Computing Geodesics on Polyhedral Surfaces
（多面体上の測地線の計算）

Kazuma Tateiri

February 2024
Division of Computer Science and Information Technology
Graduate School of Information Science and Technology
Hokkaido University

Abstract

In this paper, we consider the computation of shortest paths and geodesics, which are two types of path on the surface of a polyhedron in the three dimensional Euclidean space. Here we say that a shortest path is a path with the minimum length connecting two given points on the polyhedron, and a geodesic is a path whose length is minimal against small perturbations. Note that the shortest path between two points is a geodesic, but not all geodesics connecting those two points are the shortest path.

The shortest path problem on polyhedra is a fundamental problem in the field of computational geometry and is also important in practice. This is a generalization of the shortest path problem in graphs, which is one of the fundamental problems in the field of discrete optimization, to polyhedra. Among the problems related to shortest paths on polyhedra, the *single source shortest path problem* on polyhedra is a particularly important problem and has been well studied and there are many previous studies. In 1987, Mitchell, Mount, and Papadimitriou gave the MMP algorithm, which is the first efficient computational method for the single source shortest path problem on polyhedra, as the most relevant previous work to this paper. This was followed by the CH algorithm proposed by Chen and Han in 1990, the ICH algorithm proposed by Xin and Wang in 2009, and so on. For the *all pairs shortest path problem* (where the starting and ending points are vertices), Ying, Wang, and He have proposed the

Saddle Vertex Graph method, a graph-based framework. On the other hand, the computation of geodesics, which are locally shortest paths, has not received much attention in the field of computational geometry, and to the best of our knowledge, there seems to be little research on this topic. This is in contrast to the fact that geodesics on smooth surfaces are widely studied in mathematics, and that geodesics as local shortest paths are an important research subject in physics for applications such as mechanics and optics. Therefore, in this paper, we consider the computation of geodesics on polyhedra. In Chapter 2, after giving basic definitions of geometry and polyhedra, we define geodesics on polyhedra. In the following Chapter 3 and 4, we consider two computational problems concerning geodesics on polyhedra.

In Chapter 3, we introduce the single source geodesics enumeration problem on general polyhedra, including the case of nonconvex polyhedra, as a generalization of the enumeration problem of the shortest paths of a single source on a graph. Then, we define the basic data structure called *complete geodesic interval tree* for this problem, and discuss a method to compute it. Next, we consider reducing the number of intervals in the data structure to reduce the time and memory required to enumerate the geodesics. For this purpose, this paper proposes an improved data structure called *reduced geodesic interval tree* by removing the overlap of intervals generated around hyperbolic vertices (vertices such that the sum of angles measured around the vertex along the faces is greater than 2π). Moreover, we show that the reduced geodesic interval tree allows the result of geodesics enumeration to be succinctly encoded as a *single-pair geodesic graph*. The complexity analysis shows that the both types of geodesic interval trees can be generated in $O(N \log N)$ time and $O(N)$ space, where N is the number of intervals it contains. Furthermore, in the computer experiments, it is observed that the reduced geodesic interval tree requires less storage space than the complete geodesic interval tree and is better in terms of running time and memory consumption.

In Chapter 4, we consider the computation of *cut locus* on convex polyhedra. Assuming that a point s on the convex polyhedron $\mathcal{P}$ is specified, we can consider the isoline for any positive real number R, such that the geodesic distance from the source s is R. This is called a *wavefront*. Moreover, the set of points C for which there exist multiple shortest paths from the source s is called the cut locus. In the case of the convex polyhedron $\mathcal{P}$ considered in this chapter, the cut locus C is a graph consisting of a set of points on $\mathcal{P}$ and a line segment connecting them. In this chapter, we assume convexity of $\mathcal{P}$ and some other conditions for simplicity of discussion, and then modify the MMP algorithm by Mitchell, Mount, and Papadimitriou to construct explicit wavefront propagation and cut locus in $O(n^2 \log n)$ time and $O(n)$ space, where n is the number of vertices of the polyhedron $\mathcal{P}$. We also show that the generated intervals can be modified to support geodesic queries in the $O(n^2)$ space by keeping all the generated intervals. In our experiments with artificial data, we observed that the practical performance of the algorithm for geodesic queries is approximately $O(n^{1.5} \log n)$ time and $O(n^{1.5})$ space.

In summary, in this paper, we have considered geodesics on polyhedra in three-dimensional space and proposed efficient algorithms for the problems of enumerating single source geodesics on a nonconvex polyhedron and computing a cut locus on a convex polyhedron. We believe that these algorithms will be useful tools for various applications involving geodesics on polyhedra in the future. It is an interesting task to investigate the applicability of the proposed algorithms.

Contents

Chapter 1

Introduction

1.1 Backgrounds

In this paper, we consider the computation of shortest paths and geodesics, which are two types of path on the surface of a polyhedron in the three dimensional Euclidean space. Here we say a path on a polyhedral surface to be a curve that connects two points and passes only on the surface of the polyhedron. From now on, the entire surface of a polyhedron is simply called a polyhedron. In differential geometry, a geodesic on a smooth curved surface is defined as a locally shortest path. That is, a geodesic is a path whose length is minimal against small perturbations. The concept of geodesic can be defined similarly for polyhedrons. As a concept closely related to geodesics, the path with the minimum length connecting two given points on a polyhedron is called the shortest path on the polyhedron. Here, the shortest path between two points is a geodesic, but it must be noted that not all geodesics connecting those two points are the shortest path. The shortest path problem on polyhedra is a fundamental problem in the field of computational geometry and is also important in practice. In this problem, given a polyhedron $\mathcal{P}$ and a point s on $\mathcal{P}$ as input, it finds the shortest path from s to all vertices of $\mathcal{P}$. This is a generalization of the shortest path problem in graphs, which is one of the fundamental problems in the field of discrete optimization, to polyhedra.

Among the problems related to shortest paths on polyhedra, the *single source shortest path problem* on polyhedra is a particularly important problem and has been well studied and there are many previous studies. In 1987, Mitchell, Mount, and Papadimitriou gave the MMP algorithm [12], which is the first efficient computational method for the single source shortest path problem on polyhedra, as the most relevant previous work to this paper. This was followed by the CH algorithm [4] proposed by Chen and Han in 1990, the ICH algorithm [21] proposed by Xin and Wang in 2009, and so on. For the *all pairs shortest path problem* (where the starting and ending points are vertices), Ying, Wang, and He have proposed the Saddle Vertex Graph method [22], a graph-based framework. On the other hand, the computation of geodesics, which are locally shortest paths, has not received much attention in the field of computational geometry, and to the best of our knowledge, there seems to be little research on this topic. This is in contrast to the fact that geodesics on smooth surfaces are widely studied in mathematics, and that geodesics as local shortest paths are an important research subject in physics for applications such as mechanics and optics.

1.2 Main Results

Therefore, in this paper, we consider the computation of geodesics on polyhedra. In Chapter 2, after giving basic definitions of geometry and polyhedra, we define geodesics on polyhedra. In the following Chapter 3 and 4, we consider two computational problems concerning geodesics on polyhedra.

- In Chapter 3, we introduce the single source geodesics enumeration problem on general polyhedra, including the case of nonconvex polyhedra, as a generalization of the enumeration problem of the shortest paths of a single source on a graph. First, given (possibly nonconvex) polyhedron $\mathcal{P}$, a point $s \in \mathcal{P}$ and a positive real number R, we define our *single-source geodesics enumeration problem* to be the problem of building data structure $\mathcal{T}$ that enables to query, for any point t on $\mathcal{P}$, all geodesics from s to t whose length is less than R. Such a query is called a *geodesics enumeration query*. Then, we define the basic data structure called *complete geodesic interval tree* for this problem, and discuss a method to compute it. Next, we consider reducing the number of intervals in $\mathcal{T}$ in this problem to reduce the time and memory required to enumerate the geodesics. For this purpose, this paper proposes an improved data structure called *reduced geodesic interval tree* by removing the overlap of intervals generated around hyperbolic vertices (vertices such that the sum of angles measured around the vertex along the faces is greater than 2π). Moreover, we show that the reduced geodesic interval tree allows a query result to be succinctly encoded as a *single-pair geodesic graph*. The complexity analysis shows that the both types of geodesic interval trees can be generated in $O(N \log N)$ time and $O(N)$ space, where N is the number of intervals it contains. Furthermore, in the computer experiments, it is observed that the reduced geodesic interval tree requires less storage space than the complete geodesic interval tree and is better in terms of running time and memory consumption.

- In Chapter 4, we consider the computation of *cut locus* on convex polyhedra. Assuming that a point s on the convex polyhedron $\mathcal{P}$ is specified, we can consider the isoline for any positive real number R, such that the geodesic distance from the source s is R. This is called a *wavefront*. In general, a wavefront consists of arcs, and two arcs on a face are connected at a point on the face. As the wavefront advances with increasing R, these connection points move over the surface of the polyhedron, and the set of all these points C is called the cut locus. In other words, the cut locus is the set of points for which there exist multiple shortest paths from the source s. In the case of the convex polyhedron $\mathcal{P}$ considered in this chapter, the cut locus C is a graph consisting of a set of points on $\mathcal{P}$ and a line segment connecting them. The cut locus C is a partition of the entire surface of $\mathcal{P}$ with respect to the set of points on the surface, corresponding to a Voronoi diagram on the ordinary two-dimensional plane. However, the difference is that, unlike the case of the two-dimensional plane, a cut locus on a convex polyhedron is defined even if it has only one source. In this chapter, we assume convexity of $\mathcal{P}$ and some other conditions for simplicity of discussion, and then modify the MMP algorithm by Mitchell, Mount, and Papadimitriou to construct explicit wavefront propagation and cut locus in $O(n^2 \log n)$ time and $O(n)$ space, where n is the number of vertices of the polyhedron $\mathcal{P}$. We also show that the generated intervals can be modified to support geodesic queries in the $O(n^2)$ space by keeping all the generated intervals. In our experiments with artificial data, we observed that the practical performance of the algorithm for geodesic queries is approximately $O(n^{1.5} \log n)$ time and $O(n^{1.5})$ space.

In summary, in this paper, we have considered geodesics on polyhedra in three-dimensional space and proposed efficient algorithms for the problems of enumerating single source geodesics on a nonconvex polyhedron and computing a cut locus on a convex polyhedron. We believe that these algorithms will be useful tools for various applications involving geodesics on polyhedra in the future. It is an interesting task to investigate the applicability of the proposed algorithms.

1.3 Related work

There is a variety of research relating to geodesics on polyhedral meshes.

1.3.1 Shortest geodesics

The shortest path problem on a polyhedron has been extensively researched. On a polyhedron without boundary, a shortest path is a geodesic [12], thus the shortest path problem is equivalent to the shortest geodesic problem. Shortest geodesic algorithms can be divided into exact algorithms and approximate algorithms. Since the interest of this paper is exact algorithms, approximate algorithms are not discussed in detail here. Detailed survey of this topic is given by [3, 7].

Interval propagation algorithms for the SSSP

The *MMP algorithm* given by Mitchell, Mount and Papadimitriou [12] is an important exact algorithm for the SSSP on a polyhedron. It retains *intervals* on each edge, so that the shortest path to any point within an interval has the same combinatorial structure (faces, edges and vertices). Information required to reconstruct geodesics is appended to these intervals. However, each interval only represents geodesics passing through a particular face among the two faces incident to the edge. This can be interpreted that each interval is assigned to a directed edge and only represents geodesics through one side (in the original paper, it is the right side along the directed edge).

In the earliest stage of the MMP algorithm, intervals are created only for the edges facing s. Each interval is *propagated* using the *continuous Dijkstra* scheme. When it detects a shortest geodesic reaching a hyperbolic vertex (a vertex around which the total angle measured along the faces is greater than 2π), it generates the intervals representing the geodesics passing through that vertex. Intervals on the same directed edge are ordered, and a newly-propagated interval is inserted into the ordered list of the intervals already-existing on the directed edge. Then, it performs *trimming* among the new interval and adjacent intervals. By trimming, intervals are cut to ensure shortestness against other intervals, and intervals on a directed edge become disjoint. Intervals may become empty as a result of trimming, and such intervals are not propagated. When no non-empty intervals are newly created and the priority queue becomes empty, the algorithm terminates and outputs the shortest geodesics. Its computational complexity given by the original authors is $O(n^2 \log n)$ time and $O(n^2)$ space, where n is the number of the edges of the input polyhedron. However, according to an experiment by Surazhsky et al. [17], its practical complexity is subquadratic and could be considered as approximately $O(n^{1.5} \log n)$ time and $O(n^{1.5})$ space. They analyzed the reason behind it as that, the number of intervals per edge is approximately $O(n^{0.5})$ in practice, despite the $O(n)$ estimation by MMP.

Most of exact algorithms for the SSSP on a polyhedron, published after the MMP algorithm, contain the concept of interval propagation. The *CH algorithm* by Chen and Han [4] uses a FIFO queue instead of a priority queue. Its theoretical complexity is $O(n^2)$ time and $O(n)$ space, but its practical performance is not as good as the MMP algorithm. The *ICH* (improved CH) *algorithm* by Xin and Wang [21] introduces into the CH algorithm a priority queue as well as several new rules to exclude intervals not contributing to any shortest paths. In theory, the usage of a priority queue increases its time complexity to $O(n^2 \log n)$, but greatly improves its practical performance. While the MMP algorithm requires inserting an interval into an ordered list and solving a quadratic equation to get the intersection of the interval and certain hyperbola, the ICH algorithm does not. As a result, the ICH algorithm may outperform the MMP algorithm despite the ICH algorithm may generate more intervals than the MMP algorithm. Moreover, the ICH algorithm does not require the history of the propagated intervals to be retained. As a result, the space complexity of the ICH algorithm is $O(n)$, which is significantly smaller than the MMP algorithm.

Saddle Vertex Graph

The *Saddle Vertex Graph* (SVG) by Ying, Wang and He [22] is an approach to the vertex-to-vertex all-pairs shortest path problem on a polyhedron. They noticed that a shortest geodesic between two hyperbolic vertices may be shared by multiple longer shortest geodesics, and reduced the problem to the well-known shortest path problem on a graph by precomputing shortest geodesics connecting two vertices without passing through other vertices. However, when the given polyhedron is convex, it has no hyperbolic vertices and there is no difference from precomputing the shortest path for every pair of vertices. However, they observed real-world meshes contain 40-60% hyperbolic vertices and more than 90% of shortest paths pass through at least one hyperbolic vertex. Also, they considered the exact SVG is too large to be tractable for large meshes and proposed a method to sparsify it, and evaluated error by computer experiments.

1.3.2 General geodesics

Geodesics, which are not limited to shortest, also have been researched in the context of geodesic tracing and path shortening.

Geodesic tracing

A classical problem about a smooth surface in differential geometry is to trace the smooth geodesic from a given point p and a tangent direction v. Concerning the corresponding problem on a polyhedron, this can be done at almost every point. However, a geodesic cannot proceed beyond a spherical vertex (a vertex around which the total angle measured along the faces is less than 2π), and it cannot uniquely determine its direction when it hits a hyperbolic vertex. To cope with this problem, Polthier and Schmies suggested a *straightest geodesic* which goes through a vertex to halve the total angle of the vertex [15]. However, it is not necessarily locally-shortest anymore, and does not have continuity respect to p and v, i.e. it jumps when it moves onto or across a non-Euclidean (i.e. spherical or hyperbolic) vertex, thus its nature vastly differs from a geodesic on a smooth surface. To fill this gap, Cheng, Miao, Liu, Tu and He suggested a method to trace smooth geodesics on a tangent-continuous curved surface constructed from the input polyhedron using PN-triangles, and evaluated its accuracy by computer experiments [5].

Path shortening

In this approach, an input polyline on a polyhedron is shortened until it becomes a geodesic. The shortening process is performed according to local configuration of the path. It is implemented in e.g. [19, 13, 20] and they used it to refine a path obtained by Dijkstra's algorithm on the edge graph or other approximate shortest path algorithms on the polyhedron.

Recently, a flip-based algorithm was developed by Sharp and Crane [16]. An initial path is given as an edge sequence, and until the path becomes a geodesic, the algorithm modifies the triangulation by flipping an edge so that the new shortened path is still on the edges of the new triangulation. It works on *intrinsic triangulations* of a polyhedron — that is, the new triangulation is intrinsically embedded in the original surface, and its edges are no longer straight when it is viewed as an object in $\mathbb{R}^3$. It allows the geometry of the original surface to remain unchanged by flipping. Their method can yield not only open geodesics but also closed geodesics. Although they did not give worst-case bounds, they proved that it always obtains a geodesic by finite operations, and observed that its practical running time is on the order of milliseconds.

1.3.3 Cut-Locus

There are a few computational approaches on cut-loci on polyhedral meshes. Most of them aim to approximate a cut-locus of a smooth surface, even though they work on a polyhedral mesh. The distinction is important; cut-loci of polyhedra and smooth surfaces are quite different (for further discussion, see Chapter 5).

Thaw [9] aims to compute a cut-locus of a convex smooth surface which is approximated by a convex polyhedron. It computes an approximation of the exponential map using local unfolding of two adjacent triangles, which is similar to the MMP algorithm and other related algorithms. Since their interest is computation of a cut-locus of a convex smooth surface, they proposed an angle-based filtering method, which filters an exact cut-locus into more scarce structure similar to that of the smooth surface.

A practical approach to this problem is given by Mancinelli, Livesu and Puppo [11]. Given an approximation of the distance field from the source, they use the discrete Laplace operator to find the singularity of the distance field, as well as some topological information of the mesh to acquire the correct topology of the cut-locus. The use of the discrete Laplace operator could be considered as a key to approximate a smooth cut-locus, because the operator essentially regards the discretized surface as an approximation of a smooth surface.

