An Optimal-Time Algorithm for Shortest Paths on a Convex Polytope in Three Dimensions [*]

Yevgeny Schreiber
School of Computer Science
Tel Aviv University
Tel Aviv 69978, Israel
syevgeny@tau.ac.il

Micha Sharir
School of Computer Science
Tel Aviv University, Tel Aviv 69978, Israel
and
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012, USA
michas@tau.ac.il

ABSTRACT

We present an optimal-time algorithm for computing (an implicit representation of) the shortest-path map from a fixed source s on the surface of a convex polytope P in three dimensions. Our algorithm runs in $O(n \log n)$ time and requires $O(n \log n)$ space, where n is the number of edges of P. The algorithm is based on the $O(n \log n)$ algorithm of Hershberger and Suri for shortest paths in the plane [11], and similarly follows the continuous Dijkstra paradigm, which propagates a "wavefront" from s along ∂P. This is effected by generalizing the concept of conforming subdivision of the free space used in [11], and adapting it for the case of a convex polytope in $\mathbb{R}^3$, allowing the algorithm to accomplish the propagation in discrete steps, between the "transparent" edges of the subdivision. The algorithm constructs a dynamic version of Mount's data structure [16] that implicitly encodes the shortest paths from s to all other points of the surface. This structure allows us to answer single-source shortest-path queries, where the length of the path, as well as its combinatorial type, can be reported in $O(\log n)$ time; the actual path π can be reported in additional $O(k)$ time, where k is the number of polytope edges crossed by π.

The algorithm generalizes to the case of m source points to yield an implicit representation of the geodesic Voronoi diagram of m sites on the surface of P, in time $O((n+m)\log(n+m))$, so that the site closest to a query point can be reported in time $O(\log(n+m))$.

[*]Work on this paper was supported by NSF Grants CCR-00-98246 and CCF-05-14079, by a grant from the U.S.-Israeli Binational Science Foundation, by grant 155/05 from the Israel Science Fund, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University. The paper is based on the Ph.D. Thesis of the first author, supervised by the second author. A full version of this paper is available at [21].

Categories and Subject Descriptors: F.2.2 [Nonnumerical Algorithms and Problems]: Geometrical problems and computations

General Terms: Algorithms, Theory

Keywords: Shortest Path, Shortest Path Map, Polytope Surface, Unfolding, Continuous Dijkstra, Euclidean Distance

1. INTRODUCTION

Background. The problem of determining the Euclidean shortest path on the surface of a convex polytope P in $\mathbb{R}^3$ between two points is a classical problem in geometric optimization, which is motivated by many applications. This problem is a special case of the following basic general problem: Given a collection of obstacles (of known shapes and locations), find a Euclidean shortest obstacle-avoiding path between two given points, or, more generally, compute a compact representation of all such paths that emanate from a fixed source point. See the survey of Mitchell [13] for many variants and extensions; here we mention only the results that are most relevant to our specific problem.

Its first study in computational geometry is by Sharir and Schorr [23]. Their algorithm runs in $O(n^3 \log n)$ time, where n is the number of vertices of P. The algorithm constructs a planar layout of the *shortest path map*, and then the shortest path from the fixed source point s to any given query point q can be computed in $O(k+\log n)$ time, where k is the number of edges of the polytope that are crossed by the shortest path from s to q. Soon afterwards, Mount [15] gave an improved algorithm for convex polytopes with running time $O(n^2 \log n)$. Moreover, in [16], Mount has shown that the problem of storing shortest path information can be treated separately from the problem of computing it, presenting a data structure of $O(n \log n)$ space that supports $O(\log n)$-time shortest-path queries. However, the question whether this data structure can be constructed in subquadratic time, has been left open.

For a general, possibly nonconvex polyhedral surface, Mitchell et al. [14] presented an $O(n^2 \log n)$ algorithm for the single source shortest path problem (improving an earlier solution in [19]), extending the technique of Mount [15]. All algorithms in [14, 15, 23] use the same general approach, called "continuous Dijkstra", first formalized in [14]. The technique keeps track of all the points on the surface whose shortest path distance to the source s has the same value t, and maintains this "wavefront" as t increases. The same general approach is also used in our algorithm.

Chen and Han [4] use a rather different approach (for a not necessarily convex polyhedral surface). Their algorithm builds a shortest path sequence tree, using an observation that they call "one angle one split" to bound the number of branches, maintaining only $O(n)$ nodes in the tree in $O(n^2)$ total running time. The algorithm of [4] also constructs a planar layout of the shortest path map (which is "dual" to the layout of [23]), which can be used similarly for answering shortest path queries in $O(k + \log n)$ time. (Their algorithm is somewhat simpler for the case of a convex polytope, relying on the property, established by Aronov and O'Rourke [3], that this layout does not overlap itself.) In [5], Chen and Han follow the general idea of Mount [16] to solve the problem of storing shortest path information separately, for a general, possibly nonconvex polyhedral surface. They obtain a tradeoff between query time complexity $O(d \log n/ \log d)$ and space complexity $O(n \log n/ \log d)$, where d is an adjustable parameter. Again, the question whether this data structure can be constructed in subquadratic time, has been left open.

The problem has been more or less "stuck" after Chen and Han's paper, and the quadratic-time barrier seemed very difficult to break. For this and other reasons, several works [1, 2, 7, 8, 24] have presented approximate algorithms for the 3-dimensional shortest path problem. Nevertheless, the major problem of obtaining a subquadratic, or even near-linear, exact algorithm has remained open.

Seven years ago, in 1999, Kapoor [12] has announced such an algorithm for the shortest path problem on an arbitrary polyhedral surface P (see also a review in O'Rourke's column [17]). The algorithm follows the continuous Dijkstra paradigm, and claims to compute a shortest path from the source s to a *single target point* t in $O(n \log^2 n)$ time (so it does not preprocess the surface for answering shortest path queries). However, as far as we know, the details of Kapoor's algorithm have not yet been published, more than seven years after its sketchy conference publication, which makes it impossible to ascertain the correctness and the time complexity of the algorithm. Moreover, as it is presented, there seem to be quite a few difficulties that remain to be solved in Kapoor's approach. We list a few of these difficulties in the full version of our paper [21]. As it is presented, we feel that the algorithm of Kapoor [12] has many issues to address and to fill in before it can be judged at all.

The algorithm of Hershberger and Suri for polygonal domains. A dramatic breakthrough on a loosely related problem has taken place in 1995, when Hershberger and Suri [10] (see also [9, 11]) have obtained an $O(n \log n)$-time algorithm for computing the shortest path map *in the plane* in the presence of polygonal obstacles (where n is the number of obstacle vertices). Shortest path queries can then be processed in $O(\log n)$ time.

Since our algorithm uses (adapted variants of) many of the ingredients of Hershberger and Suri's algorithm, we provide a brief overview of their technique. The algorithm of [11] uses the continuous Dijkstra method — that is, propagation of the wavefront amid the obstacles, where each wave is generated by an obstacle vertex already covered by the wavefront. During the wavefront propagation, critical events that change the wavefront topology are processed: wavefront-wavefront collisions, wavefront-obstacle collisions, and wave elimination in a single wavefront.

The key new ingredient in Hershberger and Suri's algorithm, which makes the wavefront propagation efficient, is a quad-tree-style subdivision of the plane, of size $O(n)$, on the vertices of the obstacles (temporarily ignoring the obstacle edges). Each cell of this *conforming subdivision* is bounded by $O(1)$ straight line edges (called *transparent edges*), contains at most one obstacle vertex, and satisfies the following crucial property: for any transparent edge e of the subdivision, there are only $O(1)$ cells within distance $2\,|e|$ of e. Then the obstacle edges are inserted into the subdivision, while maintaining both the linear size of the subdivision and its conforming property — except that now a transparent edge e has the property that there are $O(1)$ cells within *shortest path distance* $2\,|e|$ of e. These transparent edges form the elements on which the Dijkstra-style propagation is performed: at each step, the wavefront is ascertained to (completely) cover some transparent edge, and is then advanced into $O(1)$ nearby cells and edges. Since each cell has constant descriptive complexity, the wavefront propagation inside a cell can be implemented efficiently. The conforming nature of the subdivision guarantees the crucial property that each transparent edge e needs to be processed *only once*, in the sense that no path that reaches e after the time at which it is processed can be a shortest path, so the Dijkstra style of propagation works correctly for the transparent edges.

