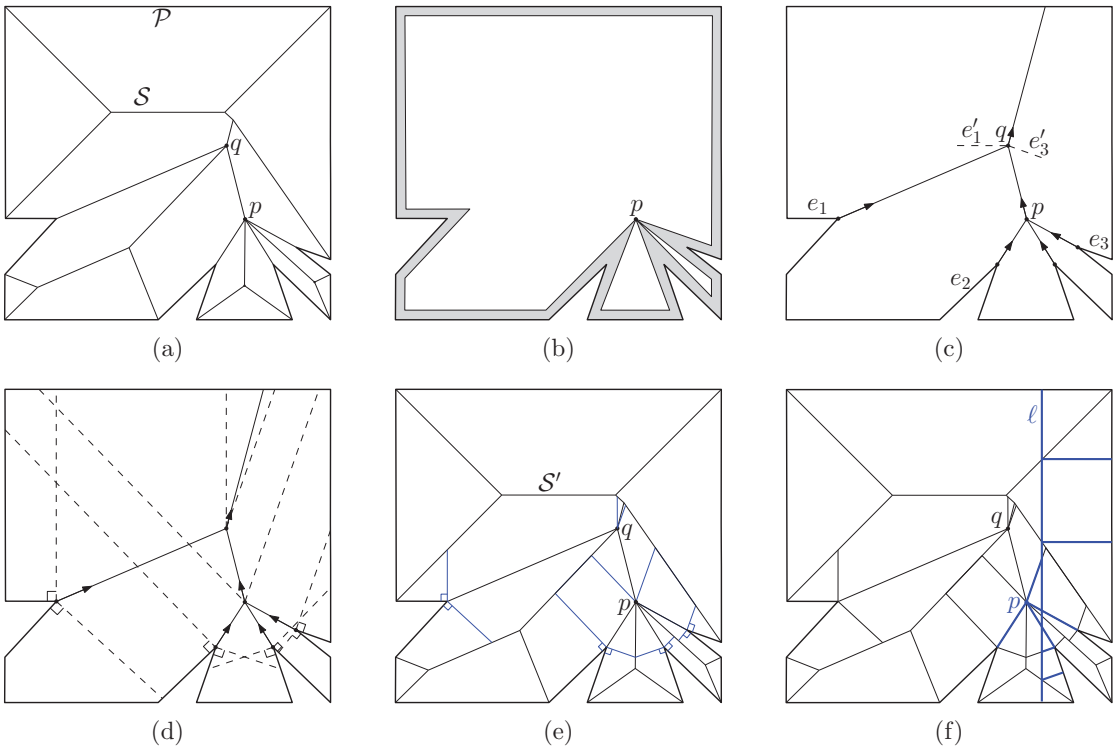


Figure 14: (a) The straight skeleton $\mathcal{S}$ of a degenerate polygon $\mathcal{P}$. Two valleys meet at q, and three meet at p. (b) Three reflex vertices collide at p during the shrinking process. (c) The motorcycle graph associated with this polygon. The left and right arms at p are e_2 and e_3. At q, the left and right arms are e_1 and e_3, and the velocity of the new motorcycle at q is given by the translates e_1' and e_3' of e_1 and e_3. (d) The slabs are bounded by the dashed segments. (e) The skeleton $\mathcal{S}'$. (f) The partition of $\mathcal{S}'$ induced by ℓ (bold). The whole subtree rooted at p is in the new boundary.

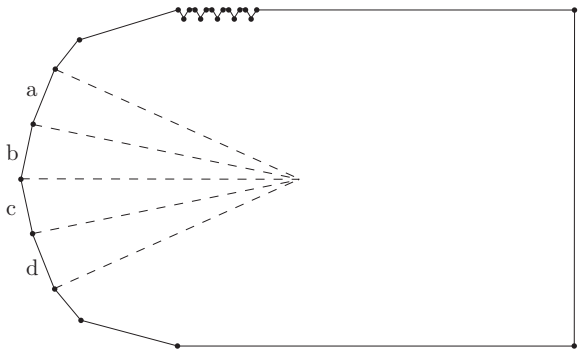


Figure 15: Tight example. For vertical cuts that are introduced from left to right, the four slabs corresponding to e_1, e_2, e_3, e_4 conflict with the cuts.

adjacent to a reflex vertex, the valleys form a forest, with leaves at the reflex vertices. So a descent path may be a polyline with arbitrarily many vertices. In Section 2 we explained how we obtain $\mathcal{S}'$ from $\mathcal{S}$. Now, we extend this definition for the degenerate case. Let v be a straight skeleton vertex at which two or more motorcycles collide and form a new motorcycle with track t. Let e_1 and e_2 be the left and right arms of t respectively. At v we draw a flat edge in the direction perpendicular to e_1. (See Figure 14e.) This flat edge goes into the adjacent face of $\mathcal{S}$ until it hits another edge of $\mathcal{S}$. A second flat edge is drawn in the opposite face in a similar way, orthogonally to e_2. Repeating this for all motorcycle start vertices of $\mathcal{G}$ gives us $\mathcal{S}'$.

In Section 3.1 we stated that there are at most three steepest descent paths to $\partial\mathcal{C}$ from a point $\hat{p}$ on $\mathcal{T}$ above a cell $\mathcal{C}$. In the degenerate case, we can only say that there are $O(r)$ paths of steepest descent from such a point $\hat{p}$. When we perform a vertical cut, we cut along all the possible descent paths. (See Figure 14f.) We can not necessarily trace a descent path in constant time. However, we can trace it in time proportional to its size, and its edges become cell boundaries. The subdivision can be updated in amortized $O(\log n)$ time for each such edge, as we update the partition by plane sweep. So the extra contribution to the overall running time is $O(n \log n)$.