Chapter 2

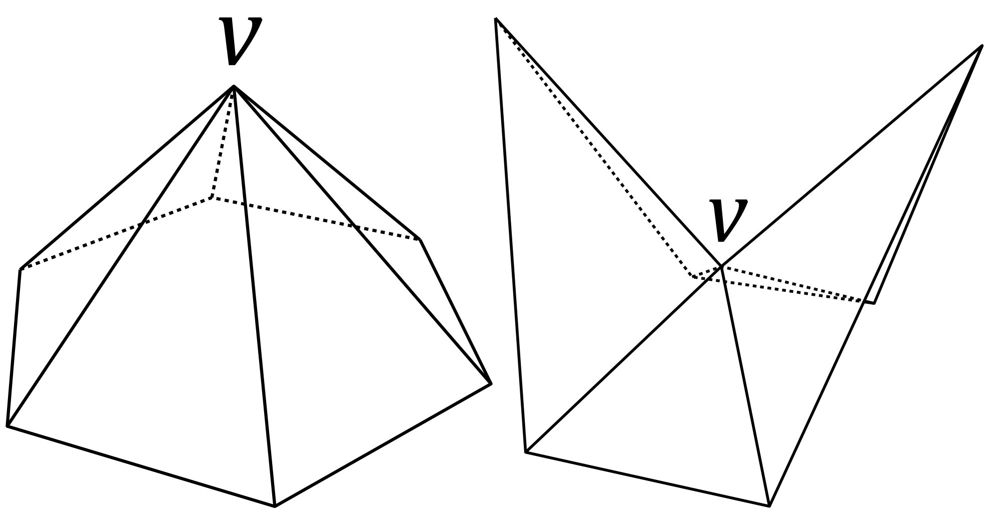
Preliminaries

2.1 Polyhedra and geodesics

Throughout this paper, we simply use the term *polyhedron* (plural *polyhedra*) to mean a (possibly non-convex) polyhedral surface, not solid polyhedron. We deal with simplicial (which means triangulated) orientable polyhedra in $\mathbb{R}^3$, and the symbol $\mathcal{P}$ is dedicated to mean such one.

Definition 2.1.1. Let v be a vertex of $\mathcal{P}$. Let τ be the sum of angles around v (measured along the faces) and we call it the *total angle* of v. Following [15], we say that

- if $\tau < 2\pi$, v is *spherical;*
- if $\tau = 2\pi$, v is *Euclidean;*
- if $\tau > 2\pi$, v is *hyperbolic.*



(a) spherical (b) hyperbolic

Figure 2.1: A spherical vertex and a hyperbolic vertex

Remark 2.1.2. If $\mathcal{P}$ is convex, every vertex is spherical or Euclidean. Note that the angle defect $2\pi - \tau$ can be interpreted as the *discrete Gaussian curvature* at v. For example, a discrete analog of the Gauss-Bonnet theorem holds for a polyhedron [6].

We define a geodesic as follows:

Definition 2.1.3. (geodesics) Let γ be a path connecting s and t on $\mathcal{P}$. We say that γ is a geodesic if and only if:

1. γ is straight inside any face, and where γ passes through an edge sequence $\mathcal{E} = (e_1, \ldots, e_k)$, γ is straight on the unfolding obtained from $\mathcal{E}$;

2. where γ passes through a vertex, the angle at the vertex made on the both sides of γ are greater than or equal to π (see Figure 2.2).

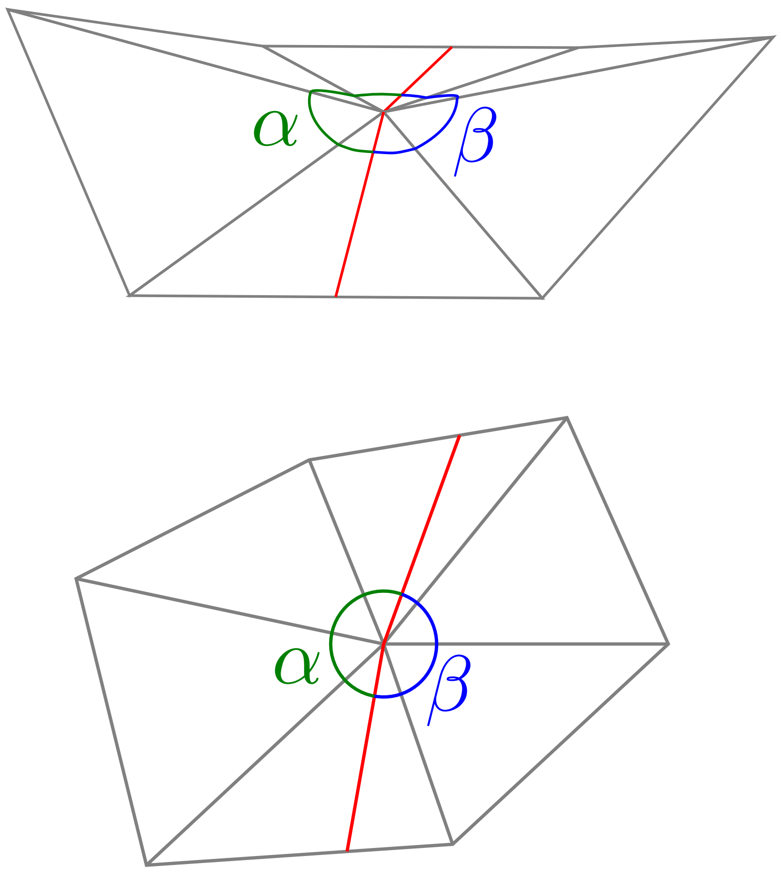


Figure 2.2: Geodesic passing through a vertex. Here $\alpha \geq \pi$ and $\beta \geq \pi$.

Upper: 3D perspective view, lower: top view

Lemma 2.1.4. A geodesic never passes through any spherical vertex.

Proof. Since the angle α and β at the vertex made on the both sides of γ satisfy $\alpha \geq \pi$ and $\beta \geq \pi$ (see Figure 2.2), the total angle of this vertex is $\tau = \alpha + \beta \geq 2\pi$. □

Remark 2.1.5. Although a geodesic does not pass through a spherical vertex, its endpoints may be spherical vertices. Moreover, a geodesic may pass through an arbitrarily close neighbor of a spherical vertex.

Remark 2.1.6. In Figure 2.2, we depicted the angles at a vertex in two different styles: in a 3D perspective view and a top view. Whenever we present a top-view style figure (like the right one in Figure 2.2), angles described in it are intended to be measured along the faces.

Intuitively, a geodesic is a locally-shortest path. Geodesics on a polyhedron, defined above, can be used to approximate geodesics on a smooth surface, defined in differential geometry. Nevertheless, there are some important differences between the discrete geodesics and the smooth geodesics. Let us take a look at some examples.

Figure 2.3 shows the shortest geodesic (red) and other geodesics (green) on a discrete ellipsoid. Since it is convex (and does not have Euclidean vertices), every vertex is spherical and geodesics pass through no vertices. Instead, there are often multiple similar geodesics that differ in how they “bypass” the vertices. As a result, when a smooth surface is discretized into a polyhedron, a single geodesic on the smooth surface often corresponds to multiple geodesics on the discretized polyhedral surface.

If the polyhedron has hyperbolic vertices, the geodesics can pass through them. Figure 2.4 shows that the shortest geodesic (red) passes through consecutive four hyperbolic vertices, as well as some of non-shortest geodesics (green) also pass through several vertices. These vertices are marked in yellow. In general, a geodesic can be decomposed into a sequence of geodesics passing through no hyperbolic vertices. We call them *primitive geodesics*.

Definition 2.1.7 (primitive geodesic). A geodesic passing through no hyperbolic vertices is called a *primitive geodesic*.

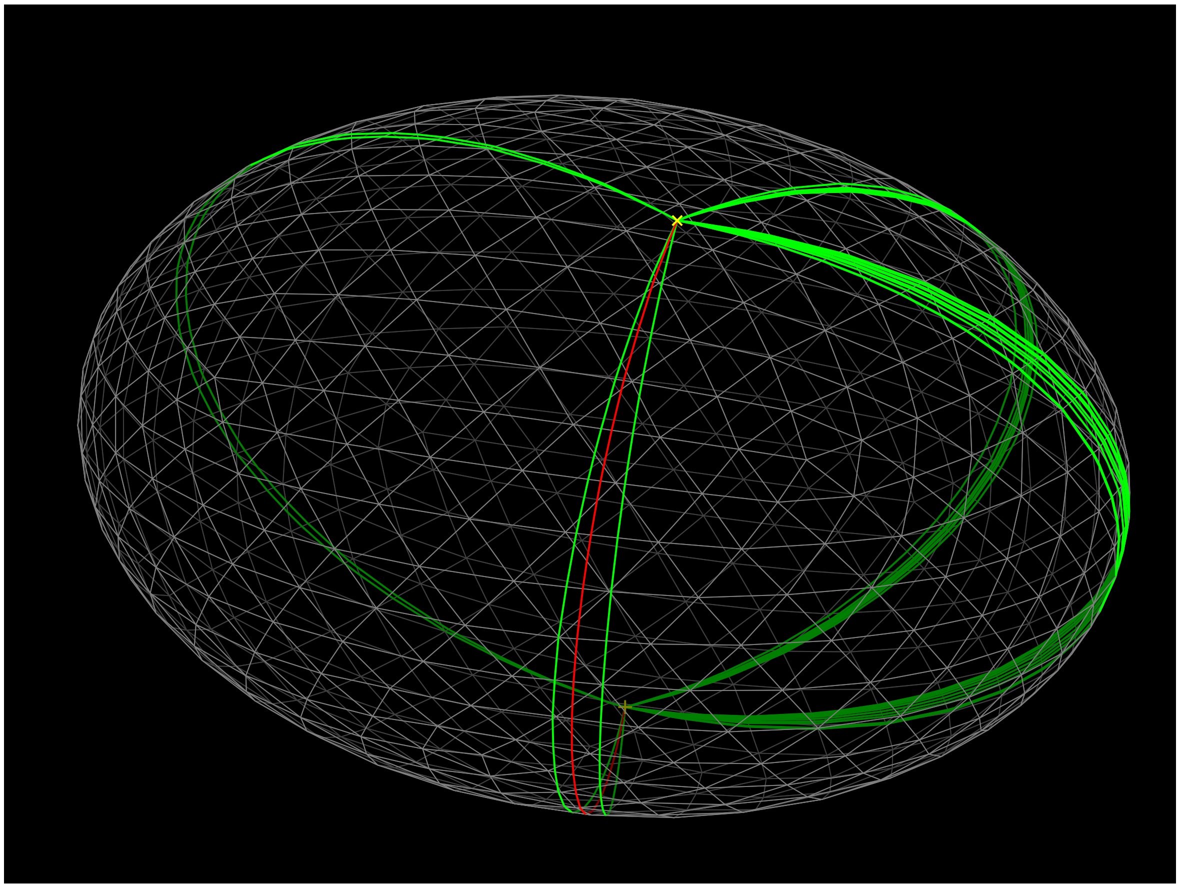


Figure 2.3: Geodesics on a discrete ellipsoid

Remark 2.1.8. Endpoints of a primitive geodesic may be hyperbolic vertices.

2.2 Geodesics and intervals

To explain how intervals can be used to express geodesics, we begin with an observation of geodesics on a polyhedron. In Figure 2.5, a geodesic g from the source s, to t on a face σ, is shown in red. Since g passes through a vertex v, it can be decomposed into the two primitive geodesics, between sv and between vt. By Definition 2.1.3 (1),

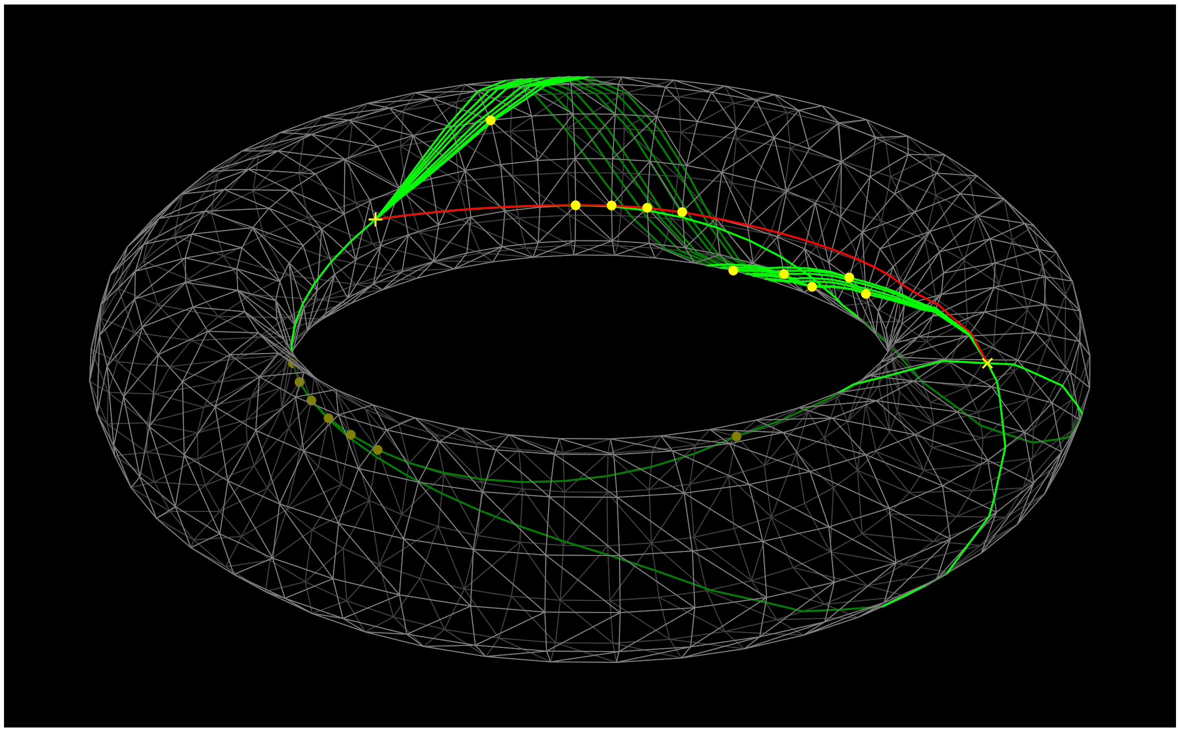


Figure 2.4: Geodesics on a discrete torus

a primitive geodesic can be unfolded into a line segment. Particularly, the primitive geodesic between vt can be unfolded into the line segment $\tilde{v}t$ on the plane containing σ (Figure 2.6).

This unfolding of the geodesic between vt is shown in 2D in Figure 2.7. As a simple observation, when one moves t in the region shaded in yellow, the line segment $\tilde{v}t$ still gives a geodesic on the unfolding. This region is chosen so that the line segment $\tilde{v}t$ does not go out of the unfolding and satisfies the condition (2) in Definition 2.1.3 at v.

When t is inside a face, one can extend the geodesic until hitting an edge, thus here we only consider geodesics to a point on an edge. We introduce concept of intervals generalizing the ones in the MMP algorithm (or other interval propagation algorithm)

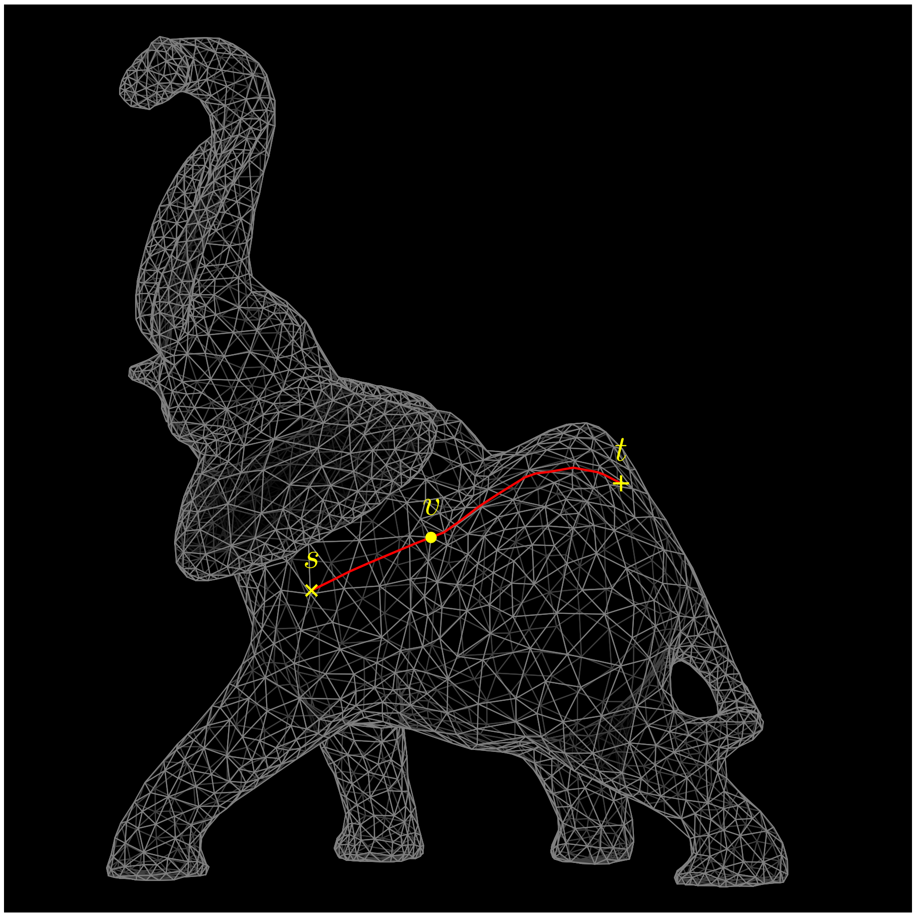


Figure 2.5: A geodesic from s to t via v

to express geodesics to a particular range on an edge. Figure 2.8 shows an outline of this expression. The yellow and green line segments represent the range of the intervals and indicates the range on which geodesics can be given in this unfolding. The region darkly shaded in their respective color indicates the region in which the interval is used in the geodesic query. As a remark, when a geodesic passes through no vertices, we consider the unfolding of the whole geodesic from s, and when it passes through multiple vertices, we consider the unfolding of the primitive geodesic between the last

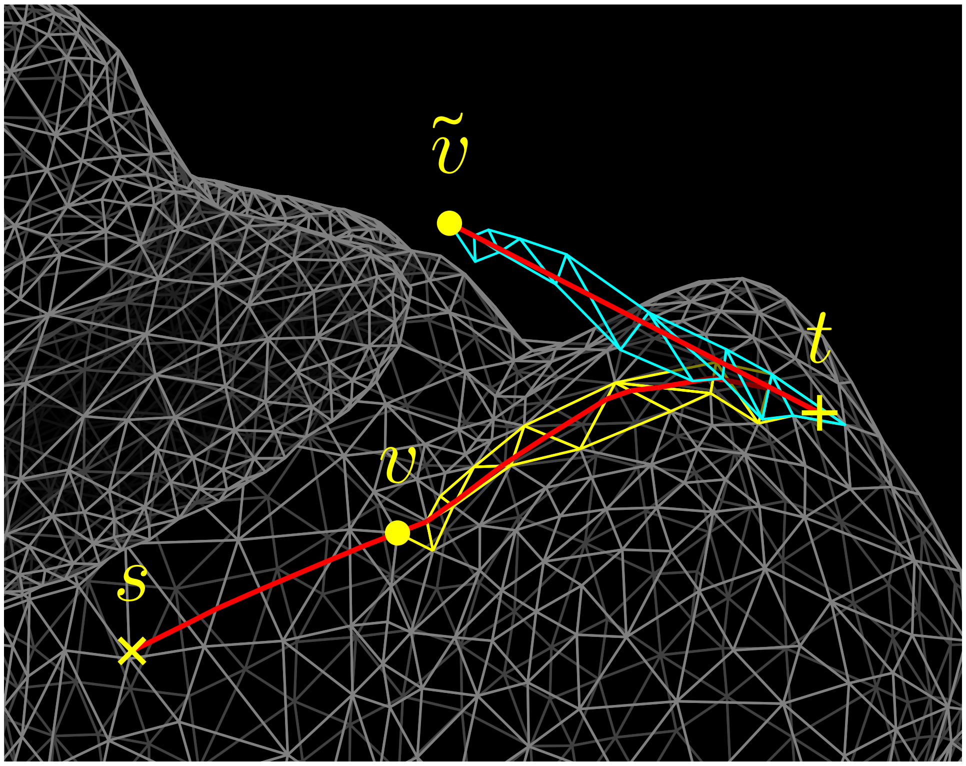


Figure 2.6: Geodesic and unfolding

vertex and t.