During the propagation, the algorithm collects the wavefront collision data, from which the edges and vertices of the final map can be constructed. Inside a cell, a wavefront-obstacle collision event is relatively easy to handle; however, a wavefront-wavefront collision is more complex, especially when the colliding waves are not neighbors in the wavefront. The collision of neighboring waves occurs when a wave is eliminated by its two neighbors, which is easy to detect and process. To process collisions between non-neighboring waves another idea is introduced in [11] — the *approximate (or one-sided) wavefront*.

Propagating the exact wavefront that reaches a transparent edge e appears to be inefficient; instead, the algorithm maintains two separate "approximate" wavefronts approaching e from opposite sides. Together, this pair of one-sided wavefronts carry all the information needed to compute the exact wavefront at e. A limited interaction between this pair of wavefronts at e allows the algorithm to eliminate some of the superfluous waves and (implicitly) detect all wavefront-wavefront collisions (that constitute the vertices of the true shortest path map) *in a small neighborhood of their actual location*. In other words, a superfluous wave that should have been eliminated in some cell may survive for a while, but it will travel through only $O(1)$ adjacent cells before being "caught" and destroyed, so the damage that it may have entailed till this point does not cause the asymptotic performance of the algorithm to deteriorate.

To track all the changes of the wavefront during the propagation, it is implemented as a persistent data structure that requires $O(\log n)$ space for each update, resulting in an algorithm with $O(n \log n)$ storage.

At the end of the propagation phase, all the collision information is collected, and then Voronoi diagram techniques are used to compute exactly the vertices of the shortest path map within each cell. The vertices in all the cells are then combined into a single map using standard plane sweeping and some additional tricks. Processing the resulting map for point location completes the algorithm.

An overview of our algorithm. Our algorithm follows the general outline of the technique of [11]: It constructs a conforming subdivision of ∂P and applies the continuous Dijkstra propagation technique to the resulting transparent edges. However, extending the ideas of [11] to our case is quite involved, and requires special constructs, careful implementation, and finer analysis. In particular, many additional technical steps that address the 3-dimensional nature of the problem are introduced. To aid readers familiar with [11], the structure of our paper closely follows that of [11], although almost every part, albeit reminiscent of the corresponding part of [11], is completely different in technical details.

We begin with an overview of our algorithm. As in [11], we construct a conforming subdivision of ∂P to control the wavefront propagation. We first construct an oct-tree-like *3-dimensional* axis-parallel subdivision S_{3D}, only on the vertices of ∂P. Then we intersect S_{3D} with ∂P, to obtain a *conforming surface subdivision* S. In our case, a transparent edge e may traverse many facets and edges of P, but we still want to treat it as a single simple entity. To this end, we first replace each actual intersection of a facet of S_{3D} with ∂P by the *geodesic path* on ∂P that connects its endpoints, and make those paths our transparent edges. We associate with each such edge its *edge sequence* (of the polytope edges that it crosses), which is stored in compact form and used to unfold it to a straight segment. To compute the unfolding efficiently, we preprocess ∂P into a *surface unfolding data structure*, that allows us to compute, in $O(\log n)$ time, the image of any query point $q \in \partial P$ in any unfolding formed by a contiguous sequence of polytope edges crossed by an *axis-parallel plane* that intersects the facet of q. This is a nontrivial addition to the machinery of [11]. (In contrast, in the planar case the transparent edges are simply straight segments, which are trivial to represent and to manipulate.)

Similarly to [11], we maintain a simulation timer to control the propagation of the wavefront from one transparent edge e of S to $O(1)$ transparent edges of nearby cells. Before doing so, we first consolidate the wavefronts that have already reached e, constructing a representation of the true wavefront at e at a time when e is ascertained to have been completely covered by the wavefront, but before the wavefront covers other transparent edges further from the source to which we want to propagate from e. The last transparent edges from which the contributing wavefronts were propagated to e bound the so-called *well-covering region* $R(e)$ of e, which has similar properties to those in [11]. A key difference is that in our case shortest paths "fold" over ∂P, and need to be unfolded onto some plane (on which they look like straight segments). We cannot afford to perform all these unfoldings explicitly — this would right away degrade the storage and running time to quadratic in the worst case. Instead we maintain partial unfolding transformations at the nodes of our structure, composing them on the fly (as rigid transformations of 3-space) to perform the actual unfoldings whenever needed (the same is done when unfolding the transparent edges themselves).

In order to simplify the algorithm, we do not construct the shortest-path map explicitly[1] in the sense of [11], but instead construct a collection of unfolded "quasi-shortest-path maps", so that the real shortest path to any point is encoded by at least one of the maps, and so that any point on ∂P is covered by only $O(1)$ maps. This does not affect the asymptotic complexity of the algorithm, and simplifies it a lot because it allows us to skip several steps, analogous versions of which are needed in [11].

We maintain two *one-sided wavefronts* instead of one exact wavefront at each transparent edge e. We enforce the invariant that, for any point $p \in e$, the true shortest path distance from s to p is the smaller of the two distances to p encoded in the two one-sided wavefronts. Unlike [11], we do not apply any *explicit* interaction between the one-sided wavefronts. We also ignore collision events between non-neighboring waves.

The need to unfold shortest paths onto a plane creates additional difficulties. On top of the main problem that a surface cell may intersect up to $\Omega(n)$ facets of P, it can in general be unfolded in more than one way, and such an unfolding may *overlap* itself (see [18, 26] for description of this problem).

To overcome this difficulty, we introduce a *Riemann structure*, constructed by subdividing each surface cell into $O(1)$ simple *building blocks*, whose planar unfolding (a) is unique, and (b) is a simply connected polygon bounded by $O(1)$ straight line segments (and does not overlap itself). A global unfolding is a concatenation of unfolded images of a sequence, or more generally a tree, of certain blocks. It may overlap itself, but we ignore these overlaps, treating them as different layers of a Riemann surface. Each building block appears a constant number of times in the Riemann structure, and the structure has the property that it contains the shortest paths from the source to all the points of ∂P.

In summary, each step of the wavefront propagation phase picks up a transparent edge e, constructs each of the one-sided wavefronts at e by *merging* the wavefronts that have already reached *e from a fixed side*, and propagates from e each of its two one-sided wavefronts to $O(1)$ nearby transparent edges f, following the general scheme of [11]. Each propagation that reaches f from e proceeds along a fixed sequence of building blocks that connect e to f. Thus each propagation traces paths from a fixed *homotopy class*: they can be deformed into one another, while continuing to trace the same edge and facet sequences of ∂P. We call such a propagation *topologically constrained*, and denote the resulting wavefront that reaches f as $W(e, f)$, omitting for convenience the corresponding block sequence (or homotopy class). For a fixed edge e, there are only $O(1)$ successor transparent edges f and only $O(1)$ block sequences for any of those f's.