6 Tightness of analysis

We give an example to demonstrate that for this algorithm the analysis is tight. A similar example is used by Huber and Held to show that a triangulation-based straight skeleton algorithm may take $\Theta(n^2 \log n)$ time [21]. Consider a polygon $\mathcal{P}$ where, on the left hand side, we have a convex chain of $\Omega(n)$ near-vertical edges. Along the top boundary of $\mathcal{P}$ we have $\Omega(r)$ small reflex dips pointing downwards. See Figure 15 for an example with a convex chain of size 4, and 5 reflex dips. The straight skeleton faces corresponding to each edge of the convex chain to the left of the polygon extend deep into the polygon. Each time we make a vertical cut to the right of all other vertical cuts previously made, it will cross through all faces of the chain, hence all the slabs must be provided to the lower envelope calculation. It then follows that Algorithm 1 spends $\Omega(n(\log n) \log r)$ time as it computes $\Omega(\log r)$ lower envelopes of $\Omega(n)$ line segments.

Acknowledgment

We thank the anonymous referees for their helpful comments and suggestions.

References

[1] A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor. A linear-time algorithm for computing the voronoi diagram of a convex polygon. *Discrete and Computational Geometry*, 4(1):591–604, 1989.

[2] O. Aichholzer, D. Alberts, F. Aurenhammer, and B. Gärtner. A novel type of skeleton for polygons. *Journal of Universal Computer Science*, 1(12):752–761, 1995.

[3] G. Barequet, M. Goodrich, A. Levi-Steiner, and D. Steiner. Straight-skeleton based contour interpolation. *Proceedings of the 14th annual ACM-SIAM symposium on Discrete algorithms*, pages 119–127, 2003.

[4] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.

[5] T. Biedl, M. Held, S. Huber, D. Kaaser, and P. Palfrader. Weighted straight skeletons in the plane. *Computational Geometry: Theory and Applications*, 48:120–133, 2015.

[6] J. Bowers. Computing the straight skeleton of a simple polygon from its motorcycle graph in deterministic O(n log n) time. *CoRR*, abs/1405.6260, 2014.

[7] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6(5):485–524, August 1991.

[8] S.-W. Cheng, L. Mencel, and A. Vigneron. A faster algorithm for computing straight skeletons. In *Proceedings of the 16th European Symposium on Algorithms*, ESA '14, pages 272–283, 2014.

[9] S.-W. Cheng and A. Vigneron. Motorcycle graphs and straight skeletons. *Algorithmica*, 47(2):159–182, 2007.

[10] F. Chin, J. Snoeyink, and C. A. Wang. Finding the medial axis of a simple polygon in linear time. *Discrete and Computational Geometry*, 21(3):405–420, 1999.

[11] F. Cloppet, J. Oliva, and G. Stamon. Angular bisector network, a simplified generalized voronoi diagram: Application to processing complex intersections in biomedical images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):120–128, 2000.

[12] S. Coquillart, J. Oliva, and M. Perrin. 3d reconstruction of complex polyhedral shapes from contours using a simplified generalized voronoi diagram. *Computer Graphics Forum*, 15(3):397–408, 1996.

[13] A. Day and R. Laycock. Automatically generating large urban environments based on the footprint data of buildings. *Proceedings of the 8th ACM symposium on Solid Modeling and Applications*, pages 346–351, 2003.

[14] E. D. Demaine, M. L. Demaine, and A. Lubiw. Folding and cutting paper. *Revised Papers from the Japan Conference on Discrete and Computational Geometry*, pages 104–117, 1998.

[15] E. D. Demaine, M. L. Demaine, and A. Lubiw. Folding and one straight cut suffice. *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 891–892, 1999.

[16] E. D. Demaine, M. L. Demaine, and J. S. B. Mitchell. Folding flat silhouettes and wrapping polyhedral packages: New results in computational origami. *Proceedings of the 15th Annual ACM Symposium on Computational Geometry*, pages 105–114, 1999.

[17] D. Eppstein and J. Erickson. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. *Discrete and Computational Geometry*, 22(4):569–592, 1999.

[18] P. Felkel and Š. Obdržálek. Straight skeleton implementation. *Proceedings of the 14th Spring Conference on Computer Graphics*, pages 210–218, 1998.

[19] J. Hershberger. Finding the upper envelope of n line segments in O(n log n) time. *Information Processing Letters*, 33(4):169–174, 1989.

[20] S. Huber and M. Held. Theoretical and practical results on straight skeletons of planar straight-line graphs. *Proceedings of the 27th Symposium on Computational Geometry*, pages 171–178, 2011.

[21] S. Huber and M. Held. A fast straight-skeleton algorithm based on generalized motorcycle graphs. *International Journal of Computational Geometry and Applications*, 22(5):471–498, 2012.

[22] O. Kariv and S. Hakimi. An algorithmic approach to network location problems. II: The p-medians. *SIAM Journal on Applied Mathematics*, 37(3):539–560, 1979.

[23] T. Kelly and P. Wonka. Interactive architectural modeling with procedural extrusions. *ACM Transactions on Graphics*, 30(2):14:1–14:15, 2011.

[24] A. Vigneron and L. Yan. A faster algorithm for computing motorcycle graphs. *Proceedings of the 29th Symposium on Computational Geometry*, pages 17–26, 2013.

[25] G. von Peschka. *Kotirte Ebenen: Kotirte Projektionen und deren Anwendung; Vorträge.* Brno: Buschak and Irrgang, 1877.