Figure 2.9 shows the intervals to express the geodesics immediately after passing through a vertex. We call them *pseudo-source intervals.*

The actual data owned by an interval may be slightly altered for individual applications. We will see two versions of intervals in Chapter 3 and Chapter 4 accordingly.

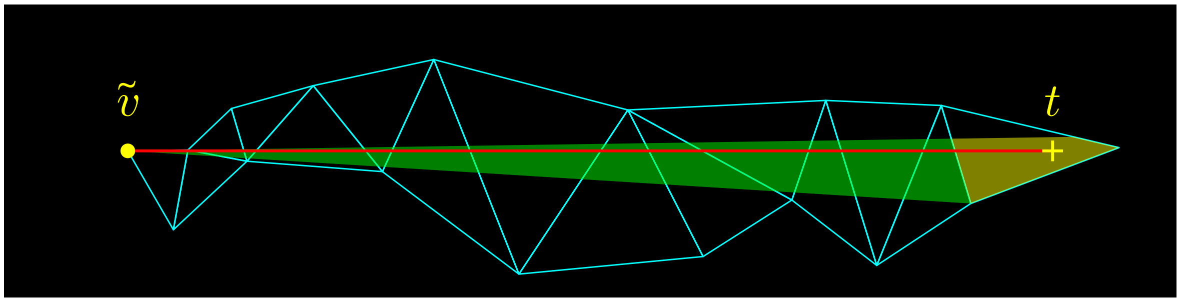


Figure 2.7: The same unfolding in 2D

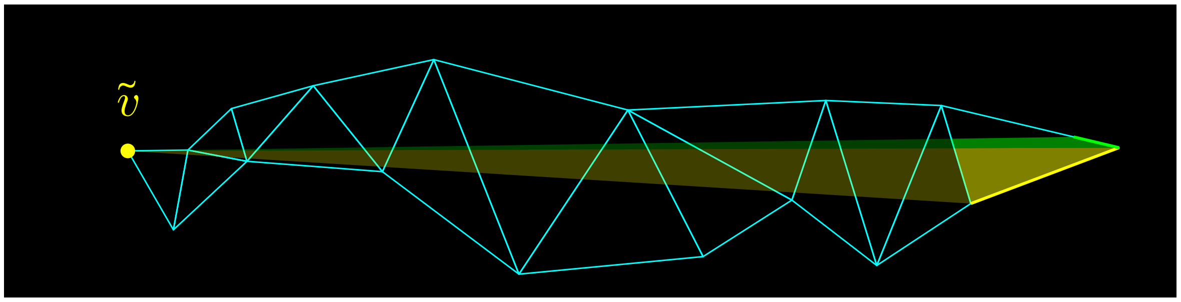


Figure 2.8: Geodesic and intervals

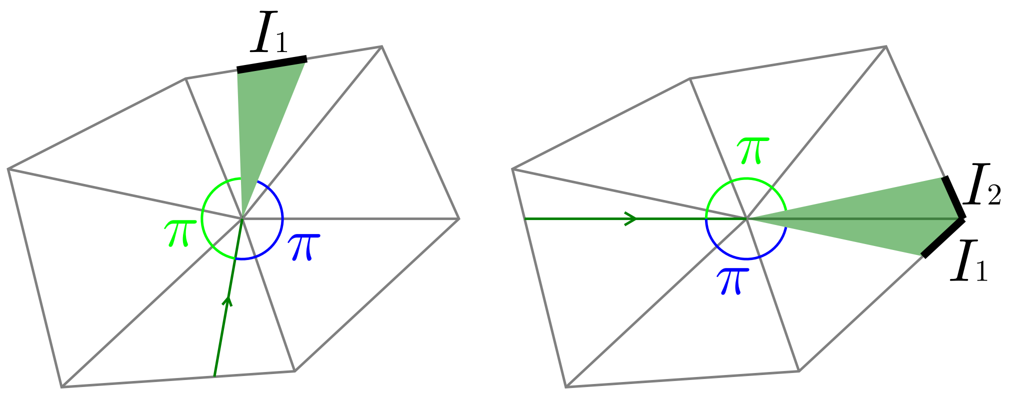


Figure 2.9: pseudo-source intervals, made at a hyperbolic vertex

Chapter 3

Enumeration of Geodesics

3.1 Our problem

Let $\mathcal{P}$ be a simplicial polyhedron in $\mathbb{R}^3$. When we give a *source* s on $\mathcal{P}$, we want to compute all geodesics from s to an arbitrary point t on $\mathcal{P}$. Since we have infinitely many choice of t, we consider a query that inputs t and outputs these geodesics. And yet, there are likely to be infinitely many such geodesics and we give an upperbound of their length R to obtain finite result.

It is widely known that single-source shortest path problems (SSSPs) for graphs or polyhedra can be efficiently solved using a queue or a priority queue. Here, we can think of a generalization of SSSPs, namely, *single-source geodesics enumeration problem* on a polyhedron.

Definition 3.1.1 (single-source geodesics enumeration problem). Suppose that $\mathcal{P}$ is a simplicial polyhedron in $\mathbb{R}^3$, s is a point on $\mathcal{P}$ and R is a positive real number. We define our *single-source geodesics enumeration problem* to be the problem of building data structure that enables to query, for any point t on $\mathcal{P}$, all geodesics from s to t whose length is less than R. We call an algorithm for this problem a *single-source geodesics enumeration algorithm*, or simply *geodesics enumeration algorithm*.

3.2 Intervals for geodesics enumeration

The basic concept of intervals is already explained in Section 2.2. Here we give a formal definition of intervals. As we stated before, the following formal definition of intervals is applied only in this chapter; We will introduce another version of intervals (enriched intervals) in Chapter 4.

Definition 3.2.1. An interval I is defined to be the following data structure:

- I.Parent: the interval which generated I (by propagation)
- I.Edge: the target edge
- I.Face: the target face
- I.Extent: the target line segment on I.Edge (identified with I itself)
- I.Center: the unfolded position of the last vertex
- I.Depth: the length of the geodesic between s and the last vertex

It is used to express all geodesics reaching the line segment I.Extent on the edge I.Edge through the last vertex. If the geodesics have not passed through a hyperbolic vertex yet, I.Center is the unfolded position of s and I.Depth is 0.

We regard I to be oriented so that I.Face is seen on the left side along I.Edge. We define the starting point of I with respect to this orientation. Moreover, I has the following two functions:

- I.Project($p : p \in I$.Face) := the intersection point of I.Edge and the line connecting p and I.Center
- I.Distance($p : p \in I$.Face) := (the distance of p and I.Center) + I.Depth

The interval I can yield a geodesic at $p \in I$.Face if I.Project(p) $\in I$.Extent, and its length is given by I.Distance(p).

Remark 3.2.2. I.Center can be expressed in either the 3D global coordinate system or a 2D local coordinate system on I.Face. 2D local coordinates are slightly faster and memory efficient, as well as ensure that given coordinates are always on the plane. Our implementation uses the 3D global coordinate system in input and output, but uses a 2D local coordinate system in internal storage and computation, and converts one to the other when necessary. That means, our algorithms could run on a polyhedron in higher-dimensional Euclidean space as well.

Remark 3.2.3. Our algorithms can also work on a self-intersecting polyhedral surface. In this case, geodesics are not bent at the intersection and pass through it as if there were no intersection — geodesics are completely defined locally. On the other hand, we assume the orientability of the surface, which may not be the case for a self-intersecting (or higher-dimensional) polyhedron. If this is an issue, one can take the orientable double covering of the surface, as demonstrated in Figure 3.1. In this figure, geodesics are computed on a discretized version of the orientable double covering, which is homeomorphic to the sphere, of the (non-orientable) Roman surface (a realization of the real projective plane in $\mathbb{R}^3$). The source s is given once, but the target t is given twice, correspondingly to its two images on the double covering. In the figure, two different colors are used accordingly.

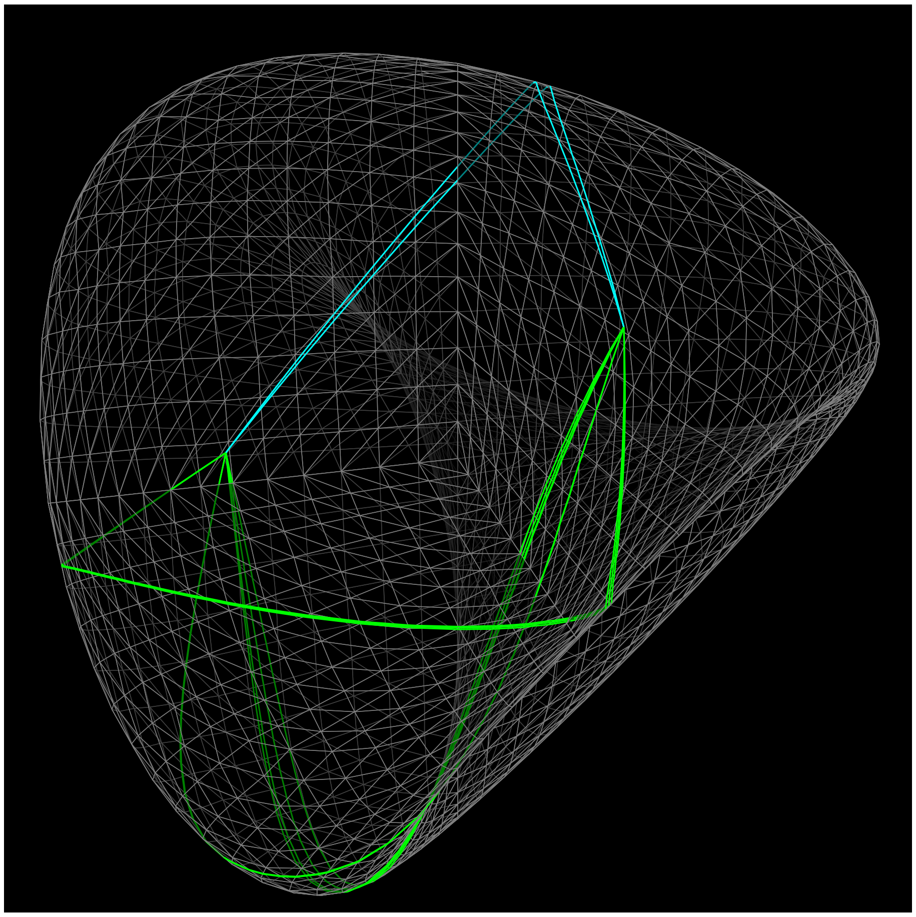


Figure 3.1: Geodesics on a discretized Roman surface

Remark 3.2.4. Our implementation uses the half-edge data structure as the internal representation of a polyhedron, thus every edge is directed. Here the 2D local coordinate system is defined by the directed edge $e = I$.Edge so that its origin is the starting point of e, its positive x direction is the direction of e, its y axis is orthogonal to the x axis and any point inside I.Face has positive y value. Complex numbers are convenient for expressing the local coordinates, because

- we often need an 1D parametric coordinate on e as well. Since a real number is considered to be a complex number whose imaginary part is zero, we can naturally treat it as a special case of 2D local coordinates;

- when we express I.Center in the 2D local coordinate system, we need to apply coordinate transformation to propagate an interval. This transformation consists of 2D rotation and translation, which can be expressed in terms of complex multiplication and addition;
- functions such as abs and arg are also useful to implement our algorithm.

3.3 Naive geodesics enumeration algorithm

We can make a naive geodesics enumeration algorithm by propagating intervals without the trimming process in the MMP algorithm. All intervals generated in this way form a tree structure by the *Parent* pointer. we call it the *complete geodesic interval tree* or the *complete GIT*.

This algorithm firstly performs initialization of generating several intervals, and proceeds by processing events. We define *propagation* to be the act of generating one or more new intervals by processing an event. Events consist of the following two types:

- edge event : when a geodesic given by I reaches I.Extent for the first time
  - this event is associated with I
  - time of occurrence: (the distance of I.Extent and I.Center) + I.Depth
- vertex event : when a geodesic reaches a vertex v
  - this event is associated with the interval I of which the starting point is v

Algorithm 1 (Building the geodesic interval tree)

```
1: function BuildGeodesicIntervalTree($s$: source, $R$: upperbound of length)
2:     $Q$ := the event queue
3:     Initialize($Q$, $s$)
4:     while the time of occurence of the top event of $Q < R$ do
5:         $Q$.Pop().Handle($Q$)
6:     end while
7: end function
```

– time of occurrence: (the distance of v and I.Center) + I.Depth

We introduce an *event queue* to manage the order of events. It is a priority queue, and lets the algorithm process events in the ascending order of their time of occurrence.

The outline is described in Algorithm 1 as a pseudocode.

Remark 3.3.1. In Algorithm 1 (and Definition 3.1.1), we assume that R is given ahead of time. Alternatively, we can rewrite the line 4 with another arbitrary terminating condition (or as an infinite loop which can be stopped by a human), and when the loop terminates, R can be obtained as the time of occurrence of the top event of Q. This also applies to the improved algorithm in the next section.

3.3.1 Initialization

Firstly, the intervals for all edges facing s are generated. Figure 3.2 shows the cases of s being inside a face (left), inside an edge (middle), and at a vertex (right). For each I of these initial intervals, I.Parent is Null, I.Center is s, I.Depth is 0, I.Extent is the whole I.Edge, and I.Face is the face containing both s and I.Edge. For each I, the associated edge event and vertex event are inserted into the event queue. (Algorithm 2)

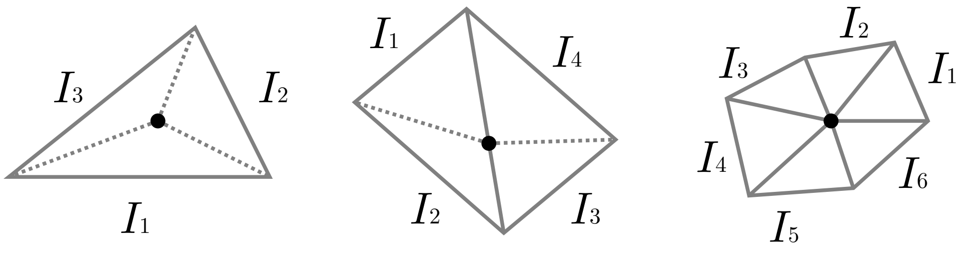


Figure 3.2: Initialization

Algorithm 2 (Initialization)

```
1: function INITIALIZE($Q$, $s$)
2:     $I_1 \dots I_k$ := the initial intervals
3:     for $i$ in $1 \dots k$ do
4:         $Q$.PUSH(EdgeEvent($I_i$))
5:         $Q$.PUSH(VertexEvent($I_i$))
6:     end for
7: end function
```

3.3.2 Procedure for edge events

When an edge event occurs, the associated interval I is projected from I.Center into the opposite edges of I.Edge (Figure 3.3). The projection result is on either one edge (left) or two edges (right). For each newly created interval I_i (left: I_1, right: I_1, I_2), I_i.Parent $= I$, I_i.Depth $= I$.Depth and I_i.Face is the triangular face in Figure 3.3. I_i.Center is obtained by certain 3D rotation around I.Edge to make it coplanar with I_i.Face. For each I_i, the associated edge event is inserted into the event queue. In the two-interval case (right), the vertex event at the intermediate vertex v is associated with the interval starting at v (that is I_2) and inserted into the event queue. (Algorithm 3)