During each propagation, we keep track of combinatorial changes that occur *within* the wavefront, as it is being propagated from some predecessor edge g to e: At each of these events, we either split a wave into two waves when it hits a vertex, or eliminate a wave when it is "overtaken" by its two neighbors. Following a modified variant of the analysis of [11], we show that the algorithm encounters a total of only $O(n)$ "events", and processes each event in $O(\log n)$ time. To achieve the latter property, we represent each wavefront by a tree structure, as in [11], which supports standard tree operations (including SPLIT and CONCATENATE), priority queue operations (to control the Dijkstra-style propagation), and, a novelty of the structure, unfolding operations (that are constantly needed to trace and manipulate shortest paths as unfolded straight segments). The collec-

[1]The map, in its folded form, has quadratic complexity in the worst case.

tion of the "unfolding fields" in the resulting data structure is actually a dynamic version of the *incidence data structure* of Mount [16] that stores the incidence information between m non-intersecting geodesic paths and n polytope edges, and supports $O(\log(n+m))$-time shortest-path queries, using $O((n+m)\log(n+m))$ space. Our data structure has similar space requirements and query-time performance; the main novelty is the optimal preprocessing time of $O((n+m)\log(n+m))$ (in Mount's technique, it can be $\Theta(nm)$). In this sense, we combine the benefits of the data structure of [11] with those of [16].

When all wavefronts have reached e, we merge them into two one-sided wavefronts at e, similarly to the corresponding procedure in [11]. This happens at some simulation time t_e, which is an upper bound on the time at which e has been completely covered by the true wavefront. The main reason for maintaining one-sided wavefronts is that merging them is easy: Two such (topologically constrained) wavefronts $W(f,e)$, $W(g,e)$ *cannot interleave along* e, and each of them "claims" a contiguous portion of e (this property is false when merging wavefronts that reach e from different sides, or that are not topologically constrained). This allows us to perform the mergings in a total of $O(n\log n)$ time.

After the wavefront propagation phase, we perform further preprocessing to facilitate efficient processing of shortest path queries. This phase is rather different from the shortest path map construction in [11], since we do not provide, nor know how to construct, an explicit representation of the shortest path map on P in $o(n^2)$ time. However, our implicit representation of the map suffices for answering any shortest path query in $O(\log n)$ time. The query "identifies" the path combinatorially. It can produce right away the length of the path, and the direction at which it leaves s to reach the query point. An explicit representation of the path takes $O(k)$ additional time, where k is the number of polytope edges crossed by the path.

The algorithm, like its predecessor [11], is quite involved and its presentation is long. The (severe) lack of space forces us to provide only few of the details in this version, omitting the proofs of all lemmas and theorems.

Preliminaries. We borrow some definitions from [14, 22, 23]. For two facets f, f' that share a common edge χ, the *unfolding* of f' onto (the plane containing) f is the rigid transformation that maps f' into the plane containing f, effected by an appropriate rotation about the line through χ, so that f and the image of f' lie on opposite sides of that line. An unfolding transformation can be represented as a single 4×4 matrix in homogeneous coordinates [20]. An *unfolding* of ∂P along an *edge sequence* $\mathcal{E} = (\chi_1, \chi_2, \ldots, \chi_k)$, with an associated *facet sequence* $\mathcal{F} = (f_0, f_1, \ldots, f_k)$, where each χ_i is the common edge of f_{i-1} and f_i, is the composition of the unfoldings along $\chi_1, \ldots, \chi_k$, which effectively aligns $f_0, \ldots, f_{k-1}$ on the plane of f_k. We denote this unfolding as $U_{\mathcal{E}}(\mathcal{F})$, and use this notation to also denote all the partial unfoldings that align the other facets with the plane of f_k.

A *geodesic path* π is a simple path along ∂P whose unfolding (i.e., the unfolding of the facet sequence that it traverses) is a straight segment. For any $a, b \in \partial P$, a *shortest (geodesic) path* between them is denoted by $\pi(a,b)$. For a fixed a, $\pi(a,b)$ is unique, except for b in a 1-dimensional polygonal *ridge set* [23]. We put $d_S(a,b) := |\pi(a,b)|$.

We consider the problem of computing shortest paths from a fixed *source* point $s \in \partial P$ to all points of ∂P. A point $z \in \partial P$ is called a *ridge* point if there exist at least two distinct shortest paths from s to z. The *shortest path map* of s, denoted $SPM(s)$, is a subdivision of ∂P into at most n connected regions, called *peels*, whose interiors are vertex-free, and contain neither ridge points nor points belonging to shortest paths from s to vertices of P, and such that for each such region Φ, there is only one shortest path $\pi(s,p) \in \Pi(s,p)$ to any $p \in \Phi$, which also satisfies $\pi(s,p) \subset \Phi$. All these paths traverse the same (maximal) edge sequence. When unfolded onto the plane of the facet containing s, these peels form a star-shaped region with respect to s. See [23], and Figure 1 for an illustration.

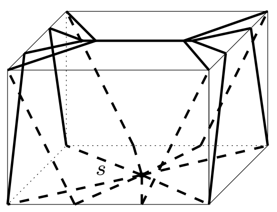

Figure 1: ***Peels are bounded by the bisectors (the set of all the ridge points), drawn as thick solid lines, and by the shortest paths from s to vertices of P, drawn dashed.***

2. CONFORMING SURFACE SUBDIVISION

We begin by constructing a special *conforming subdivision* S of ∂P, in the two following steps. First, we build in a bottom-up fashion a rectilinear oct-tree-like subdivision S_{3D} of $\mathbb{R}^3$, by considering only the set V of $O(n)$ points: The vertices of P and the source point s. We simulate a growth process of a cube box around each point of V, until their union becomes connected. The resulting S_{3D} is composed of $O(n)$ 3D-cells, each of which is either a whole axis-parallel cube or an axis-parallel cube with a single axis-parallel cube-shaped hole; the sizes of the cubes correspond to the L_∞-distances between the points of V. The boundary of each 3D-cell is divided into $O(1)$ square sub-faces with axis-parallel sides. This step is very similar to the analogous 2D construction in [11].

Next, we intersect ∂P with the sub-faces of S_{3D}. Each intersection defines (albeit does not coincide with) a *transparent edge* of S, thereby yielding an implicit representation of S. Informally, for each face h of S_{3D}, we "stretch" $h \cap \partial P$ along ∂P to obtain the corresponding transparent edge as a *geodesic path* between its endpoints; see Figure 2. To do the "stretching" efficiently, we preprocess ∂P into a *surface unfolding data structure*, as follows.

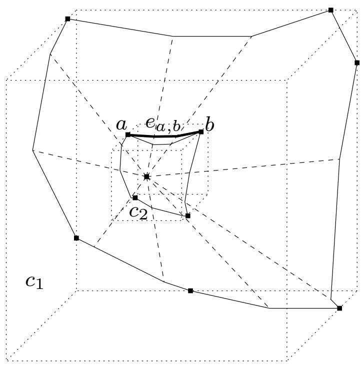

Figure 2: ***The 3D-cells c_1 and c_2 are bounded by dotted lines. The cuts of their boundaries with ∂P are drawn as thin solid lines, and the dashed lines denote polytope edges. The geodesic transparent edge $e_{a,b}$ is a thick solid line.***

The surface unfolding data structure. We sort the vertices of P in ascending z-order, and sweep a horizontal plane

ζ upwards through P. At each height z of ζ, the cross section $P(z) = \zeta \cap P$ is a convex polygon, whose vertices are intersections of some polytope edges with ζ. The cross-section remains combinatorially unchanged as long as ζ does not pass through a vertex of P. When ζ crosses a vertex v, the polytope edges incident to v and pointing downwards are deleted (as vertices) from $P(z)$, and those that leave v upwards are added to $P(z)$.

We use a persistent search tree T_z to represent the cross-sections. Since the total number of combinatorial changes in $P(z)$ is $O(n)$, the total storage required by T_z is $O(n \log n)$, and it can be constructed in $O(n \log n)$ time, using path copying. We construct, in a completely symmetric fashion, two additional persistent search trees T_x and T_y, by sweeping P with planes orthogonal to the x-axis and to the y-axis, respectively.

We can use the trees T_x, T_y, T_z to perform the following type of queries: Given an axis-parallel sub-face h of S_{3D}, compute efficiently the convex polygon $P \cap h$, and represent its boundary in compact form.

With each subtree τ of T_z at some fixed z (and similarly for T_x, T_y), we precompute and store the unfolding $U_{\mathcal{E}}$ of the polytope edge sequence $\mathcal{E}$ that consists of the leaves of τ. This allows us to compute, in $O(\log n)$ time, the image of any query point $q \in \partial P$ in any unfolding formed by a contiguous sequence of polytope edges crossed by an *axis-parallel plane* that intersects the facet of q.

Properties of S. We classify the properties of S as *structural (S)*, *conforming (C)*, and *well covering (W)*:
(S1) Each cell of S is a connected region on ∂P bounded by $O(1)$ transparent edges.
(S2) Each (transparent) edge of S is a geodesic path on ∂P, whose unfolding is a straight segment.
(S3) No pair of transparent edges cross each other.
(C1) Each cell of S contains at most one vertex of P.
(C2) Each edge of S is *well-covered* (see below).
(C3) The *well-covering region* of each edge of S contains at most one vertex of P.

An edge e is *well-covered* if the following properties hold:
(W1) There exists a set of $O(1)$ cells $C(e) \subseteq S$ such that e lies in the interior of their *connected* union $R(e) = \bigcup_{c \in C(e)} c$, which is called the *well-covering region* of e.
(W2) The total complexity of all the cells in $C(e)$ is $O(1)$.
(W3) For each transparent edge f on $\partial R(e)$, $d_S(e, f) \geq 2 \max\{|e|, |f|\}$.

THEOREM 2.1. *Every convex polytope P with n vertices in $\mathbb{R}^3$ admits a conforming surface subdivision S of $O(n)$ size, which can be implicitly constructed in $O(n \log n)$ time, and which satisfies all of the above properties.*

3. RIEMANN STRUCTURES

We represent various unfolded portions of ∂P as *Riemann structures*. Informally, this representation consists of planar "flaps", all lying in a common plane of unfolding, that are locally glued together without overlapping, but may globally have some overlaps, which however are ignored, since we consider the corresponding flaps to lie at different "layers" of the unfolding. The basic units of this structure are the *building blocks* (the "flaps"). We define the following four types of blocks in a cell c; see Figure 3.

(Type I) For each facet f of ∂P, any connected component B of $c \cap f$ that has an endpoint of some transparent edge of ∂c in its closure is a *building block of type I* of c.
(Type II) Let v be the (only) vertex in c and e a transparent edge in ∂c. Then the union B, over all facets f in a maximal sequence of adjacent facets, which are incident to v and e but neither to endpoints of e nor to another transparent edge $e' \neq e$ between e and v, of the portion of f between $e \cap f$ and v, is a *building block of type II* of c.
(Type III) Let e, e' be two distinct transparent edges in ∂c. Then the union B, over all facets f in a maximal sequence of adjacent facets, which are incident to e and e' but neither to their endpoints nor to another transparent edge e'' between e and e', of the portion of f between $e \cap f$ and $e' \cap f$, is a *building block of type III* of c.
(Type IV) Let f be a facet of ∂P. Any connected component B of the region $c \cap f$ that does not contain endpoints of any transparent edge, and whose boundary contains a portion of each of the *three* edges of f, is a *building block of type IV* of c.

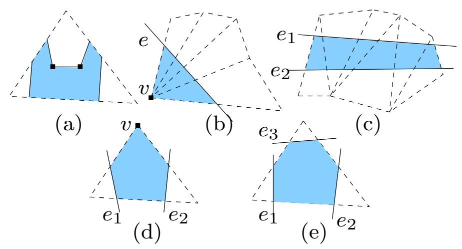

Figure 3: ***Building blocks (shaded) of types: (a) I, (b) II (unfolded), (c) III (unfolded), (d,e) IV.***

It is easy to show that the boundary complexity of each unfolded block is $O(1)$, and that the unfolding of any block of type II, III, along the polytope edge sequence that it spans, is non-overlapping (this is trivial for blocks of type I, IV).

Since there is only a constant number of elements in each surface cell that define various types of building blocks (i.e. at most one vertex of P, $O(1)$ transparent edges and their endpoints), the following lemma holds (part (ii) requires more work [21]).

LEMMA 3.1. *(i) Each surface cell c has only $O(1)$ building blocks, whose disjoint union covers c. (ii) All the building blocks of all the surface cells of S can be computed in total $O(n \log n)$ time.*

Block trees. A *contact interval* of a building block B is a maximal *open* straight segment of ∂B that lies on one polytope edge $\chi \subset \partial B$ and is not intersected by transparent edges. Contact intervals connect between building blocks within the same cell c of S. A shortest path that crosses c from one bounding transparent edge to another traverses a sequence of blocks, where consecutive blocks are separated by contact intervals that the path crosses. See Figure 4(a).

Let e be a transparent edge on the boundary of some surface cell c, and let B be a building block of c so that e appears on its boundary. The *block tree* $T_B(e)$ is a rooted tree whose nodes are building blocks of c, whose root is B, where block B'' is a child of block B' if they have a common contact interval, and where no path in $T_B(e)$ from the root contains the same block twice (except for the root B that may appear as a *leaf*; this reflects special situations where a shortest path may traverse the same block twice [21]). See Figure 4(b). Note that $T_B(e)$ has only $O(1)$ nodes.

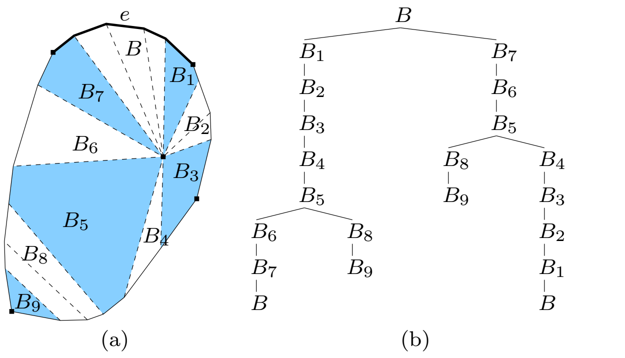

Figure 4: ***(a) A surface cell c containing a single vertex of P and bounded by four transparent edges (solid lines) is partitioned here into ten building blocks (whose shadings alternate). Contact intervals are drawn dashed. (b) The tree $T_B(e)$ of building blocks of c, where e is the (thick) transparent edge that bounds the building block B.***

We denote by $\mathcal{T}(e)$ the set of all block trees $T_B(e)$ of e (constructed from the building blocks of all cells containing e on their boundaries), and call it the *Riemann surface structure of* e; it will be used in Sections 4 and 5 for wavefront propagation block-by-block from e in all directions. This structure is indeed similar to standard Riemann surfaces (see, e.g., [25]); its main purpose is to handle effectively (i) the possibility of *overlap* between distinct portions of ∂P when unfolded onto some plane, and (ii) the possibility that shortest paths may traverse a cell c in "homotopically inequivalent" ways (e.g., by going around a vertex or a hole of c in two different ways). The use of the Riemann surface is justified by the following.