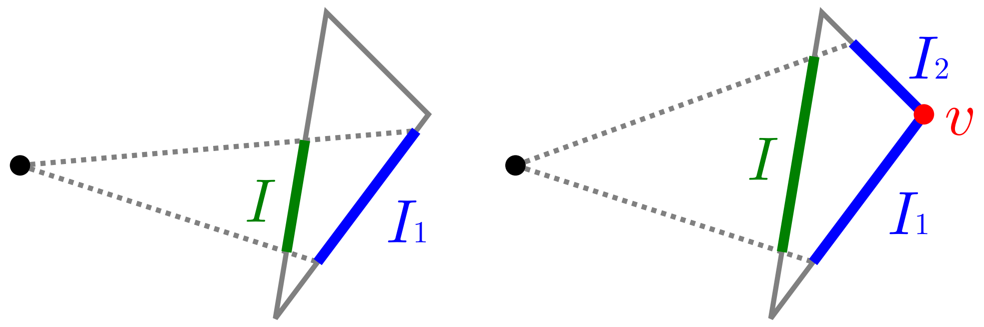


Figure 3.3: Projection of an interval

Algorithm 3 (Processing an edge event)

```
1: function EdgeEvent.Handle($Q$)
2:     $I$ := the associated interval
3:     if $I$ is projected into one interval $I_1$ then
4:         $Q$.Push(EdgeEvent($I_1$))
5:     else // $I$ is projected into two intervals $I_1$ and $I_2$
6:         $Q$.Push(EdgeEvent($I_1$))
7:         $Q$.Push(EdgeEvent($I_2$))
8:         $Q$.Push(VertexEvent($I_2$))
9:     end if
10: end function
```

3.3.3 Procedure for vertex events

When a vertex event occurs, the associated interval I is recorded on v as it gives the geodesic arriving at v. If v is hyperbolic, one or more pseudo-source intervals are generated. In Figure 2.9, one interval in the left subfigure, two intervals in the right are

Algorithm 4 (Processing a vertex event, for the complete GIT)

```
1: function VertexEvent.Handle($Q$)
2:     $I$ := the associated interval
3:     $v$ := the starting point of $I$
4:     Record $I$ on $v$
5:     if $v$ is not hyperbolic then
6:         return
7:     end if
8:     $I_1 \ldots I_k$ := the corresponding pseudo-source intervals
9:     for $i$ in $1 \ldots k$ do
10:        $Q$.Push(EdgeEvent($I_i$))
11:    end for
12:    for $i$ in $2 \ldots k$ do
13:        $Q$.Push(VertexEvent($I_i$))
14:    end for
15: end function
```

generated. For each newly-created interval I_i, I_i.Parent $= I$, I_i.Center $= v$, I_i.Depth is the length of the geodesic to v (= the time of occurrence of this event), and the associated edge event and vertex event (if exists) are inserted into the event queue. (Algorithm 4)

3.3.4 Geodesics enumeration query

After building the complete geodesic interval tree, it can accept the geodesics enumeration query which inputs a point t on $\mathcal{P}$ and outputs the set G_{st}^R of geodesics from s to

t, whose length is less than R. First, we must determine which intervals are responsible for the output. It can be done using the GetIntervals function:

Definition 3.3.2. We define the GetIntervals(t) function as follows:

- If t is a vertex v, GetIntervals(t) returns the set of intervals recorded on v.
- If t is on an edge e, GetIntervals(t) returns the set of all generated intervals I such that I.Edge $= e$, $t \in I$.Extent and I.Distance$(t) < R$.
- If t is on a face f, GetIntervals(t) returns the set of all generated intervals I such that I.Face $= f$, I.Project$(t) \in I$.Extent and I.Distance$(t) < R$.

For each obtained interval, using Algorithm 5, the geodesic is constructed from t to s by backtracking I.Parent. The whole procedure is given by Algorithm 6. In the pseudocode, Intersect is the function of getting the intersection, and AddFront is the operation of inserting a point at the front of the list.

3.4 Improved geodesics enumeration algorithm

The difference between the MMP algorithm and the geodesics enumeration algorithm in the previous section based on the complete geodesic interval tree is that, since we are also interested in non-shortest geodesics, our intervals are never trimmed. However, this change largely increases the computational complexity. Here, when $\mathcal{P}$ is non-convex, we can reduce required amount of time and memory by placing pseudo-source intervals without overlap. We can understand the redundancy of the naive algorithm using Figure 3.4. In the figure, the source s is indicated as the yellow cross sign ($\times$) and the target t is indicated as the yellow plus sign ($+$). While multiple geodesics

Algorithm 5 (Construct a geodesic)

```
1: function ConstructGeodesic($I$, $p$)
2:     $g := (p)$
3:     while $I$.Parent $\neq$ Null do
4:         $e := I$.Parent.Edge
5:         $p :=$ Intersect($e$, the line segment connecting $p$ and $I$.Center)
6:         $g$.AddFront($p$)
7:         $I := I$.Parent
8:     end while
9:     $g$.AddFront($s$)
10:    return $g$
11: end function
```

Algorithm 6 (Geodesics enumeration query, for the complete GIT)

```
1: function GeodesicEnumQuery($t$)
2:     $G := \emptyset$ (the set of geodesics for output)
3:     for $I$ in GetIntervals($t$) do
4:         $G$.Add(ConstructGeodesic($I$, $t$))                ▷ Algorithm 5
5:     end for
6:     return $G$
7: end function
```

are outgoing from s, they merge at some hyperbolic vertices until reaching t and they all share some part near t. However, the naive algorithm cannot recognize and utilize this property; the shared part of the geodesics is independently encoded in multiple intervals and rediscovered for each geodesic during the query process. The basic idea of improvement is shown in Figure 3.5. In this figure, a geodesic g_1 already arrived at a hyperbolic vertex and yielded a pseudo-source interval I_1. Now, another geodesic g_2 arrives at the vertex. Although g_2 can go through the dotted blue line, I_2 can be chosen to exclude the region already searched by I_1. As we will see later, we can still restore all geodesics in the query process.

3.4.1 Procedure of reduction

For the purpose of generalizing this argument, for each hyperbolic vertex v, we arbitrarily choose an edge e_v among those incident to v and we fix the choice (Figure 3.6). It allows us to numericalize the direction of g (seen from v) into α (measured along the faces). When g is incoming into v, we call α the *incoming angle* of g, and when g is outgoing from v, we call α the *outgoing angle* of g. Also, we use the term *outgoing angle range* $\iota = [\mu, \nu]$ $(\nu - \mu < \tau)$ to assert that all values within this range are considered as outgoing angles, here τ is the total angle of v. By convention, if $\mu > \nu$ then ι is empty, and if $\nu \geq \tau$ then all values within ι are subject to the "mod τ" operation.

Let us explain how it works in the example shown in Figure 3.7. In this figure, the five geodesics $g_1, \ldots, g_5$ of incoming angles $\alpha_1, \ldots, \alpha_5$ (respectively) arrive at the vertex in this order. Here we assume $\alpha_1 < \alpha_2 < \alpha_5 < \alpha_4 < \alpha_3 < \alpha_1 + \tau$. Each incoming angle α_i is mapped to the corresponding outgoing angle range $\iota_i = [\mu_i, \nu_i]$ as follows:

1. α_1: the outgoing angle range is $\iota_1 = [\alpha_1 + \pi, \alpha_1 - \pi + \tau]$ and corresponding one pseudo-source interval (not shown) is generated.

2. α_2: μ_2 is limited by g_1 while ν_2 is not, thus $\iota_2 = [\alpha_1 - \pi + \tau, \alpha_2 - \pi + \tau]$ and two corresponding pseudo-source intervals (not shown) are created.

3. α_3: neither g_1 nor g_2 limits ι_3, thus $\iota_3 = [\alpha_3 + \pi, \alpha_3 - \pi + \tau]$.

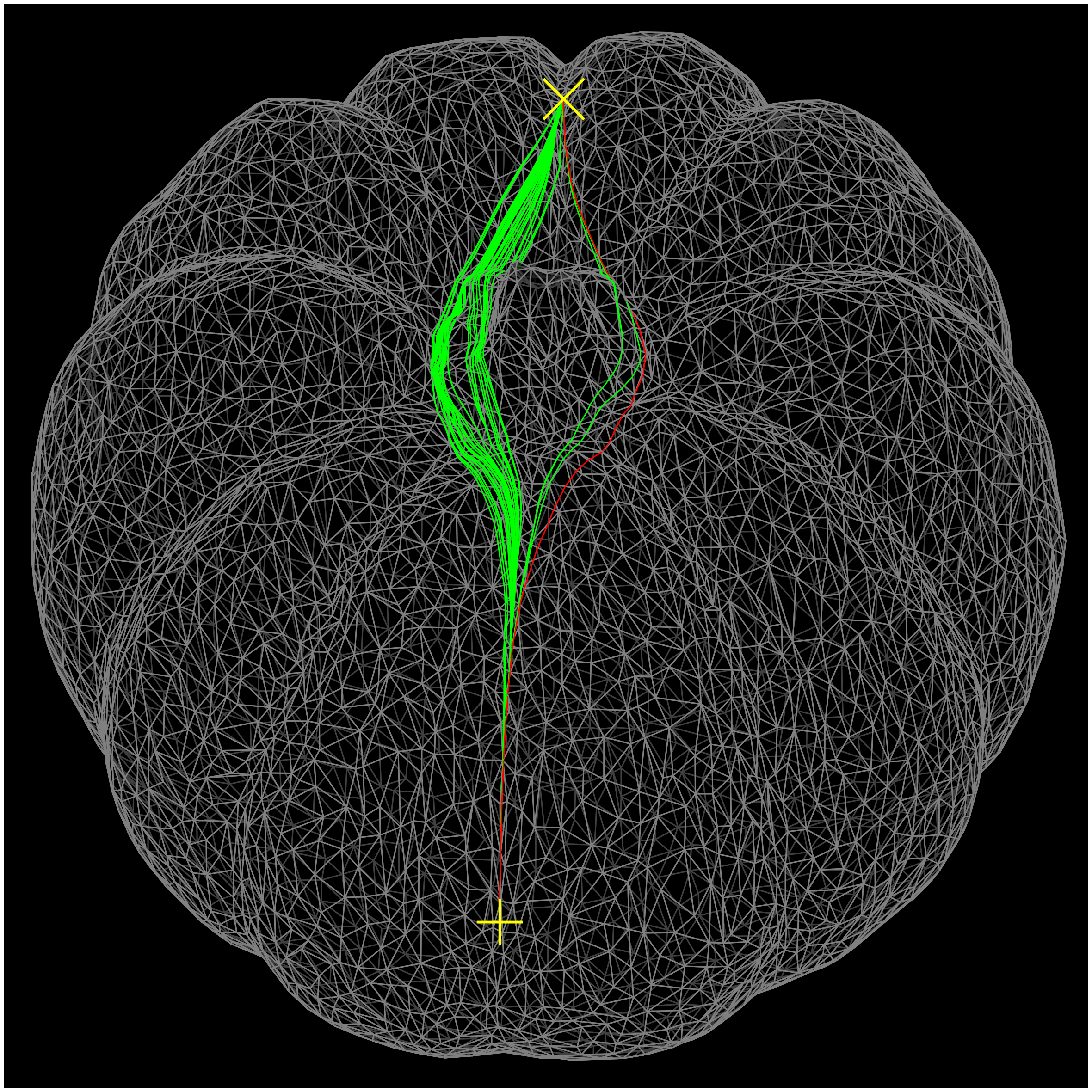


Figure 3.4: Geodesics on a pumpkin

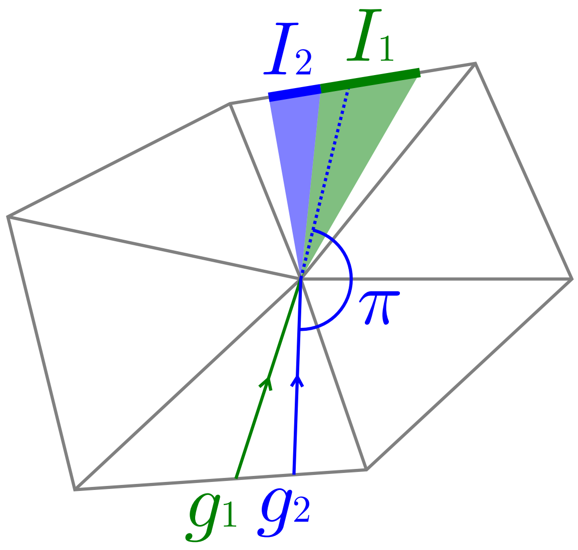


Figure 3.5: When g_2 arrives at a hyperbolic vertex after g_1, I_2.Extent is chosen not to have overlap with I_1.Extent.

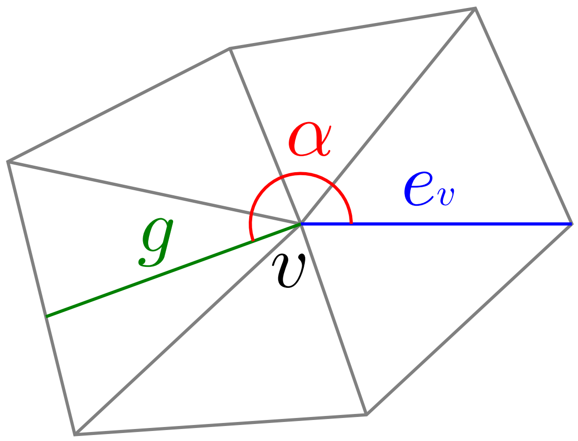


Figure 3.6: Fix e_v and encode the direction of g into α

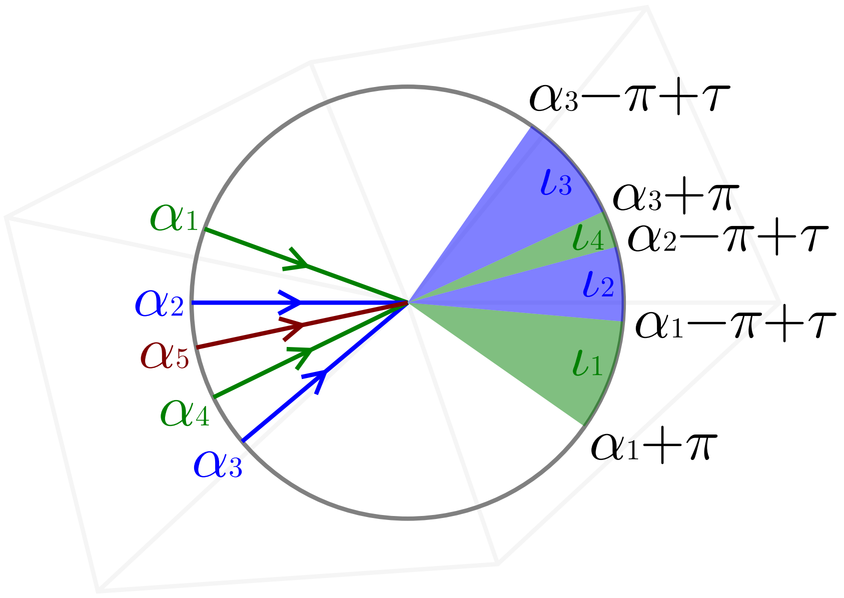


Figure 3.7: Incoming angles α_i and outgoing angle ranges ι_i. $\iota_5 = \emptyset$

4. α_4: ι_4 is limited by g_2 and g_3, thus $\iota_4 = [\alpha_2 - \pi + \tau, \alpha_3 + \pi]$.

5. α_5: ι_5 is limited by g_2 and g_4 thus ι_5 is temporarily computed as $\iota_5 = [\alpha_2 - \pi + \tau, \alpha_4 + \pi]$. However, it is empty and no pseudo-source intervals are created.

Since the outline, initialization and procedure for edge events (Algorithm 1, 2, 3) are identical, we only explain the procedure for vertex events and geodesics query. All intervals generated in this way form a tree structure by the *Parent* pointer. we call it the *reduced geodesic interval tree* or the *reduced GIT*.

Remark 3.4.1. In the reduced geodesic interval tree, only pseudo-source intervals generated at the same vertex are ensured to be disjoint. There may be an overlap between two non-pseudo-source intervals, or between a pseudo-source interval and a non-pseudo-source interval.

3.4.2 Procedure for vertex events

Like the naive version, when a vertex event occurs, the associated interval I is recorded on v. If v is hyperbolic, it performs the procedure of the reduction explained in the previous subsection, and, if the resulting outgoing angle range is not empty, pseudo-source intervals are generated and the associated edge events and vertex events (if exist) are inserted into the event queue. (Algorithm 7)

3.4.3 Geodesics enumeration query

In this subsection, we discuss the geodesics enumeration query for the reduced geodesic interval tree, which inputs a point t on $\mathcal{P}$ and outputs the set G_{st}^{R} of directed geodesics from s to t, whose length is less than R.

In the complete geodesic interval tree, a geodesic is given for each pair (I, p) by the ConstructGeodesic function (Algorithm 5). On the other hand, in the reduced geodesic interval tree, geodesics of the same path between the last vertex v and p are given together for the pair (I, p). The primitive geodesic between vp is given by the ConstructPrimitiveGeodesic function (Algorithm 8). The geodesic is constructed from t to s, and every time it hits a hyperbolic vertex, possible branches must be enumerated.

Let us take a look at the example illustrated in Figure 3.8. In this figure, each arrow indicates the direction from s to t, although the actual query is performed backwards. Let us assume that we have just found a geodesic g outgoing from v, and there are three geodesics g_1, g_2 and g_3 incoming into v, and each of g_1 and g_2 is connectable with g as a geodesic while g_3 is not. Then, for each of g_1 and g_2, we check it can satisfy the length upperbound, and if so, we connect it with g and perform this process recursively.

Algorithm 7 (Processing a vertex event, for the reduced GIT)

```
function VertexEvent.Handle($Q$)
    $I$ := the associated interval
    $v$ := the starting point of $I$
    Record $I$ on $v$
    if $v$ is not hyperbolic then return
    $\alpha$ := the incoming angle of the geodesic at $v$
    $\tau$ := the total angle of $v$
    $\delta := \tau - 2\pi$
    if there exist no incoming angles within $(\alpha - \delta, \alpha)$ then
        $\mu := \alpha + \pi$
    else
        $\mu :=$ (the largest incoming angle within $(\alpha - \delta, \alpha)$) $- \pi + \tau$
    end if
    if there exist no incoming angles within $(\alpha, \alpha + \delta)$ then
        $\nu := \alpha - \pi + \tau$
    else
        $\nu :=$ (the smallest incoming angle within $(\alpha, \alpha + \delta)$) $+ \pi$
    end if
    if $\mu \geq \nu$ then return
    $I_1 \dots I_k$ := the pseudo-source intervals for the outgoing angle range $[\mu, \nu]$
    for $i$ in $1 \dots k$ do
        $Q$.Push(EdgeEvent($I_i$))
    end for
    for $i$ in $2 \dots k$ do
        $Q$.Push(VertexEvent($I_i$))
    end for
end function
```

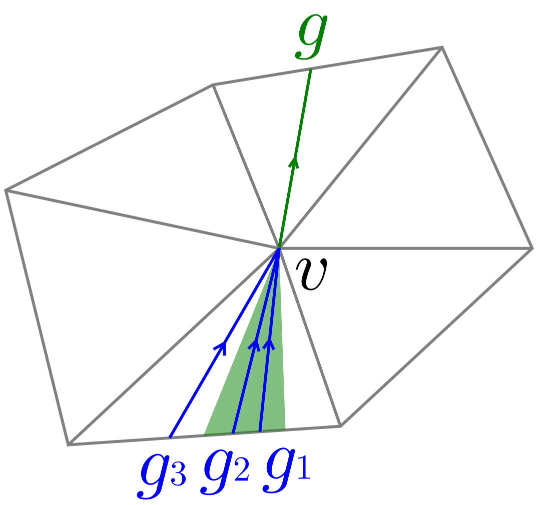


Figure 3.8: During the geodesics query, a geodesic is traced backwards. For the geodesic g outgoing from v, we enumerate all connectable geodesics (g_1 and g_2) incoming into v and check whether each can satisfy the upperbound. We recursively perform this process.

In general, we can use a simple depth-first search here. For each interval I, I.Depth is the minimum of the depths of these geodesics grouped together. Since G_{st}^{R} contains at least one geodesic represented by the pair (I, t) if and only if the minimum of their lengths is less than R, the procedure of the GetIntervals function (Definition 3.3.2) is identical. The whole procedure is given by Algorithm 9. In the pseudocode, RemoveLast is the operation of removing the last point from the sequence, and is used for endpoints of primitive geodesics not to appear twice.

Algorithm 8 (Construction of a primitive geodesic)

```
1: function ConstructPrimitiveGeodesic($I$, $p$) // $p \in I$.Face
2:     $g := (p)$
3:     // when reached $s$, $I$.Parent = Null
4:     // when reached a hyperbolic vertex, $I$.Parent.Depth < $I$.Depth
5:     while $I$.Parent $\neq$ Null and $I$.Parent.Depth = $I$.Depth do
6:         $e := I$.Parent.Edge
7:         $p :=$ Intersect($e$, the line segment connecting $p$ and $I$.Center)
8:         $g$.AddFront($p$)
9:         $I := I$.Parent
10:    end while
11:    $g$.AddFront($I$.Center)                ▷ $I$.Center is $s$ or a hyperbolic vertex
12:    return ($g$, $I$.Parent = Null)          ▷ tuple of a path and a Boolean value
13: end function
```

3.4.4 Constructing single-pair geodesic graph

Since a geodesic is a sequence of primitive geodesics, we can reduce G_{st}^R into a graph whose edges are primitive geodesics (Figure 3.9). We call it a *single-pair geodesic graph*, or simply a *geodesic graph*. It can be computed directly using the reduced geodesic interval tree.

Definition 3.4.2 (Single-pair geodesic graph). The *(single-pair) geodesic graph* $\mathcal{G}_{st}^R$ with respect to G_{st}^R, is the directed graph satisfying the following conditions:

- The vertices of $\mathcal{G}_{st}^R$ are s, t and the vertices of $\mathcal{P}$ through which at least one geodesic in G_{st}^R passes.

Algorithm 9 (Geodesics enumeration query, for the reduced GIT)

```
1: function GeodesicEnumQuery($t$)
2:     $G := \emptyset$
3:     for $I$ in GetIntervals($t$) do                                   ▷ Definition 3.3.2
4:         $d :=$ Distance($t$, $I$.Center)              ▷ length of the primitive geodesic
5:         GeodesicEnumQueryRec($G$, $(t)$, $I$, $t$, $d$)
6:     end for
7:     return $G$
8: end function
1: function GeodesicEnumQueryRec($G$, $g$, $I$, $p$, $d$)
2:     ($h$, IsSource) := ConstructPrimitiveGeodesic($I$, $p$)          ▷ Algorithm 8
3:     $h$.RemoveLast()
4:     if IsSource then
5:         $G$.Add($h + g$)                    ▷ + denotes concatenation of sequences
6:         return
7:     end if
8:     $v :=$ the starting point of $h$                       ▷ this is a hyperbolic vertex
9:     $\alpha :=$ the outgoing angle of $h$ at $v$
10:    for all $J$ : the intervals of incoming angle within $[\alpha + \pi, \alpha - \pi + \tau]$ at $v$ do
11:        $l :=$ Distance($v$, $J$.Center)              ▷ length of the primitive geodesic
12:        if $d + l + J$.Depth $< R$ then
13:            GeodesicEnumQueryRec($G$, $h + g$, $J$, $v$, $d + l$)      ▷ recursive call
14:        end if
15:    end for
16: end function
```

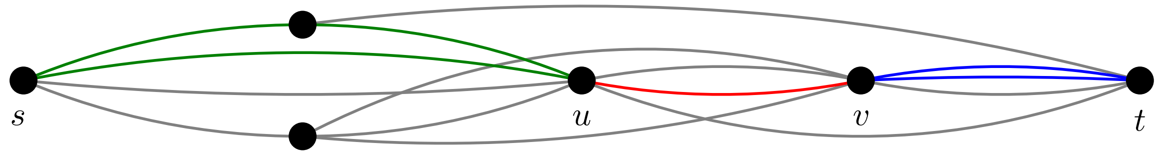


Figure 3.9: Concept of geodesic graph. Each arc represents a primitive geodesic. The red arc is connectable to the green arcs and the blue arcs.

- The edges of $\mathcal{G}_{st}^R$ are the directed primitive geodesics connecting two vertices of $\mathcal{G}_{st}^R$, such that each of them is contained in at least one directed geodesic in G_{st}^R.

Remark 3.4.3. In a geodesic graph $\mathcal{G}_{st}^R$, s and t act as the source and the sink (respectively). Even if s (resp. t) is placed at a vertex and geodesics pass through the vertex, we regard s (resp. t), as a vertex of $\mathcal{G}_{st}^R$, to be distinct from any other vertex of $\mathcal{P}$. This convention allows us to design the algorithm without special treatment of s (resp. t) being a vertex or not.

Remark 3.4.4. A geodesic graph can have multi-edges. That is, there may exist pairs of edges such that the both endpoints coincide.

Remark 3.4.5. Our geodesic graph bears some resemblance to the Saddle Vertex Graph (SVG) [22], but fundamentally differs as follows:

- The SVG only considers shortest geodesics, while our geodesic graph considers non-shortest geodesics as well.
- The SVG considers shortest geodesics for all pairs of vertices, while our geodesic graph only considers geodesics connecting s and t.
- More importantly, the SVG is constructed before their geodesic query and the query is performed on the SVG, while our geodesic query is performed on the

reduced geodesic interval tree and yields a geodesic graph. In other words, our geodesic graph is a representation of the query result.

Remark 3.4.6. Our geodesic graph $\mathcal{G}_{st}^R$ is an exact representation of the query result for the given s, t and R. The naive representation G_{st}^R often have many overlapped sub-geodesics. In this situation, we can compress them into a graph, eliminating redundancy that can be combinatorially reconstructed from the graph.

Remark 3.4.7. Every figure in this paper that renders computed geodesics (such as Figure 2.4 and Figure 3.4) is a 3D-rendered version of a geodesic graph presented in this subsection.

A geodesic graph is constructed from t to s using Algorithm 10. We use a modified version of Dijkstra's algorithm to determine the shortest possible length from t for each primitive geodesic. We have two points to remark:

- When t is given, an interval can yield at most two primitive geodesics, and each primitive geodesic can be specified by the pair $(I, \textsc{IsTarget})$ where I is an interval and IsTarget is a Boolean value indicating whether the primitive geodesic ends with t (otherwise it ends with the starting point of I). Note that IsTarget is set to be True only through the initialization of the algorithm since we regard t to be a vertex of $\mathcal{G}_{st}^R$ distinct from any other vertex (Remark 3.4.3). Since we can assume that no duplicated intervals are supplied by the GetIntervals function, we need to check visitedness only when IsTarget is set to False (the line 11–14).
- In the line 23, l is the length of the primitive geodesic given by $(J, \textsc{False})$. A conceptual figure is described in Figure 3.9. Let the red curve be the primitive

geodesic and denoted as h. Not necessarily all geodesics in G_{st}^R contain h. Let the green part and blue part be the subgraph of G_{st}^R between su and vt respectively through which a geodesic containing h can pass (in general the green and blue subgraph may be overlapped). Then, the distance between su on the green subgraph is J.Depth, because of the construction method of the reduced geodesic interval tree. Moreover, the distance between vt on the blue subgraph is d, because of the construction algorithm of the geodesic graph, which is derived from Dijkstra's algorithm. Therefore the minimum length of geodesics between st containing h is $J.\text{Depth} + l + d$, and h is contained in $\mathcal{G}$ as an edge if and only if it is less than R.

3.5 Performance

In this section, since we evaluate both the naive version that generates a complete geodesic interval tree and the improved version that generates a reduced geodesic interval tree, we call them *geodesic interval trees*. Since the size of an output geodesic interval tree greatly depends on the geometry, it is difficult to express it only in terms of input size and R. The efficiency of a reduced geodesic interval tree is due to its small size compared with the corresponding complete one, and we evaluate it by experiments in the next subsection. First, we state an output-sensitive complexity evaluation:

Theorem 3.5.1. Let $\mathcal{T}$ be a (complete or reduced) geodesic interval tree and $N = |\mathcal{T}|$ be the number of intervals in it. The corresponding algorithm for generating $\mathcal{T}$ runs in $O(N \log N)$ time and $O(N)$ space.

Proof. Since the number of the whole generated intervals is N, the numbers of edge events and vertex events are $O(N)$. Thus, the size of the event queue is $O(N)$ at

Algorithm 10 (Construction of a geodesic graph, for the reduced GIT)

```
1: function ConstructGeodesicGraph($t$)
2:     $Q_r$ := a priority queue
3:     $\mathcal{I}_{\text{vis}} := \emptyset$ (the set of visited intervals)
4:     $\mathcal{G} := \emptyset$ (the set of edges of the output geodesic graph)
5:     for $I$ in GetIntervals($t$) do                              ▷ Definition 3.3.2
6:         $d$ := Distance($t$, $I$.Center)             ▷ length of the primitive geodesic
7:         $Q_r$.Push(($I$, True, priority: $d$))
8:     end for
9:     while $Q_r$ is not empty do
10:        ($I$, IsTarget, priority: $d$) := $Q_r$.Pop()
11:        if not IsTarget then
12:            if $\mathcal{I}_{\text{vis}}$ contains $I$ then continue
13:            $\mathcal{I}$.Add($I$)
14:        end if
15:        $p$ := if IsTarget then $t$ else the starting point of $I$
16:        ($g$, IsSource) := ConstructPrimitiveGeodesic($I$, $p$)   ▷ Algorithm 8
17:        $\mathcal{G}$.AddEdge($g$)
18:        if IsSource then continue
19:        $v$ := the starting point of $h$                  ▷ this is a hyperbolic vertex
20:        $\alpha$ := the outgoing angle of $g$ at $v$
21:        for all $J$ : the intervals of incoming angle within $[\alpha+\pi, \alpha-\pi+\tau]$ at $v$ do
22:            $l$ := Distance($v$, $J$.Center)
23:            if ($\mathcal{I}_{\text{vis}}$ does not contain $J$) and ($d + l + J$.Depth $< R$) then
24:                $Q_r$.Push(($J$, False, priority: $d + l$))
25:            end if
26:        end for
27:    end while
28:    return $\mathcal{G}$
29: end function
```

any time. Therefore, the pop operation of the event queue takes $O(\log N)$ time per event. Concerning processing time of an event, an edge event can be processed in constant time. A vertex event requires time proportional to the number of the intervals it generates, but it sums up to $O(N)$. Also, in the reduced version, the two adjacent incoming angles of an incoming angle must be acquired, but it can be done in $O(\log N)$ time per event using a balanced binary search tree. Therefore, the time complexity is $O(N \log N)$. On the other hand, since the required space is the whole output tree $\mathcal{T}$ and the event queue, and each interval or event consumes constant space, the space complexity is $O(N)$. □

3.5.1 Experimental Result

For the purpose of evaluating performance of our methods, we mainly use the Elephant mesh contained in the CGAL (Computational Geometry Algorithms Library) [18]. The proposed methods consist of two parts, i.e., the construction of a geodesic interval tree, and the geodesics query or the construction of a geodesic graph. Since the geodesics query consumes much less (often negligible) time, we only evaluate the performance of the construction of a geodesic interval tree, except Figure 3.25. We implemented our (single-threaded) algorithms in C++ and tested them using a machine with Intel(R) Core(TM) i9-9980XE CPU @ 3.00GHz and 128GB RAM running Linux. Except Figure 3.18, 3.19 and 3.25, we ran our program for 300 seconds and took statistics every second. In Figure 3.18 and 3.19, we manually recorded the amount of memory consumption measured by the OS, when it elapsed 10, 20, 30, 40, 60, 80, 100, 125, 150, 180, 210, 240, 270 and 300 seconds since each program started.

Relation of R (normalized so that the mean edge length is 1) and running time is shown in Figure 3.11, and its log-linear plot and log-log plot are shown in Figure 3.12

and Figure 3.13. We can observe that the running time of the naive version grows exponentially to R, while that of improved version grows more slowly. A log-linear plot and log-log plot of the total number of generated intervals $|\mathcal{T}|$ are shown in Figure 3.14 and Figure 3.15, and they exhibit a pattern similar to that of the computation time. Figure 3.16 shows the slope of the log-log plot, i.e. $\Delta(\log|\mathcal{T}|)/\Delta(\log R)$, which can be considered as the exponent α when $|\mathcal{T}|$ is locally fit by kR^α (k and α are constants). This value is increasing in the naive version but decreasing in the improved version, as R increases. Figure 3.17 shows relation of $|\mathcal{T}|$ and computation time, and exhibits a pattern similar to the quasilinear growth. Memory consumption is shown in Figure 3.18 (to R) and Figure 3.19 (to $|\mathcal{T}|$). We can observe that the memory consumption seems to grow linearly to $|\mathcal{T}|$. Figure 3.20 shows the ratio of the number of propagating vertex events (vertex events that generated at least one interval) to the number of hyperbolic vertex events (vertex events that occurs at a hyperbolic vertex), among newly-processed events in one second. This value is 1 in the naive version but around 0.2 in the improved version in this instance, which means that, in the improved version, there are less vertex events that actually generate intervals compared with the naive version.

We also tested on a synthesized simple torus (Figure 3.22, 3.23, 3.24). We can observe that, for sufficiently large R, the total number of generated intervals $|\mathcal{T}|$ in the improved version is approximately $\Theta(R^3)$. We have not established a theory, but we could *guess* the reason behind it as follows:

- For sufficiently (very) large R (and in generic cases), pseudo-source intervals are likely to "fill up" all angles around hyperbolic vertices so that no new pseudo-source intervals are generated.

- In this situation, the area "swept" by the imaginary wavefront of intervals is

$\Theta(R^2)$. That is, the number of vertex events is likely to be $\Theta(R^2)$. Since an interval in the wavefront splits into two intervals when it meets a vertex, the number of intervals in the wavefront is also likely to be $\Theta(R^2)$. Since $|\mathcal{T}|$ can be obtained by summing the size of each generation, it is likely to be $\Theta(R^3)$.

To evaluate relationship of smoothness of the mesh and performance, we used the recursively subdivided surfaces of Elephant using the Loop subdivision scheme [10]. In Figure 3.21, the computation time is shown for the original mesh (orig), and the surface subdivided once (sub1) and twice (sub2). Here R is relative to the mean edge length of the original mesh. We can see that the computation time of the improved version becomes closer to that of the naive version as the mesh becomes more detailed. The reason behind it is that, pseudo-source intervals become more unlikely to overlap in the naive version, since the total angle of each vertex becomes closer to 2π. The surface subdivided twice (sub2) from Elephant is used in Figure 3.25 to evaluate practical performance. The detail of the experiment is explained in the caption of this figure, but compared with the naive version, we can observe that the improved version took nearly half computation time and used approximately 56% memory to produce this result.