Theorem 3.2. *Let e be a transparent edge bounding a surface cell c, and let q be a point in c, such that the shortest path $\pi(s,q)$ intersects e, and the portion $\tilde{\pi}(s,q)$ of $\pi(s,q)$ between e and q is contained in c. Then $\tilde{\pi}(s,q)$ is contained in the union of building blocks that define a single path in some tree of $\mathcal{T}(e)$.*

4. THE SHORTEST PATH ALGORITHM

The algorithm uses the *continuous Dijkstra paradigm*, which simulates a unit-speed *wavefront* expanding from the given source point s, and spreading along ∂P. However, to ensure efficiency, we do not simulate the true wavefront, but an implicit representation thereof, using *one-sided wavefronts* (cf. [14] and [11]). At *simulation time* t, the true wavefront consists of points whose shortest path distance to s along ∂P is t. The wavefront is a set of closed cycles. Each cycle is a sequence of circular arcs, called *waves.*

Each wave w_i at time t (denoted also as $w_i(t)$) is the locus of endpoints of a collection $\Pi_i(t)$ of shortest paths of length t from s, all traversing (prefixes of) the same maximal polytope edge sequence $\mathcal{E}_i$. Denote by $\mathcal{F}_i$ the corresponding facet sequence of $\mathcal{E}_i$ (the facets delimited by these edges). The wave w_i is centered, in $U_{\mathcal{E}_i}(\mathcal{F}_i)$, at the source image $s_i = U_{\mathcal{E}_i}(s)$, called the *generator* of w_i. When w_i reaches, at some simulation time t, a point $p \in \partial P$, so that no other wave has reached p prior to time t, we say that s_i *claims* p, and put $claimer(p) := s_i$. We say that $\mathcal{E}_i$ is the *maximal polytope edge sequence of* s_i *at* t. For each $p \in w_i(t)$ there exists a unique shortest path $\pi(s,p) \in \Pi_i(t)$ that intersects all the edges in the corresponding prefix of $\mathcal{E}_i$, and we denote it as $\pi(s_i,p)$.

The wave w_i has (at most) two neighbors w_{i-1}, w_{i+1} in the wavefront, each of which shares a single common point with w_i. As t increases and the wavefront expands accordingly (as well as the sequences $\mathcal{E}_i$ of its waves), the meeting point of w_i with w_{i+1} traces a *bisector* of (locus of points equidistant from) the corresponding generators s_i, s_{i+1}, and is denoted by $b(s_i,s_{i+1})$; its unfolded image is a straight ray in $U_{\mathcal{E}_i}(\mathcal{F}_i)$ and in $U_{\mathcal{E}_{i+1}}(\mathcal{F}_{i+1})$. See Figure 5 for an illustration of the true wavefront.

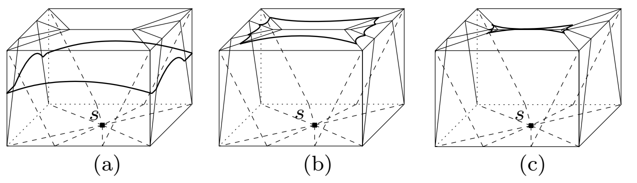

Figure 5: ***The wavefront W (drawn as a cycle of thick circular arcs) generated by s at different times t: (a) After the first four vertex events W consists of four (folded) waves. (b) After four additional vertex events, W consists of eight waves. (c) W splits into two independent cycles.***

During the wavefront simulation, the combinatorial structure of the wavefront changes at certain *critical events*, which may also change the topology of the wavefront. There are two kinds of critical events:
(i) **Bisector event**, where an existing wave is eliminated by other waves — the bisectors of all the involved generators meet at the event point. See Figure 6(a). Our algorithm detects and processes only some of the bisector events, as detailed below.
(ii) **Vertex event**, where the wavefront reaches either a vertex of P or some other boundary vertex of the Riemann structure through which we propagate the wavefront — as described in Section 5, the wave in the wavefront that reaches a vertex event splits into two new waves after the event, as depicted in Figure 6(b,c). These are the only events when a new wave is generated. Our algorithm detects and processes all vertex events.

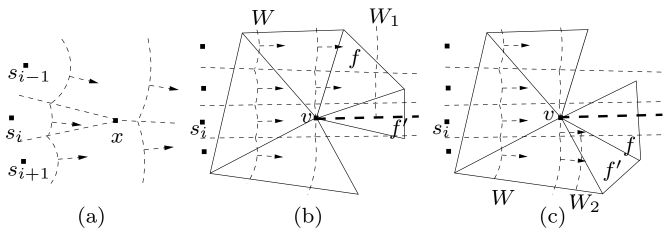

Figure 6: ***(a) The wavefront W before and after the bisector event at the point x, at which the wave of the generator s_i is eliminated from W. (b,c) Vertex event: splitting a wavefront W at a vertex v into two new wavefronts W_1, W_2, which are propagated separately beyond v through different unfoldings of the facet sequence around v — note that the images of the facets f, f' in (b) are different from those in (c).***

At each vertex of $SPM(s)$ either a vertex event, or some bisector event (either detected by our algorithm or not) takes place. Our algorithm might detect some "phony" events of this kind, but this does not affect the correctness nor the asymptotic efficiency of the algorithm.

We simulate a wavefront that may differ from the real wavefront, in that it may contain spurious waves, which, in the real wavefront, are eliminated by other waves, at bisector events that we do not detect. Each spurious wave is the locus of endpoints of *geodesic* paths that traverse the same maximal edge sequence, but they need not be shortest paths. The previous notions of bisectors, maximal polytope edge sequences, and critical events, also apply to the wavefronts propagated by our algorithm.

The propagation algorithm. The algorithm propagates *one-sided* wavefronts between transparent edges. Each such wavefront $W(e)$ is associated with some transparent edge e, and represents the true shortest paths that reach e from a fixed side, *if we ignore paths that reach* e *from the other side.* See Figure 7.

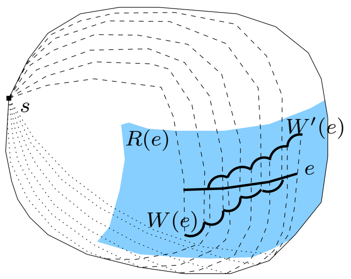

Figure 7: ***Some of the waves of the two one-sided wavefronts $W(e)$ and $W'(e)$ are absent from the true wavefront, since there is another wave in the opposite one-sided wavefront that claims the same points of e (before they do).***

Each of the one-sided wavefronts $W(e)$ at a transparent edge e is represented as a sequence of the generators (images of s) of its waves, all unfolded to a common plane, in which e becomes a straight segment, and lying on the same side from which $W(e)$ reaches e.

Following [11], the main step of our algorithm is a procedure that computes a one-sided wavefront at an edge e based on the one-sided wavefronts of "nearby" edges in a set *input*(e) of transparent edges that bound $R(e)$, the well-covering region of e. To compute a one-sided wavefront at e, we propagate the one-sided wavefronts from each $f \in$ *input*(e) which has already been processed by the algorithm, to e inside $R(e)$. Then we merge the results, separately on each side of e, to get the two one-sided wavefronts that reach e from each of its sides. The algorithm propagates the wavefronts inside $O(1)$ unfolded images of (portions of) $R(e)$, using the Riemann structure defined earlier. For each such wavefront $W(f)$, the algorithm propagates only those portions of $W(f)$ that reach e from f along straight (unfolded) segments fully contained in $R(e)$; that is, it takes into account visibility constraints that arise due to the fact that portion of $R(e)$ through which we propagate need not be convex (nor even simply connected). As mentioned, we regard each of these propagations as forming a separate "homotopy class" of paths, and refer to each such propagated wavefront as *topologically constrained.*

We denote by *output*(e) the set of direct "successor" edges to which the one-sided wavefronts of e should be passed; specifically, *output*$(e) = \{f \mid e \in$ *input*$(f)\}$. The size of (number of edges in) *input*(e), for any edge e, is constant, by construction, and the same holds for *output*(e).

The simulation clock. The simulation maintains a time parameter t, called *simulation clock*, which the algorithm strictly increases during execution and processes each edge e when t reaches the value *covertime*(e), a conservative upper bound of the real time $\max\{d_S(s,q) \mid q \in e\}$ at which e is completely run over by the true wavefront, computed on the fly for each edge e. At each step, the algorithm picks up the unprocessed edge e with minimum *covertime*(e), and sets $t :=$ *covertime*(e). It then computes the two one-sided wavefronts $W_1(e)$, $W_2(e)$ at e, by *merging* the wavefronts that have already been propagated to e. Next, for each edge $g \in$ *output*(e), it computes the time $t_{e,g}$ at which one of $W_1(e)$, $W_2(e)$ first reaches an endpoint of g, by *propagating* this wavefront from e to g, along each of the $O(1)$ possible "topologically constrained" sequences of building blocks that connect e to g. It then updates *covertime*$(g) := \min\{$*covertime*$(g), t_{e,g} + |g|\}$. Each transparent edge e is processed just once, at simulation time at which e has already been fully covered by the true wavefront, but, because of the well-covering property, before the time at which any yet unprocessed $g \in$ *output*(e) is reached at all by the wavefront (see [21]).

Merging wavefronts. A crucial property of one-sided wavefronts, from which the efficiency of the algorithm is derived, is (cf. also [11], and note that it may fail for the true "two-sided" wavefront):

LEMMA 4.1. *Let e be a transparent edge, and let $W(f,e)$ be a (topologically constrained) contributor to one of the one-sided wavefronts $W(e)$. Then the claim of $W(f,e)$ on e is* connected*—no other wavefront that reaches e from the same side can claim any point between two points claimed by $W(f,e)$.*

Assume for now that each such contributor $W(f,e)$ has already been correctly computed. That is, the correct order of the generators in each $W(f,e)$ at time $t_e =$ *covertime*(e) is known, each surviving wave does claim some points on e, and the outermost points of e that $W(f,e)$ reaches are also known.

Since the claim of each contributing wavefront $W(f,e)$ on e is connected, the merge can proceed in a linear fashion along e, eliminating waves of $W(f,e)$ that "lose" to waves from other wavefronts, processing them one by one, in the order they appear in $W(f,e)$; see Figure 8. We show:

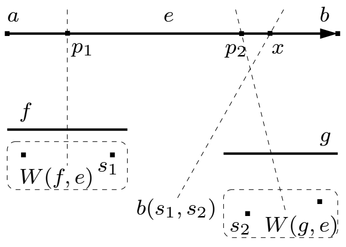

Figure 8: ***Merging $W(f,e)$ and $W(g,e)$ at e: s_2 is eliminated from $W(e)$, because its contribution to $W(e)$ must be to the left of p_2 and to the right of x, and therefore does not exist.***

LEMMA 4.2. *For each transparent edge e and for each of its two sides, we can compute the one-sided wavefront $W(e)$ that reaches e from that side at simulation clock t_e, from the collection of $O(1)$ contributing wavefronts $W(f,e)$, over all previously processed $f \in$ input(e) and connecting block sequences in $R(e)$, in $O((1+k)\log n)$ time, where k is the overall number of generators in all these wavefronts $W(f,e)$ that are absent from $W(e)$.*

The following lemma is the crucial ingredient for the correctness of the propagation algorithm, and is proved by induction on its steps.

LEMMA 4.3. *Any generator deleted during the construction of a one-sided wavefront at the transparent edge e does not contribute to the* true wavefront *at e. Every generator that contributes to the true wavefront at e belongs to one of the one-sided wavefronts at e.*

Bisector events of the first kind are detected when we simulate the advance of the wavefront $W(e)$ from e to g through some connecting sequence of building blocks, to compute the topologically constrained wavefront portion $W(e,g)$, for some $g \in output(e)$. In any such event, two non-adjacent generators s_i, s_j become adjacent due to the elimination of the intermediate wave(s); see Figure 6(a). This event is the starting point of $b(s_i, s_j)$, which reaches g in $W(e,g)$ if both waves survive the trip. Storing and maintaining these events by their "priorities" (distances from s), the algorithm processes all such events that occur before time *covertime*(g). From the properties of a topologically constrained wavefront follows that only triplets of its *neighbor* waves collide in such events.

Bisector events of the second kind occur when waves from different topologically constrained wavefronts collide. Our algorithm does not compute these events (although it implicitly "senses" that some of them have taken place, when waves are eliminated during the merging step along a transparent edge), and we ignore them in what follows.

LEMMA 4.4. *The total number of processed bisector events, over all the well-covering regions, is $O(n)$.*

5. IMPLEMENTATION DETAILS

The data structure. A one-sided wavefront is an ordered list of generators (source images), which is stored in an appropriate balanced binary tree structure, which supports:
(i) *List operations*: CONCATENATE, SPLIT, INSERT, and DELETE.
(ii) *Priority queue operations*, maintaining priorities assigned to generators, each equal to the time in which the generator is eliminated by its two neighbors.
(iii) *Source unfolding operations*, which (a) Compute explicitly any source image s_i in the wavefront at time t, or the bisector between two adjacent source images, by unfolding the appropriate maximal polytope edge sequences at t. We update the unfoldings as the wavefront advances. (b) SEARCH in the generator list for a claimer of a given query point (ignoring other wavefronts or possible visibility constraints).

The first two types of operations are similar to those in [11], and their implementation is fairly standard, using a persistent[2] red-black tree with an embedded heap structure.

The source unfolding queries are supported by adding an unfolding transformation field $U[v]$ to each node v of the binary tree, in such a way that, for any queried generator s_i, the unfolding of s_i is equal to the product (composition) $U[v_1]U[v_2]\cdots U[v_k]$ of the transformations stored in the nodes $v_1 = root, v_2, \ldots, v_k =$ leaf storing s_i, of the path from the leaf v_k storing s_i to the root.

[2]We require the data structure to be *confluently persistent* [6], since we need the ability to operate on and modify past versions of any list (wavefront), and we need the ability to merge existing distinct versions into a new version.

To perform the SEARCH operation efficiently, we precompute and store in each internal node v of the tree the *bisector image* $b[v]$, which is the bisector between the source image stored in the rightmost leaf of the left subtree of v and the one stored in the leftmost leaf of the right subtree of v, unfolded into the destination plane of $U[v]$. Given a query point q in the destination plane of $U[root]$, we determine on which side of $b[root]$ q lies, in constant time, and proceed to the left or to the right child of the root, accordingly. When we proceed from a node v to its child, we maintain the composition $U^*[v]$ of all unfolding transformations on the path from the root to v (by initializing $U^*[root] := U[root]$ and updating $U^*[w] := U^*[u]U[w]$ when processing a child w of a node u on the path). Thus, denoting by b the bisector whose corresponding image $b[v]$ is stored at v, we can determine on which side of b q lies, by computing the image $U^*[v]b[v]$, in $O(1)$ time. It thus takes $O(\log n)$ time to SEARCH for the claimer of q.

In a typical step of updating some wavefront W, we have a contiguous subsequence W' of W, which we want to advance through a new polytope edge sequence $\mathcal{E}$. We perform two SPLIT operations that split T into three subtrees T^-, T', T^+, where T' stores W', and T^- (resp., T^+) stores the portion of W that precedes (resp., succeeds) W' (either of these subtrees can be empty). Then we take the root r' of T', and replace $U[r']$ by $U_{\mathcal{E}}U[r']$ and $b[r']$ by $U_{\mathcal{E}}b[r']$. Finally, we concatenate T^-, the new T', and T^+, into a common new tree T. See [21].

Wavefront propagation. Let e be a transparent edge, and let c be a surface cell so that $e \subset \partial c$. We describe next a procedure for computing $W(e,g)$ for any transparent edge $g \subset \partial c$. Because the edges of *output*(e) belong to $O(1)$ cells in the vicinity of e, we can use this primitive repeatedly to compute $W(e,g)$ for all $g \in output(e)$, including the edges that do not belong to ∂c.