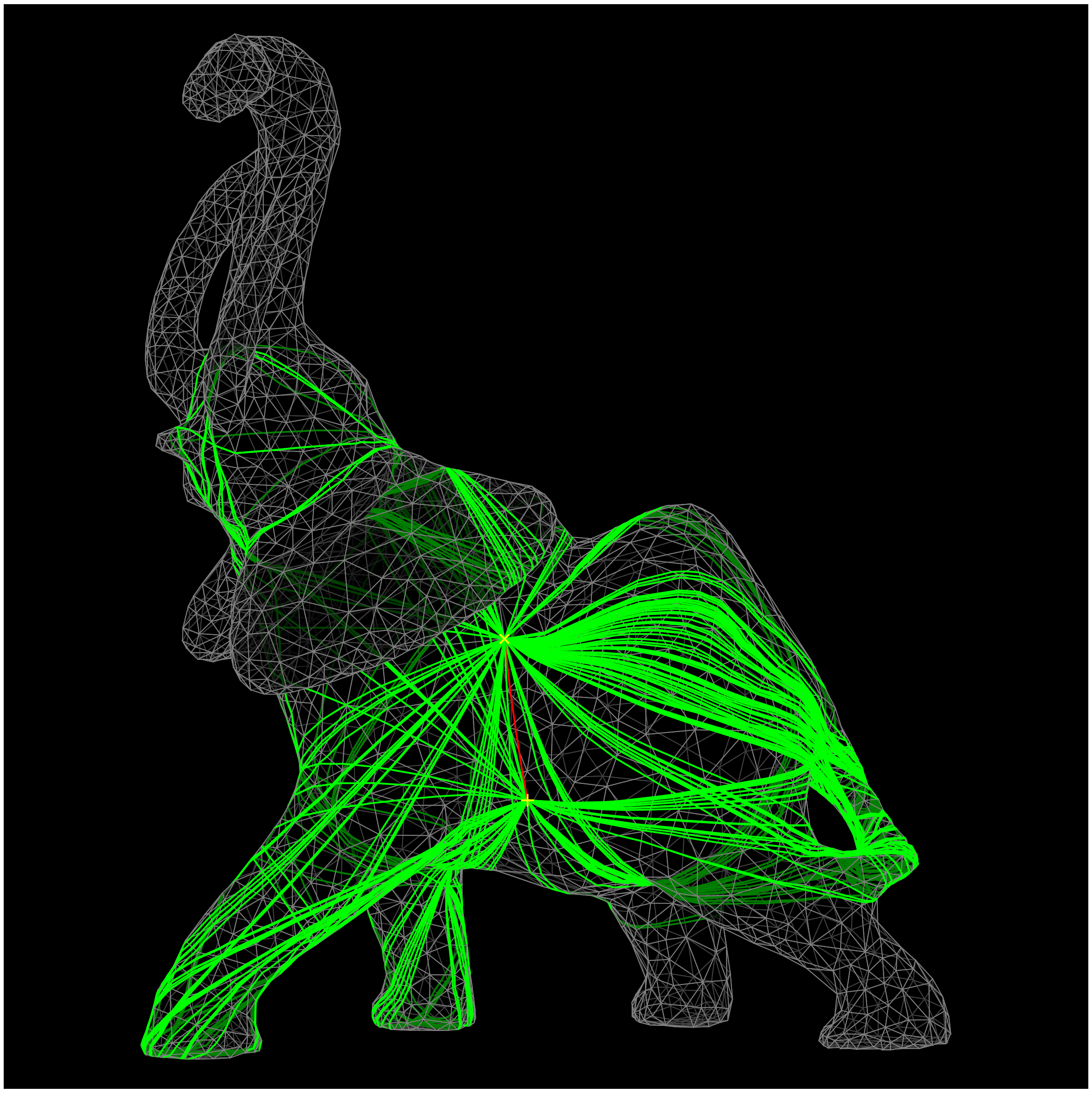


Figure 3.10: Elephant (2775 vertices and 5558 faces). Figures 3.11–3.21 are concerned on this mesh.

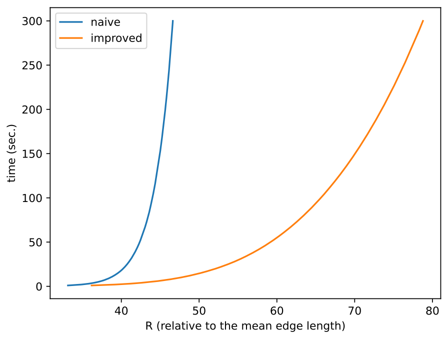


Figure 3.11: Linear plot of time vs R

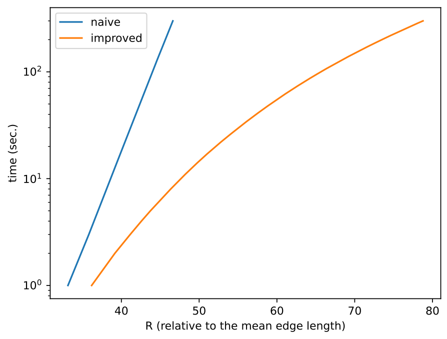


Figure 3.12: Log-linear plot of time vs R

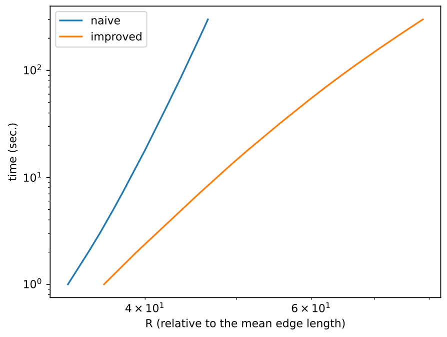


Figure 3.13: Log-log plot of time vs R

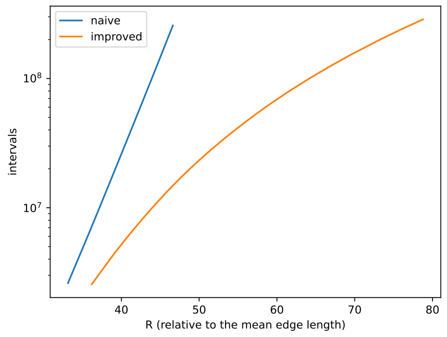


Figure 3.14: Log-linear plot of $|\mathcal{T}|$ vs R

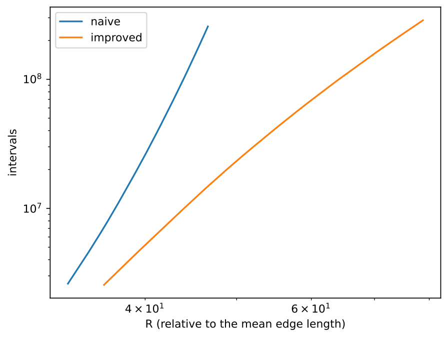


Figure 3.15: Log-log plot of $|\mathcal{T}|$ vs R

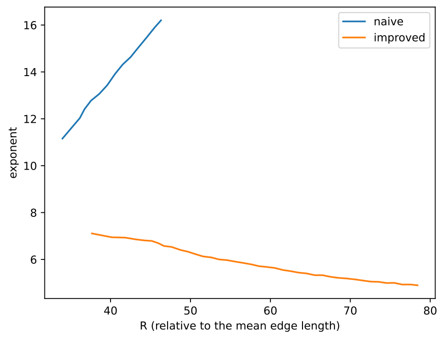


Figure 3.16: $\Delta(\log|\mathcal{T}|)/\Delta(\log R)$ vs R

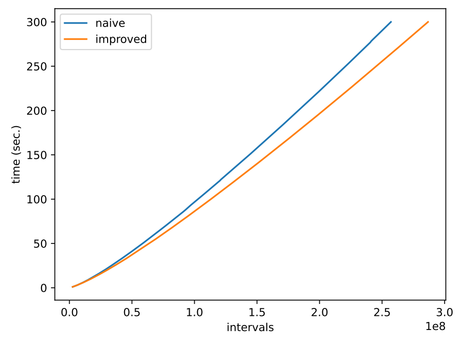


Figure 3.17: Time vs $|\mathcal{T}|$

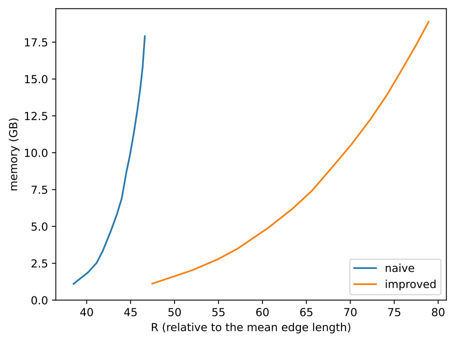


Figure 3.18: Memory vs R

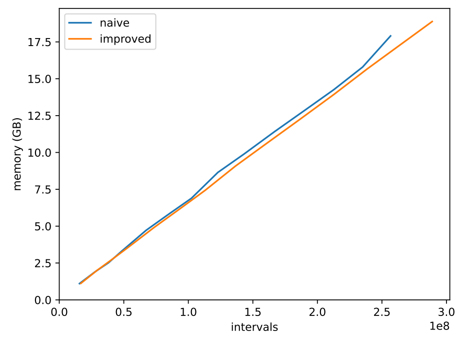


Figure 3.19: Memory vs $|\mathcal{T}|$

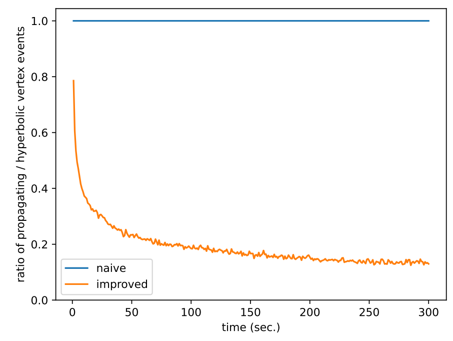


Figure 3.20: Ratio of the propagating vertex events to the hyperbolic ones among newly-processed events in one second

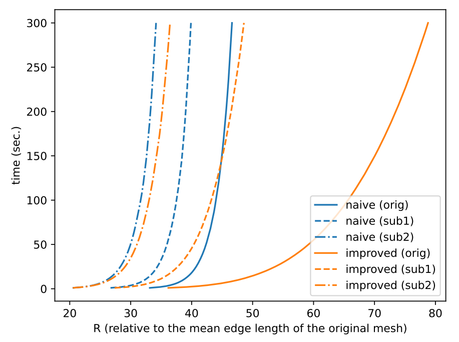


Figure 3.21: Subdivision and computation time of Elephant; sub1: 11K vertices, sub2: 44K vertices

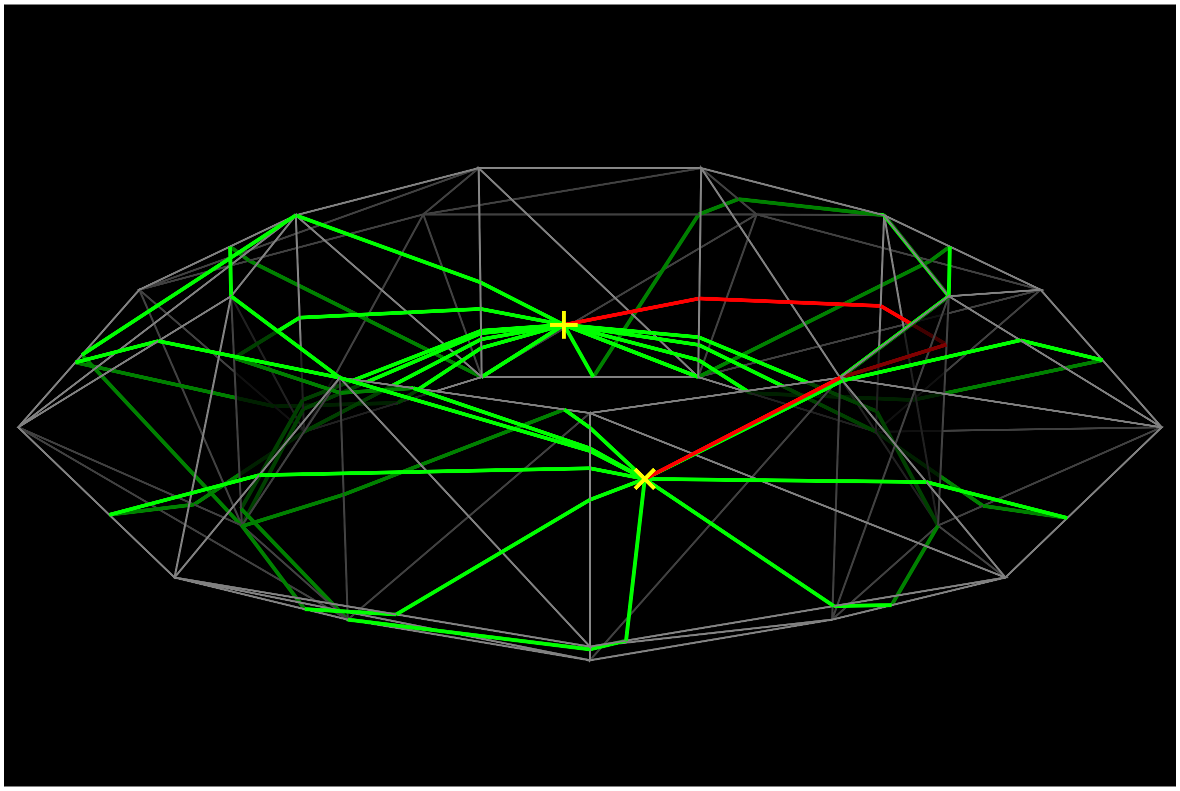


Figure 3.22: Simple Torus. Figures 3.23–3.24 are concerned on this mesh.

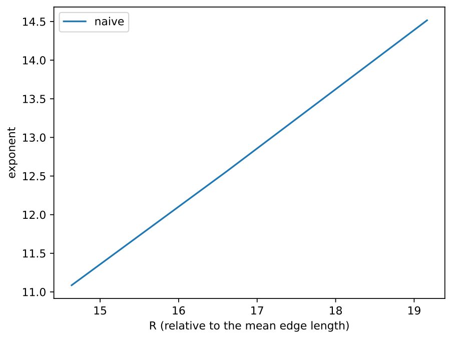


Figure 3.23: $\Delta(\log|\mathcal{T}|)/\Delta(\log R)$ vs R in the naive version

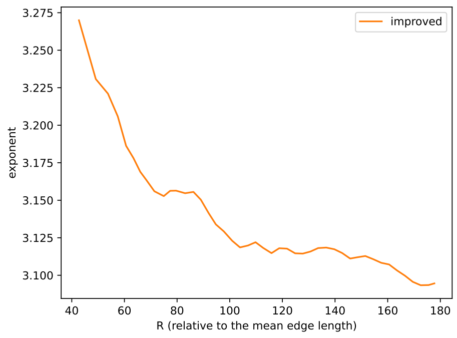


Figure 3.24: $\Delta(\log|\mathcal{T}|)/\Delta(\log R)$ vs R in the improved version

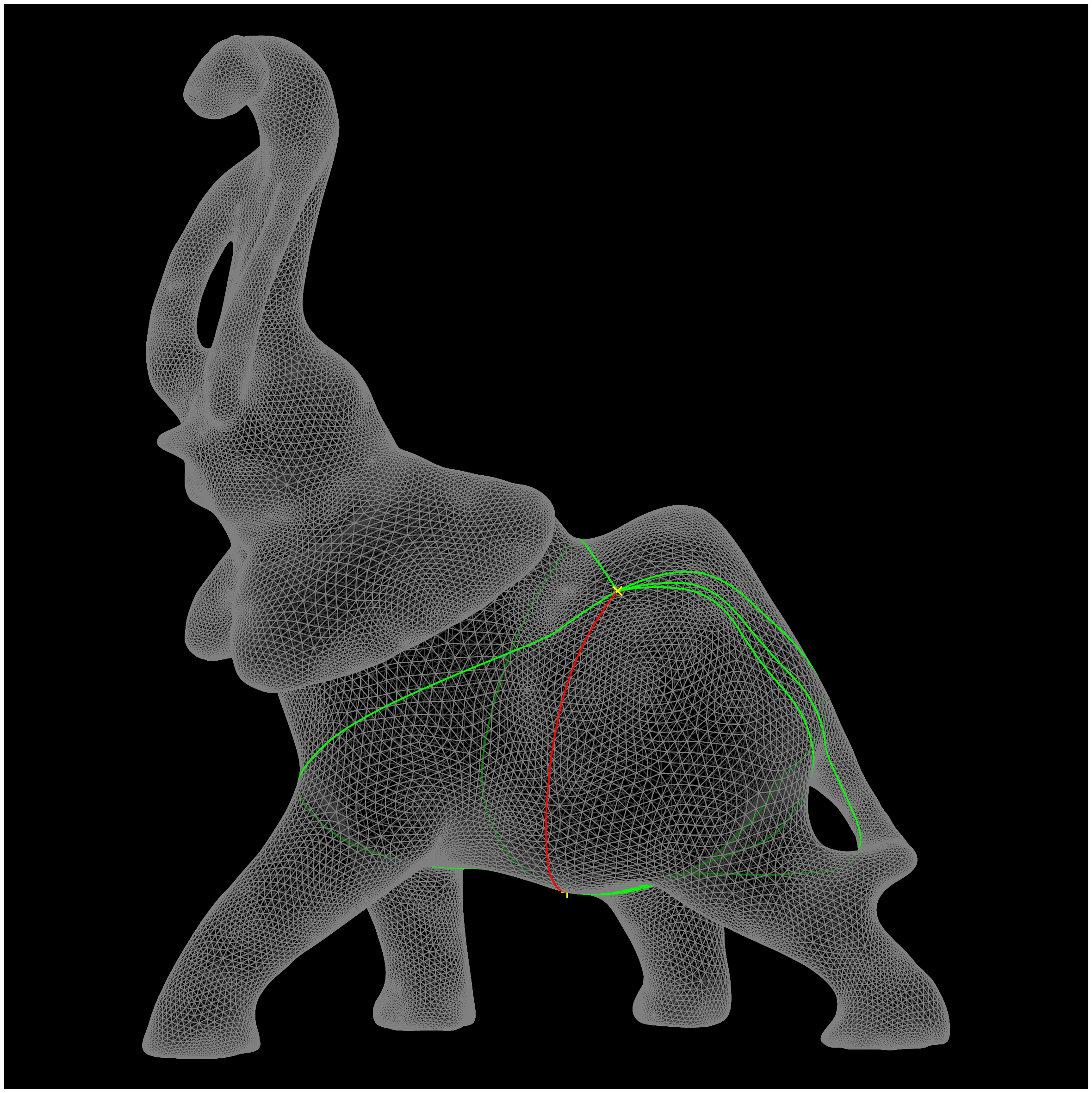


Figure 3.25: Surface (with 44460 vertices) obtained by subdividing twice from Elephant, $R = 131$ (relative to the mean edge length); building the geodesic interval tree took 60.4 seconds (naive), 30.3 seconds (improved) and the total memory consumption (measured by the OS) is 5.5GB (naive), 3.1GB (improved); 83K intervals (naive), 46K intervals (improved) are generated; geodesics query took less than 1ms (both version).

Chapter 4

Cut-Locus on a Convex Polyhedron

4.1 Overview

As we have seen before, the interval is an useful framework for polyhedral geodesics, and the MMP algorithm may be used as a starting point to make algorithms relating to the geodesics. In this chapter, we study the problem of propagating the wavefront and computing the cut locus on a convex polyhedron, and we propose an algorithm based on the MMP algorithm. While our method has some limitations (discussed later), we emphasize our actual implementation to computer program for instantaneous visualization and numerical computation [1].