Choose a block tree $T_B(e)$ in $\mathcal{T}(e)$. Let $W = W(t)$ denote the kinetic wavefront within the blocks of $T_B(e)$ at any time t during the propagation; initially, $t = covertime(e)$ and $W = W(e)$. Denote by $\mathcal{C}$ the *boundary chain* of $T_B(e)$, which contains all the boundary segments of each block B' of $T_B(e)$ that do not connect B' to another block of $T_B(e)$ (these are either transparent edges or "dead-end" contact intervals). There are only $O(1)$ segments in $\mathcal{C}$, and each instance of a transparent edge or a contact interval in $\mathcal{C}$ can be reached only by a *single topologically constrained sub-wavefront* of W. We propagate W through the block sequences of $T_B(e)$ towards $\mathcal{C}$, updating W at the critical events that change its topology. The purpose of the propagation of W in $T_B(e)$ is the computation of the wavefront $W(e,f)$, for each transparent edge f in $\mathcal{C}$ that W reaches, which will be valid at the time when f is known to be fully covered by W.

However, it is difficult to determine *in advance* the exact set of those critical events that are *true events with respect to the propagation of W in $T_B(e)$*. Instead, we determine on the fly a larger set of *candidate* critical events, which might also contain false events, either because they lie outside $T_B(e)$, or are computed based on an incomplete information of earlier true events (at least one of which has not been detected and processed in time). A careful implementation ensures that not too many of these false events are collected.

Specifically, let x be such a *candidate bisector event* that takes place at simulation time t_x. If all the true events of W

that *have taken place before* t_x were *processed before* t_x, then x can be *foreseen* at the last critical event at which one of the bisectors involved in x was updated before time t_x, using *priorities* assigned to the source images in W. The priority of a source image s_i is the distance from s_i to the point at which the two (unfolded) bisectors defined by s_i and its neighbors intersect beyond e_B, either in B or beyond it. (In the latter case we cannot yet locate the intersection point, because it may depend on polytope edge sequences that "continue the unfolding", which are not immediately available; we overcome this problem by "tracing" the involved paths to the location of x beyond B block-by-block through $T_B(e)$ up to $\mathcal{C}$.) The priority is $+\infty$ if the bisectors do not intersect beyond e_B. If x is a true candidate event, in the sense of the paths from the involved generators is blocked by $\mathcal{C}$ before x, we DELETE s_i from W, and recompute the priorities of its neighbors in W.

False bisector events generally occur because we failed to detect an earlier *vertex event*, which eliminates or separates the generators involved in the bisector event. The algorithm eventually detects these vertex events, backtracks to them, and "restarts" the propagation from them. After $O(1)$ such restarts, all false bisector events will be eliminated.

In more detail, a *candidate vertex event* may not be foreseen, because the time t_v at which it occurs is not known in advance, nor does it appear among the priorities stored at the structure. Searching for the claimer of v ahead of time may fail, because some contenders, which would be eliminated at some in-between bisector event. Instead, we detect the vertex event at v only post factum, either when processing some later false candidate event, or when the propagation of W in $T_B(e)$ is stopped at a later simulation time $t_{stop}(W)$, when a segment f of $\mathcal{C}$ incident to v is ascertained to be fully covered by W (then we try to split out from W the sub-wavefront W' that claims f, since W' should not be propagated further). See Figure 9.

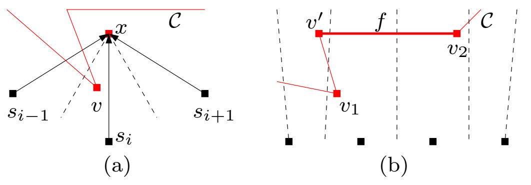

Figure 9: ***A vertex event is detected: (a) At vertex v, while processing a later false candidate bisector event x. (b) At v_1, while processing a later false vertex event at v' or v_2, when the segment f is ascertained to be fully covered by W.***

To detect a vertex event at v while we process a later candidate critical event x, we SEARCH in W for the claimer of each vertex u of $\mathcal{C}$ that "blocks" a path of W that is involved in x; we choose v at that vertex u that minimizes $d(claimer(u), u)$, and set $t_v := d(claimer(v), v)$. *All the versions of the (persistent) data structure that encode W after time t_v become invalid, since they do not reflect the update that should have occurred at t_v.* To correct this situation, we discard all the invalid versions of W, and *restart the simulation of the propagation of the last valid version of W from time t_v.* This time, however, we SPLIT W at $claimer(v)$ (at time t_v) into two new sub-wavefronts, making the ray from $claimer(v)$ to v the new extreme bisector of both. Since there are only $O(1)$ vertices in $\mathcal{C}$, these restarts do not affect the asymptotic time complexity of the propagation of W.

Most importantly, we need to know when to stop the propagation, so as not to process too many events. Some events that we encounter may occur outside the blocks in $T_B(e)$; e.g., when we reach a segment f of $\mathcal{C}$, we have to keep propagating until we can ascertain (at most $2|f|$ simulation time units later than the time when f was first reached by W) that f is fully covered by W, which may force us to process events that lie beyond f. See Figure 10. However, we only need to handle events that lie at distance at most $2|f|$ from f: If f is a transparent edge, the well-covering of f ensures that this happens at most $O(1)$ cells away from f; if f is a dead-end contact interval, the wave that reaches f does not leave the cell c (since it crosses a *cycle* of building blocks). This in turn implies that the total number of these events remains linear.

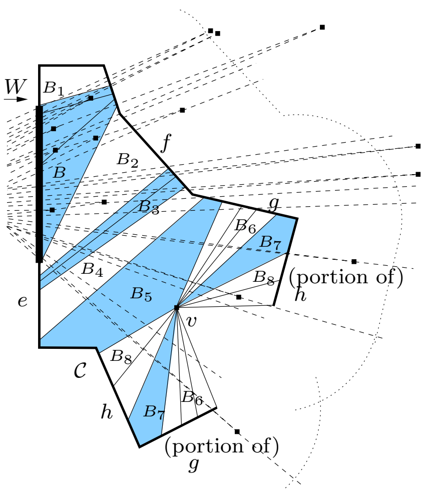

Figure 10: ***The union of all the blocks (whose shadings alternate) in $T_B(e)$ is bounded by $e \cap B$ and by $\mathcal{C}$. For each segment f of $\mathcal{C}$, we stop propagating the portion of W that has covered f after it crosses the dotted line which lies at distance $2|f|$ from f.***

We maintain a time $t_{stop}(W)$ that tells us when to stop the propagation of W. For each segment f in $\mathcal{C}$, we maintain an individual time $t_{stop}(f)$, which is a conservative upper estimate of the time when f is completely covered by W; we update $t_{stop}(f)$, whenever we trace a path that reaches f, to the minimal length of any such path, plus $|f|$. The time $t_{stop}(W)$ is the minimum of all such times $t_{stop}(f)$ in the *range* of W (the current portion of $\mathcal{C}$ that the current W is trying to reach; initially it is the whole $\mathcal{C}$). When W reaches a vertex of $\mathcal{C}$, it splits there into two sub-wavefronts W_1, W_2 (the range of W is split accordingly); we regard W as being terminated at this event, and replace it by W_1, W_2, each now maintaining its own $t_{stop}(\cdot)$ value.

The structure of the conforming subdivision, combined with the careful implementation sketched above, imply that the algorithm processes a total of only $O(n)$ candidate events, including false ones, each in $O(\log n)$ time, so the overall cost of the wavefront propagation phase is $O(n \log n)$.

Shortest path queries. Finally, we consider the stage of preprocessing the implicitly constructed shortest-path map for shortest path queries. Consider a building block B that was covered by a wavefront W (and by the wavefronts into which W has been split during its propagation in B). We partition B into *active* and *inactive* regions, and denote this partition of B by W and its descendants by $local(W, B)$. The active regions are those portions of B that are claimed by waves of W that have a (vertex or bisector) event in

B, and each inactive region is a band of waves of W that cross B in an "uneventful" manner, delimited by a sequence of pairwise disjoint bisectors; see Figure 11. The edges of $local(W, B)$ are those bisectors of generators of W, at least one of which is active in B. The first and the last bisectors of W are also defined to be edges of $local(W, B)$. The partition can actually be computed "on the fly" during the propagation of W in B, in a number of operations proportional to the number of detected critical events of W in B.