As a toy example, look at Figure 4.1. Here we take the convex hull of *Stanford Bunny* (the upper-left picture, viewed transparently). On this polyhedron, we choose freely a source point indicated by $\times$ colored by yellow, then our algorithm creates the time-evolution of wavefronts (yellow curve in the upper-right figure) instantaneously and accurately enough, and finally it ends at the lower picture. Red dots represent

[1]The source code is available at https://github.com/Raysphere24/IntervalWavefront

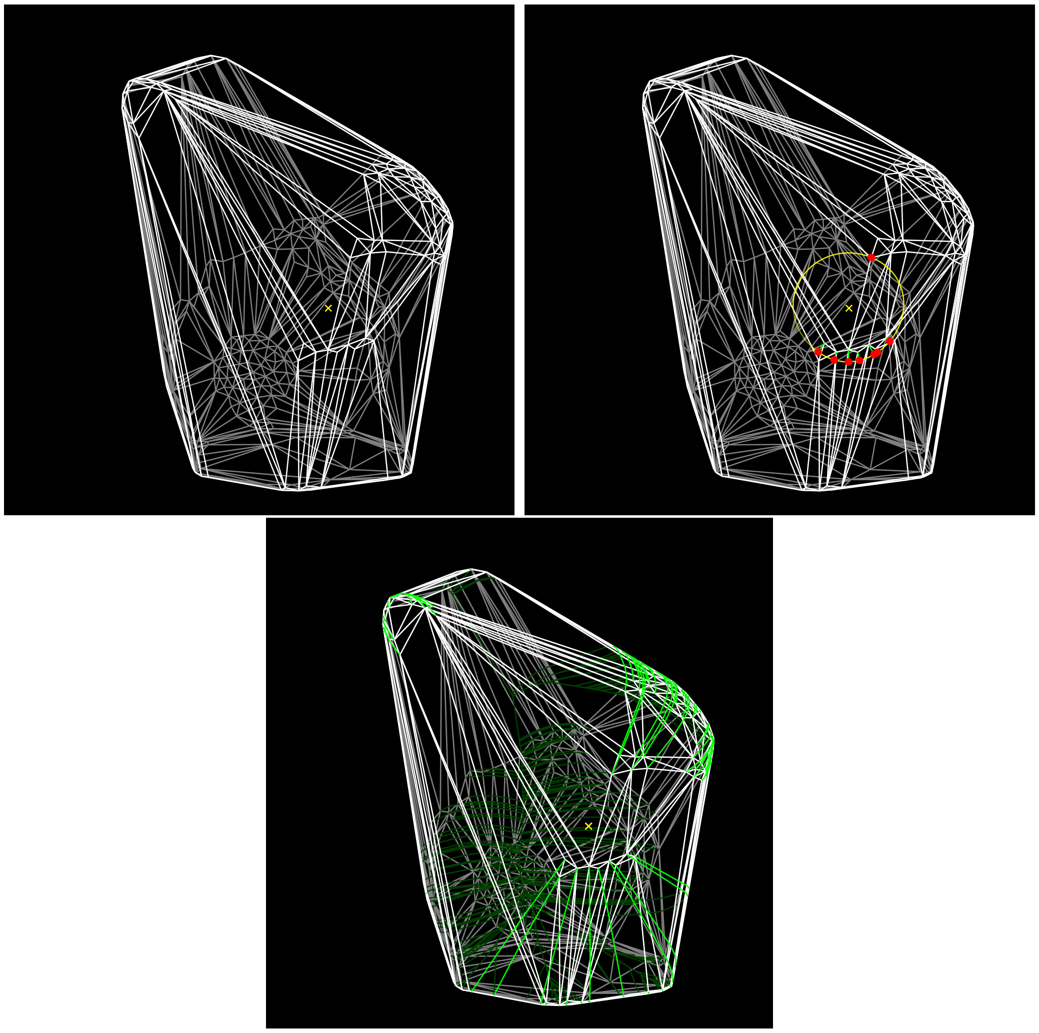


Figure 4.1: Wavefront and cut-locus.

ridge points on the wavefront curve. As the time increases, the ridge points sweep out the cut-locus colored by green. In Figure 4.2, the wavefront propagation is observed from different viewpoints, and in Figure 4.3 the cut locus is viewed opaquely.

First, we fix basic terminologies precisely. Let $\mathcal{P}$ be a convex polyhedral surface in Euclidean space $\mathbb{R}^3$. In this chapter, we only consider convex polyhedra, thus the

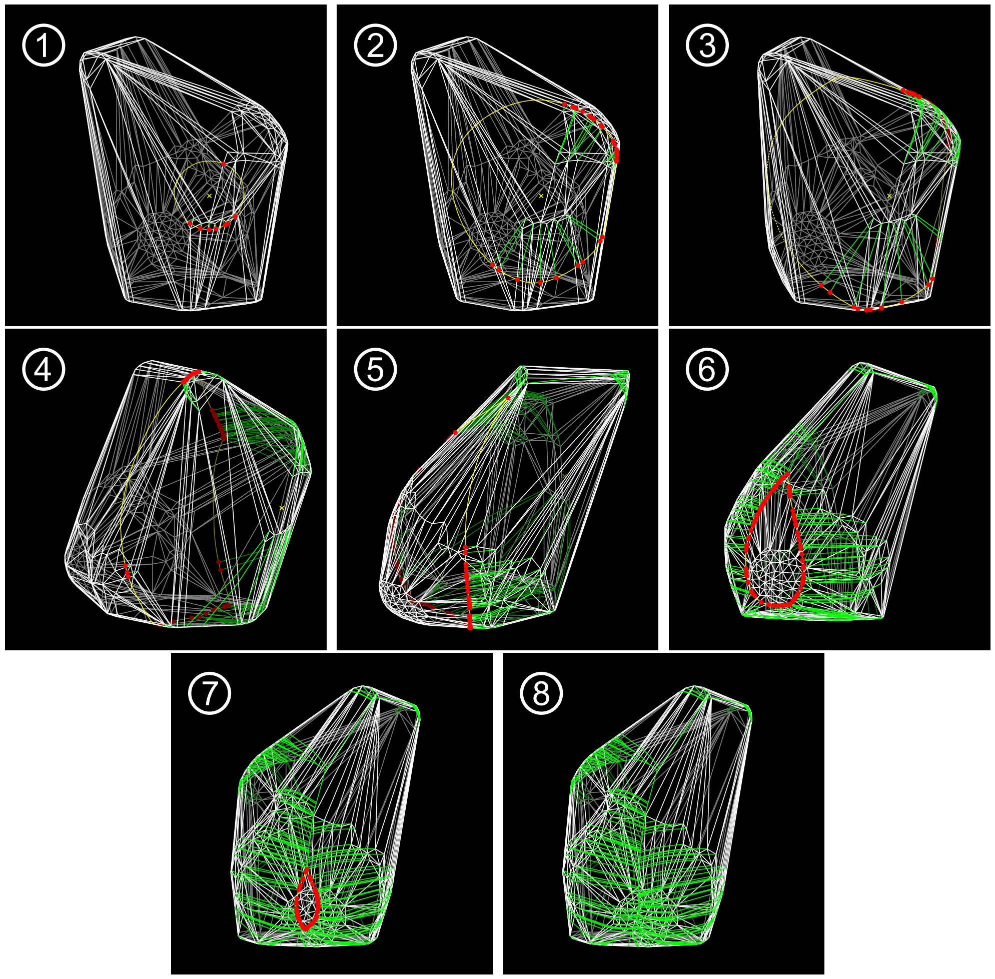


Figure 4.2: The wavefront $W(r)$ (yellow curve) propagates from no.1 to no.8. In our program, the viewpoint can be chosen freely in an interactive way; no.8 is a different view of the right picture in Figure 4.1.

symbol $\mathcal{P}$ only represents a convex polyhedron. Let $d : \mathcal{P} \times \mathcal{P} \to \mathbb{R}$ denote the distance function on $\mathcal{P}$. Pick a point s of $\mathcal{P}$, and call it the *source point*. Given $r > 0$, the *wavefront* on $\mathcal{P}$ caused from s is defined by the set of points of $\mathcal{P}$ with iso-distance r

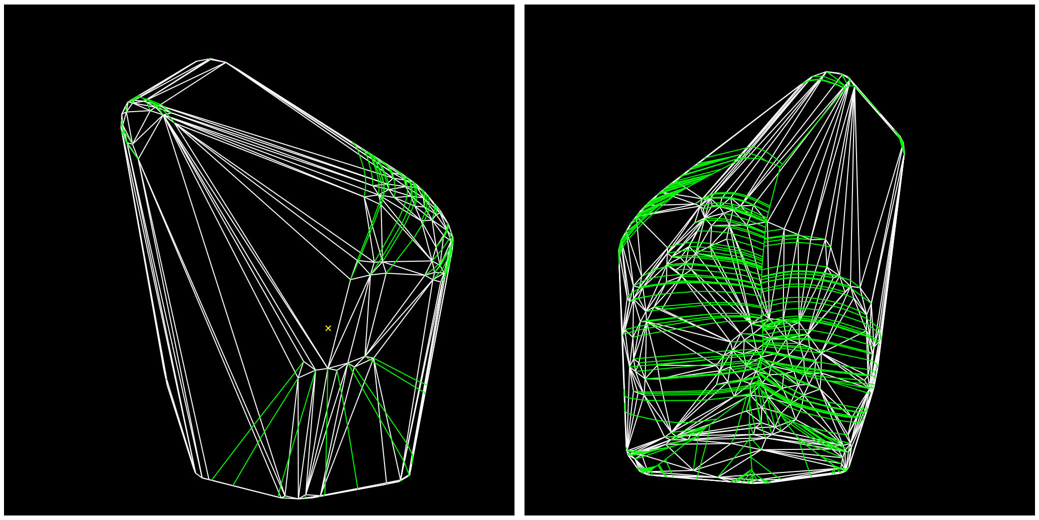


Figure 4.3: Opaque renderings of the cut-locus (two different viewpoints)

from s:

$$W(r) := \{\ x \in \mathcal{P} \mid d(s,x) = r\ \}$$

(also we may write it by $W(r,s)$). Suppose that the wavefront $W(r)$ propagates on $\mathcal{P}$ with a constant speed, as r varies. If s is an interior point of a face, then initially $W(r)$ is just a circle centered at s with small radius r on the face. As r increases, $W(r)$ is still a loop on $\mathcal{P}$, until it collapses to the farthest point from s (Figure 4.2) or it occurs a self-intersection and breaks off into multiple disjoint pieces (we call the moment a *bifurcation event*, see Figure 4.8 (b)). Here our method has a limitation of incapable of cope with a bifurcation event, and in that case, we are interested in the wavefront propagation up to that moment. We will discuss this point later.

Geometrically, $W(r)$ is made up of circular arcs. Two neighboring arcs may be joined by a *ridge point* of $W(r)$, which is a point having at least two distinct shortest paths from s. A new ridge point is created when $W(r)$ hits a vertex of $\mathcal{P}$ (we call it a *vertex event*), see Figure 4.4. The locus of ridge points of $W(r)$ for all $r > 0$ is called

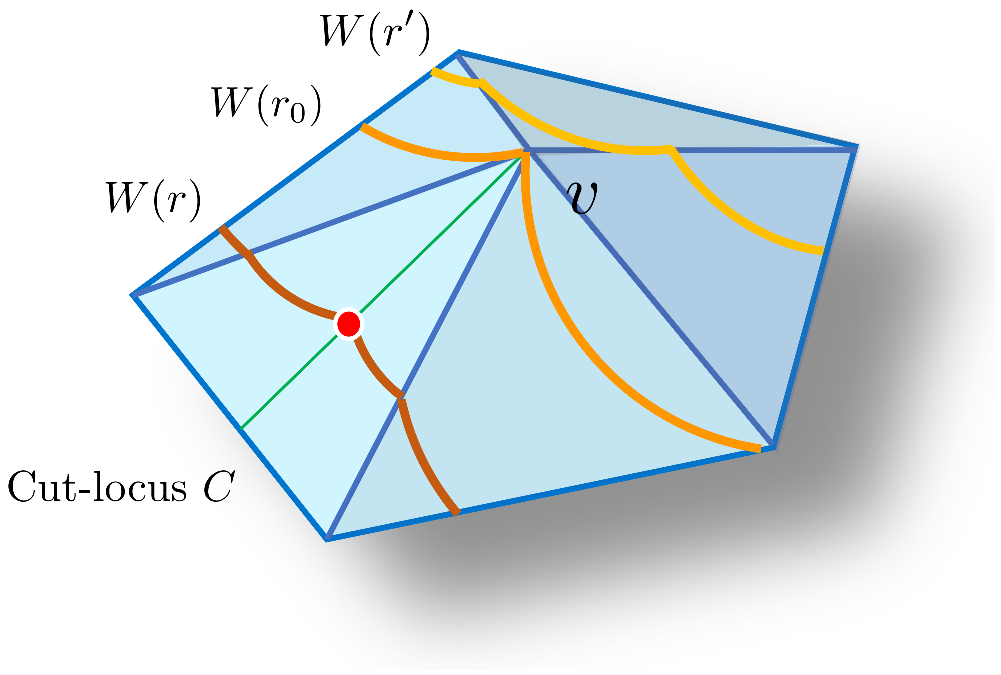


Figure 4.4: Ridge points are born at vertices and sweep out the cut-locus ($r' < r_0 < r$).

the *cut-locus* $C = C(\mathcal{P}, s)$ on $\mathcal{P}$ (here let C contain all vertices of $\mathcal{P}$, at which vertex events happen). It is a graph embedded in $\mathcal{P}$ and has a clear geometric meaning. When one cuts $\mathcal{P}$ along C by a scissor, the whole of $\mathcal{P}$ is expanded to the plane so that the obtained unfolding (net, development) is a star-shaped polygon without any overlap – every point of the unfolding can be joined with s by a line segment lying on it (Figure 4.5), which corresponds to a shortest path on $\mathcal{P}$. We call it the *source unfolding of* $\mathcal{P}$ *centered at* s.

Research in computational geometry on the cut-locus and its source unfolding has been investigated so far by several authors, e.g., [4, 8, 9, 12, 14]. Nevertheless, our approach seems to be new. Our problem is to interactively visualize the wavefront

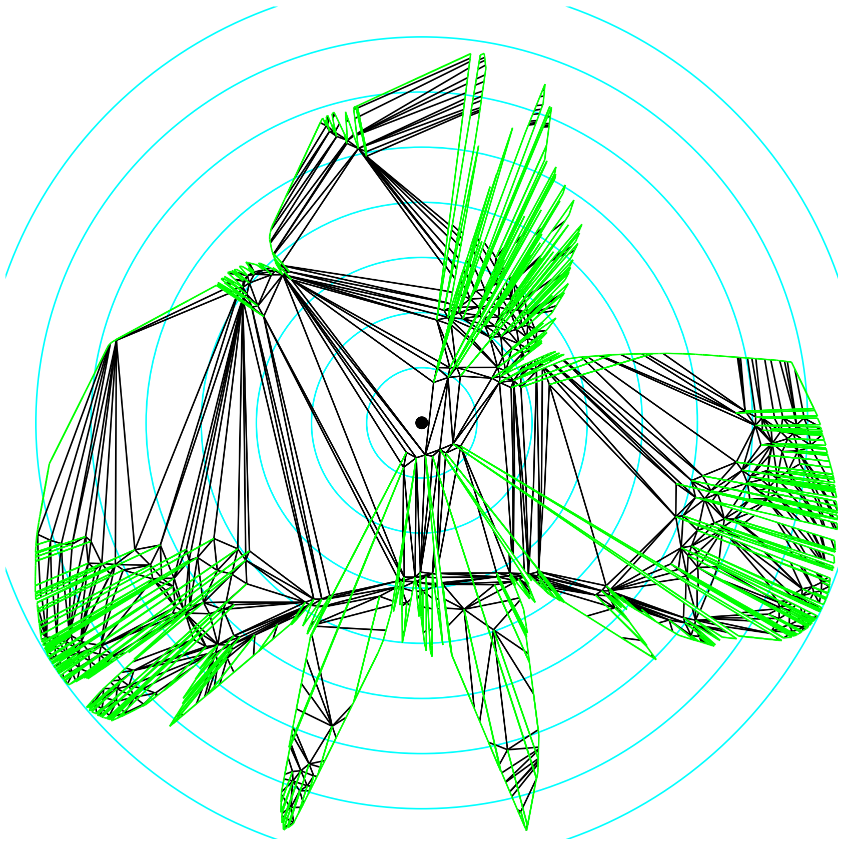


Figure 4.5: Source unfolding for the same example as in Figures 4.1 and 4.2; our program immediately produces it just after the wavefront propagation ends. Concentric circles on the unfolding represent the wavefronts on $\mathcal{P}$.

propagation $W(r)$ and compute the cut-locus C as r varies, and to finally produce the source unfolding of $\mathcal{P}$ precisely. That is designed for a practical and interactive use – for instance, in our specifications, the source point s is chosen by a click on the screen and the viewpoint for $\mathcal{P}$ can freely be rotated manually. Here is a key point that we may assume that s lies in sufficiently general position; this practical assumption enables us to classify geometric events arising in the propagation into several types

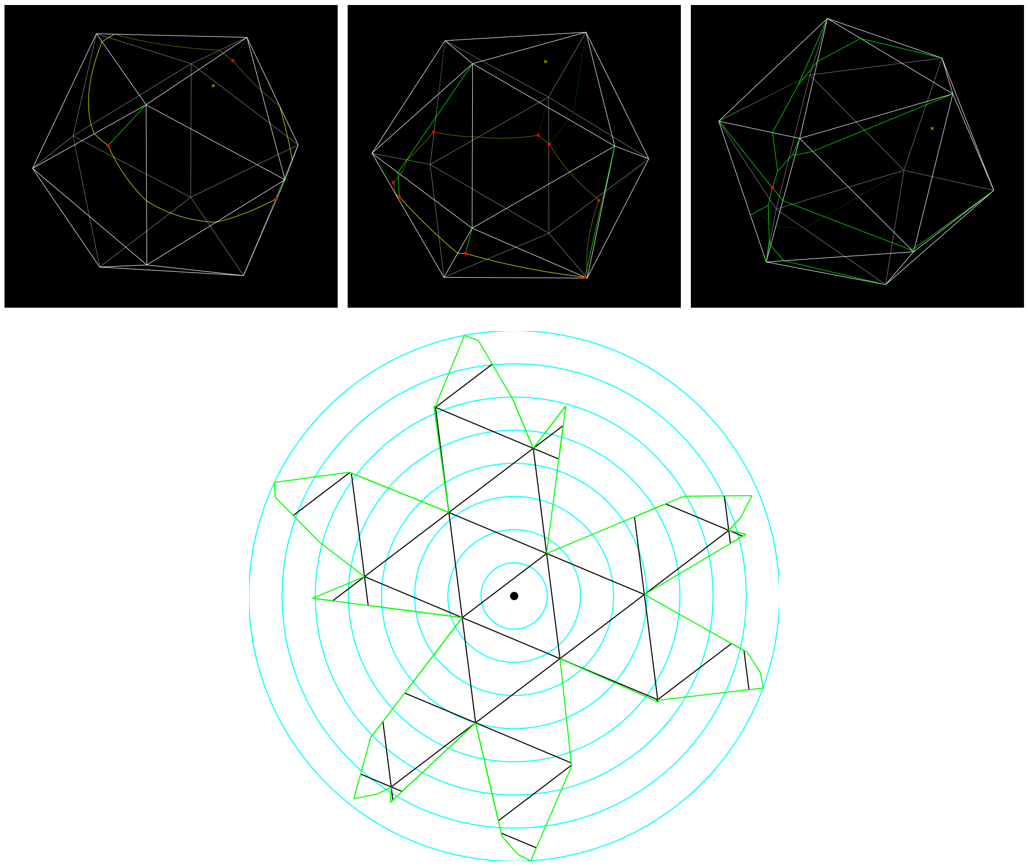


Figure 4.6: Source unfolding of an icosahedron. One can create a number of examples by choosing different source points.