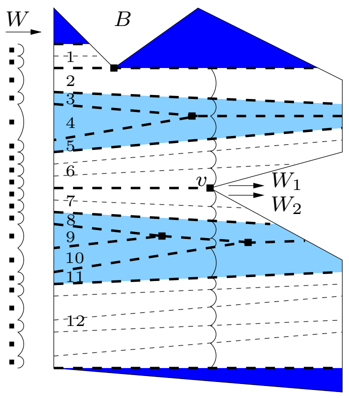

Figure 11: *The edges of $local(W, B)$ are thick dashed lines; thin dashed lines are the remaining bisectors of W. The regions of $local(W, B)$ are numbered from 1 to 12 (active regions are lightly shaded); the portions of B that were not traversed by W because of visibility constraints are darkly shaded.*

We preprocess each such $local(W, B)$ for point location, so that, given a query point $p \in B$, we can determine which region of $local(W, B)$ contains the unfolded image q of p; if this region is traversed by a single wave of W (which will always be the case for active regions, and may sometimes also hold for inactive regions), it uniquely defines the generator of W that claims p (in the absence of other wavefronts). This can be done in $O(\log n)$ time, after we have found, in $O(\log n)$ time, the building block B that contains p. If q is in an inactive region of $local(W, B)$, this region is traversed by a portion of W that was propagated through B without events; hence we can SEARCH in W for the claimer of p in $O(\log n)$ time. We repeat this procedure for each of the $O(1)$ wavefronts that traverse B, and pick up the generator that yields the shortest distance to p. We thus obtain:

THEOREM 5.1 (MAIN RESULT). *Let P be a convex polytope with n vertices. Given a source point $s \in \partial P$, we can construct an implicit representation of the shortest path map from s on ∂P in $O(n \log n)$ time and space. Using this structure, we can compute the length and initial direction of the shortest path from s to any point $t \in \partial P$ in $O(\log n)$ time. A shortest path $\pi(s, t)$ can be computed in additional time $O(k)$, where k is the number of straight edges in the path.*

6. EXTENSIONS AND REMARKS

As in the planar case of [11], our algorithm can also be easily extended to a more general instance of the shortest path problem on a convex polytope surface that involves *multiple sources*. See the abstract and [21].

Finally, we conclude with two open problems:
(i) Can the space complexity of our algorithm be reduced to linear? Can an efficient tradeoff between the query time and the space complexity be achieved, using, say, the $SPM(s)$-representations of Chen and Han [4, 5]?
(ii) Does the wavefront propagation method extend to the shortest path problem on the surface of a *nonconvex* polyhedral surface? Say, on a polyhedral terrain?

Acknowledgment. We are grateful to Haim Kaplan for his help in designing the data structures, to Joe O'Rourke for valuable comments and material on surface unfolding and overlapping, as well as for remarks on Kapoor's paper, and to Joe Mitchell for his comments of Kapoor's paper.

7. REFERENCES

[1] P. K. Agarwal, S. Har-Peled, M. Sharir, and K. R. Varadarajan, Approximate shortest paths on a convex polytope in three dimensions, *J. ACM* 44:567–584, 1997.

[2] L. Aleksandrov, A. Maheshwari, and J.-R. Sack, An improved approximation algorithm for computing geometric shortest paths, *14th FCT, Lecture Notes Comput. Sci.* 2751:246–257, 2003.

[3] B. Aronov and J. O'Rourke, Nonoverlap of the star unfolding, *Discrete Comput. Geom.*, 8:219–250, 1992.

[4] J. Chen and Y. Han, Shortest paths on a polyhedron, Part I: Computing shortest paths, *Internat. J. Comput. Geom. Appl.* 6:127–144, 1996.

[5] J. Chen and Y. Han, Shortest paths on a polyhedron, Part II: Storing shortest paths, Tech. Rept. 161-90, Comput. Sci. Dept., Univ. Kentucky, Lexington, KY, February 1990.

[6] J. R. Driscoll, D. D. Sleator, and R. E. Tarjan, Fully persistent lists with catenation, *J. ACM* 41(5):943–949, 1994.

[7] S. Har-Peled, Approximate shortest paths and geodesic diameters on convex polytopes in three dimensions, *Discrete Comput. Geom.*, 21:216–231, 1999.

[8] S. Har-Peled, Constructing approximate shortest path maps in three dimensions, *SIAM J. Comput.*, 28(4):1182–1197, 1999.

[9] J. Hershberger and S. Suri, An optimal algorithm for Euclidean shortest paths in the plane, in *Proc. 34th IEEE Sympos. Found. Comput. Sci.*, 508–517, 1993.

[10] J. Hershberger and S. Suri, An optimal algorithm for Euclidean shortest paths in the plane, Manuscript, Washington University, 1995.

[11] J. Hershberger and S. Suri, An optimal algorithm for Euclidean shortest paths in the plane, *SIAM J. Comput.* 28(6):2215–2256, 1999.

[12] S. Kapoor, Efficient computation of geodesic shortest paths, in *Proc. 32nd Annu. ACM Sympos. Theory Comput.*, New York, NY, USA: ACM Press, 770–779, 1999.

[13] J. S. B. Mitchell, Shortest paths and networks, in J. E. Goodman and J. ORourke, editors, *Handbook of Discrete and Computational Geometry (2nd Edition)*, chapter 27, 607–641, North-Holland, Chapman & Hall/CRC, Boca Raton, FL, 2004.

[14] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou, The discrete geodesic problem, *SIAM J. Comput.* 16:647–668, 1987.

[15] D. M. Mount, On finding shortest paths on convex polyhedra, Tech. Rept., Computer Sience Dept., Univ. Maryland, College Park, October 1984.

[16] D. M. Mount, Storing the subdivision of a polyhedral surface, *Discrete Comput. Geom.*, 2:153–174, 1987.

[17] J. O'Rourke, Computational geometry column 35, *Internat. J. Comput. Geom. Appl.*, 9:513–515, 1999; also in *SIGACT News*, 30(2):31–32, (1999) Issue 111.

[18] J. O'Rourke, Folding and unfolding in computational geometry, in *Lecture Notes Comput. Sci.*, Vol. 1763, J. Akiyama, M. Kano, M. Urabe, editors, Springer-Verlag, Berlin, 2000, pp. 258–266.

[19] J. O'Rourke, S. Suri, and H. Booth, Shortest path on polyhedral surfaces, Manuscript, The Johns Hopkins Univ., Baltimore, MD,1984.

[20] R. P. Paul, *Robot Manipulators: Mathematics, Programming, and Control*, MIT Press, Cambridge, Massachusetts, 1981.

[21] Y. Schreiber and M. Sharir, An optimal-time algorithm for shortest paths on a convex polytope in three dimensions, `http://www.tau.ac.il/~syevgeny/ShortestPath.ps`.

[22] M. Sharir, On shortest paths amidst convex polyhedra, *SIAM J. Comput.* 16:561–572, 1987.

[23] M. Sharir and A. Schorr, On shortest paths in polyhedral spaces, *SIAM J. Comput.* 15:193–215, 1986.

[24] K. R. Varadarajan and P.K. Agarwal, Approximating shortest paths on a nonconvex polyhedron, in *Proc. 38th Annu. IEEE Sympos. Found. Comput. Sci.*, 182–191, 1997.

[25] E. W. Weisstein, Riemann Surface, *MathWorld — A Wolfram Web Resource*, `http://mathworld.wolfram.com/RiemannSurface.html`.

[26] E. W. Weisstein, Unfolding, *MathWorld — A Wolfram Web Resource*, `http://mathworld.wolfram.com/Unfolding.html`.