(see §2), and then the algorithm becomes simple enough to be treated. Actually, our computer program certainly works, even when we choose s lying on an edge or a vertex in a visible sense (i.e., choose s within a very small distance $\varepsilon(\ll 1)$ from the edge or the vertex), see Figure 4.7.

The MMP algorithm aims to compute the shortest geodesic between two points; it receives $\mathcal{P}$ and s as inputs, and results a specialized data structure called *intervals*, which are subdivisions of all edges of $\mathcal{P}$ equipped with some additional information.

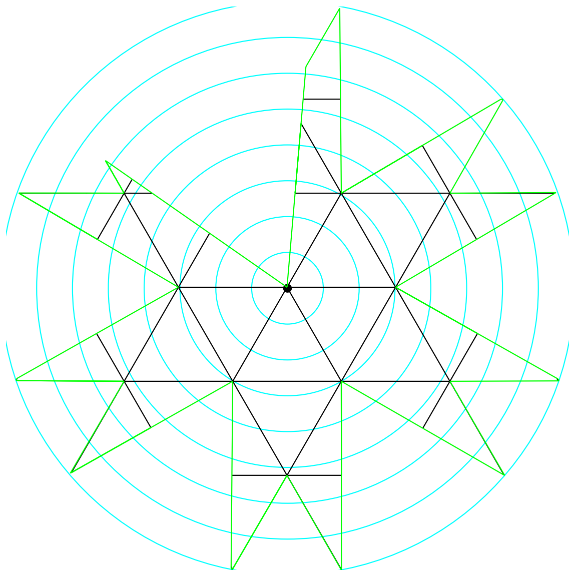
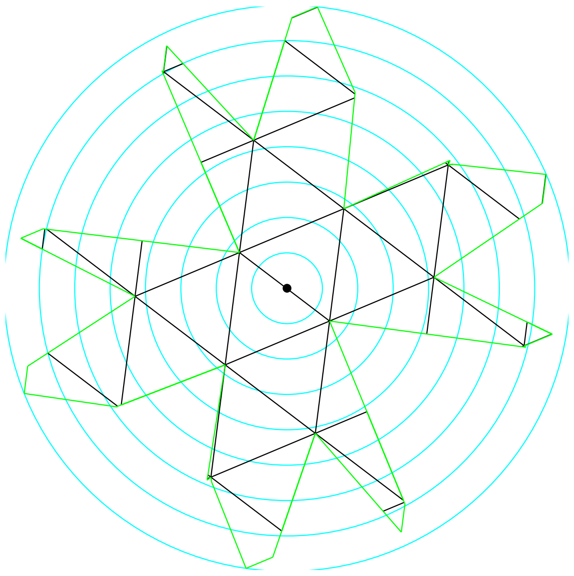



Figure 4.7: Different source unfoldings. Manually selecting the source point s on a vertex or edge contains a small invisible error. This enables our algorithm to work and properly create the cut-locus.

However, the MMP algorithm itself and other existing algorithms are insufficient for our practical purpose. We try to improve the MMP algorithm – one of our major ideas is to introduce a new data structure, called an *interval loop* indexed by the parameter r, which is a recursive sequence of *enriched intervals* $I_i \, (= I_i^{(r)})$

$$\mathbf{I}_r : I_0 - I_1 - I_2 - \cdots - I_{k(r)} - I_0.$$

Roughly speaking, $\mathbf{I}_r$ is the data structure representing the wavefront $W(r)$; each enriched interval corresponds to a circular arc participating in $W(r)$, and the sequence is closed, because $W(r)$ is assumed to be an oriented closed curve embedded in $\mathcal{P}$. Again, since our method disallows a bifurcation event, $W(r)$ is connected and one interval loop represents all of $W(r)$.

As r increases, there arise some particular moments r_0, e.g., a new ridge point is born from a vertex as depicted in Figure 4.4, a ridge point meets an edge, two or more ridge points collide on a face, and so on. We call them *geometric events* of the wavefront propagation.

Here we make a simplification to this event model. An interior point of an arc often reaches an edge before either of the surrounding two ridge points meets the edge, and then the arc is pushed out to the next face (or the next of the next, or so on), while the two ridge points stay on the original face. Since we are primarily interested in the ridge points, we do not recognize this phenomenon as an event, while the interval loop does not precisely represent the wavefront. This definition makes our algorithm much simpler, as we explain later (see Remark 4.2.2). This simplification could also improve practical performance by a constant factor, which is hidden behind the big-O notation.

Until a new event occurs at the time $r_0\,(> r)$, we simply keep the same interval loop $\mathbf{I}_r$, and for $r' \geq r_0$, the data structure is updated to $\mathbf{I}_{r'}$ by certain manipulations with the following three steps:

- Detection detects the forecast events from the data $\mathbf{I}_r$, and updates the event queue;
- Processing deletes and inserts temporarily several intervals related to that event;
- Trimming resolves overlaps of inserted intervals, and generates $\mathbf{I}_{r'}$.

The last step is similar as in the original MMP algorithm, while the first two steps contain several new ideas. Detection (re-)computes the forecast events and updates the event queue, which involves insertion, deletion and/or replacement of some events. Processing produces a provisional interval loop, which may have overlapped intervals. Trimming makes it a valid interval loop in a true sense. Our algorithm ends when $W(r)$ collapses to the farthest point or is found to be inconsistent (which occurs at some moment after an occurrence of a bifurcation event or a non-generic event). Then we obtain the cut-locus C and the distance from s to every vertex passed though. As for the computational complexity, our algorithm takes $O(n^2 \log n)$ time and $O(n)$ space, where n is the number of vertices of $\mathcal{P}$, and it can be modified to be able to find the shortest geodesic with $O(n^2)$ space (see subsection 4.4.1).

A particular feature of our algorithm is that, unlike the MMP algorithm, we can visualize the ongoing wavefront propagation, i.e., we compute the set $W(r)$ of points having shortest paths of length r from s *all at once*, as well as partially-constructed cut-locus during execution. As a remark, in [14] Mount describes how to find the cut-locus C by the information of obtained intervals – for each face σ one can detect $C \cap \sigma$ by computing an associated *Voronoi diagram*. However, it requires a bit heavy new task additionally to the MMP and it seems not quite obvious how to implement it to computer program which actually works. In contrast, our algorithm instantly produces the complete information of the cut-locus C.

In general, bifurcation events may occur, and then $W(r)$ break off into several connected components. This phenomenon is the most difficult obstacle for tracing the wavefront propagation beyond the MMP algorithm. In fact, our algorithm is designed to depend only on the *local data* (data of neighboring arcs participating in the wavefront), not *global data* of the wavefront, and therefore, our implemented program may stop at a certain moment after some bifurcation event actually happens.

4.2 Preliminaries

Throughout this chapter, let $\mathcal{P}$ be a convex simplicial polyhedron such that every vertex v of $\mathcal{P}$ is spherical, i.e., the sum of angles around v (measured along the faces) is less than 2π [1, 8] (see Section 2.1). We assume that every face of $\mathcal{P}$ is an oriented triangle so that the orientation is anti-clockwise when one sees the 3D body from outside.

Let us assume two points s and t on $\mathcal{P}$ are given. Among all paths on the surface connecting s and t, we can consider a shortest one (there may be multiple shortest paths connecting s and t). The length of a shortest path connecting s and t defines the *distance* $d(s, t)$ on $\mathcal{P}$. A shortest path γ is a geodesic, which is defined by Definition 2.1.3 [12]. Since every vertex is spherical, γ satisfies the following properties:

1. γ is straight inside any face, and where γ passes through an edge sequence $\mathcal{E} = (e_1, \ldots, e_k)$, γ is straight on the unfolding obtained from $\mathcal{E}$;

2. γ never passes through any vertex.

Given a convex polyhedron $\mathcal{P}$ and a point $s \in \mathcal{P}$, the wavefront $W(r)\,(= W(r, s))$ for $r > 0$ and the cut-locus $C(\mathcal{P}, s)$ are defined as in Section 4.1. As r increases, the geometric shape of the wavefront changes.

Definition 4.2.1. **(Geometric events)** We define several *events* of the wavefront propagation at $r = r_0$ as follows:

(v) a *vertex event* occurs when $W(r_0)$ hits a vertex of $\mathcal{P}$;

(e) an *edge event* occurs when a ridge point of $W(r_0)$ hits an edge;

(c) a *collision event* occurs when multiple ridge points of $W(r)$ $(r < r_0)$ collide at once, and result in a single ridge point of $W(r')$ $(r_0 \leq r')$, or (a component of) the wavefront converges at the point and disappears.

(b) a *bifurcation event* occurs when $W(r_0)$ intersects itself and breaks off into several pieces for $r > r_0$.

We divide edge events (e) into the following two patterns. Let A and B be neighboring arcs in $W(r_0)$ joined by the ridge point a which hits an edge e. Unfold the two faces incident to e, and divide the plane by the line containing e. We set

(ec) a *cross event*: if the centers of A and B are located in the same half-plane;

(es) a *swap event*: if the centers of A and B are located in opposite half-planes.

Furthermore, among collision events, we distinguish the following special one:

(cf) the *final event* occurs when the wavefront reaches the farthest point and disappears.

Remark 4.2.2. At some moment, the wavefront can be tangent to an edge at some point and go through to partially propagate to the next face. We do not include this case into the above list of geometric events, as the arc simply expands on the unfolding along the edge. In other words, our data structure does not need to be changed. As seen later (§4.3.6), it makes our algorithm much simpler, while our instantaneous visualization of the wavefronts does not depict this partial propagation precisely (but it does not affect the calculation of the cut-locus). By this definition, we can ensure that every interval appears exactly *once* in the interval loop, and every interval with non-empty true extent (see §4.3.3) is involved in exactly *two* (vertex, edge or swap) events, where it is propagated in the first one and removed in the second one. Otherwise, it requires special treatment of intervals which appear *twice* in the interval loop, which also have one or more descendants. Also, they would be involved in *three* events, where it is propagated in the first one and removed in the second and third ones, whereas some intervals are still involved in only two events. See also Remark 4.3.2.

Definition 4.2.3. **(Generic source point)**

(i) We say that the source point s is *generic* if the following three properties hold:

   (1) s is an interior point of a face,

   (2) every ridge point of $W(r)$ for any $r > 0$ does not pass through vertices and does not move along an edge,

   (3) every collision (including the final) event happens in the interior of a face, and the number of ridge points collide at once is three in the final event and two otherwise.

(ii) When choosing s to be generic, the wavefront propagation admits only geometric events as depicted in Figure 4.8; we call them *generic geometric events*.

In this chapter, as mentioned in Section 4.1, we consider the wavefront propagation with a generic source s for the period until the final event or a bifurcation event happens.

The cut-locus $C = C(\mathcal{P}, s)$ is a graph embedded on $\mathcal{P}$ whose edges are linear segments. If there happens a vertex event at a vertex v with $r = r_0$, the propagation around v creates the cut-locus, i.e., $W(r_0 - \varepsilon)$ for small $\varepsilon > 0$ on the unfolding around v is locally one circular arc, while $W(r_0 + \varepsilon)$ has one ridge point locally (Figure 4.4 in Section 4.1). Then v is an end of the cut-locus C. Therefore, we see that

Lemma 4.2.4. If the source s is generic and the bifurcation event does not appear during the wavefront propagation, then the obtained cut-locus C is connected and has a tree structure with leaves at vertices of $\mathcal{P}$ and nodes with degree 3, which are points at which collision events and the final event happen.

Remark 4.2.5. If the source point s is generic, the shape of the cut-locus $C(\mathcal{P}, s)$ is stable with respect to small perturbations of s. Namely, for sufficiently near generic s',

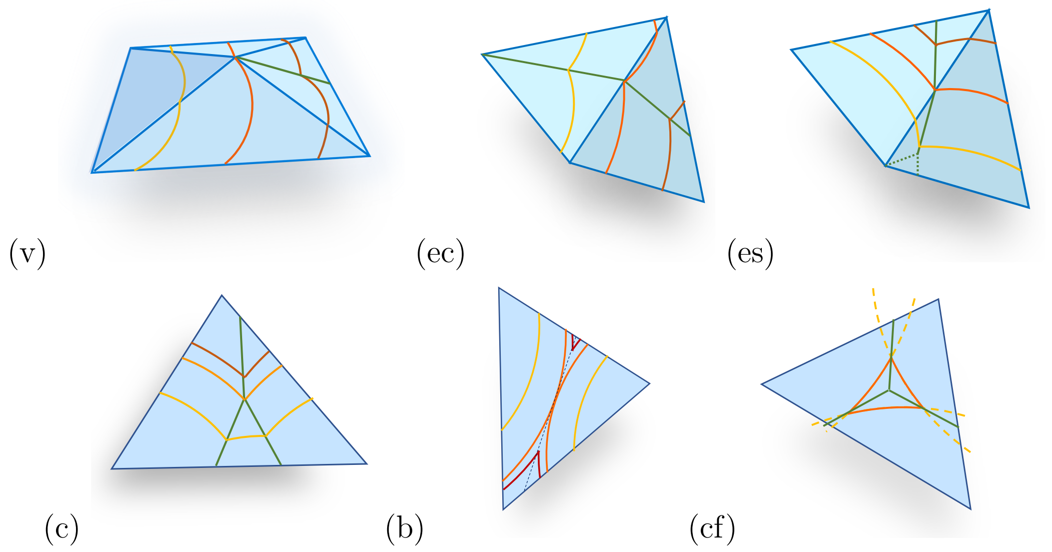


Figure 4.8: Generic geometric events: the wavefront propagates from the yellow one to the (dark) red one. (v) *Vertex event*: a new ridge point is created at a vertex, and no ridge point tends to a vertex as r increases; (ec), (es) *Cross/Swap edge event*: a ridge point hits an edge (it does not move along the edge); there are two types – two arcs come across the edge from the same side or from the opposite side; (c) *Collision event*: only two ridge points collide at once inside a face; (b) *Bifurcation event*: two local components of the wavefront get to be tangent to each other, and breaks off into two pieces; (cf) *Final event*: the triangle-shaped wavefront goes to shrink and disappears.

$C(\mathcal{P}, s)$ and $C(\mathcal{P}, s')$ are the same graph so that corresponding two edges have almost the same length. Now suppose that the source point s is not generic. Even though the cut-locus $C(\mathcal{P}, s)$ exists but possesses some degenerate vertices or edges. When perturbing s to a generic s', such degenerate points locally break into generic geometric events as indicated in Figure 4.8, and the new cut locus $C(\mathcal{P}, s')$ should be sufficiently close to $C(\mathcal{P}, s)$.

Remark 4.2.6. Theoretically, it is possible to determine whether a chosen point s is generic or not, if we have an unfolding of the whole of $\mathcal{P}$ in advance. In our specifications, however, we are not supposed to have such prior information; rather to say, as mentioned before, we are aiming to produce a nice planar unfolding. In practice, non-generic geometric events do not occur unless we intentionally set up such input of $\mathcal{P}$ and s.

Remark 4.2.7. In the contexts of differential geometry and singularity theory, wavefronts, caustics, cut-loci and ridge points on a smooth surface have been well investigated, see e.g., Arnol'd [2]. Our classification of generic geometric events is motivated as a sort of corresponding discrete analog.

4.3 Main algorithm

4.3.1 MMP algorithm

The MMP algorithm [12] (and Mount's earlier algorithm [14]) encodes geodesics as the data structure named by *intervals*:

- Input: a polyhedron $\mathcal{P}$ and a source point s on $\mathcal{P}$.
- Output: a set of intervals for each edge, which enables us to find the shortest geodesic from s to any given point t on $\mathcal{P}$.
- Complexity: $O(n^2 \log n)$ time, $O(n^2)$ space, where n is the number of edges of $\mathcal{P}$.

An interval I is a segment, called the *extent* of I, in an edge e of $\mathcal{P}$ endowed with additional data being necessary to find the shortest path from s to points of the extent. Intervals are inductively propagated – each interval generates a new one (its child interval) step-by-step by manipulations called *projection* and *trimming.* The algorithm uses a priority queue to manage the order of the propagation of the intervals. The priority of an interval is the *shortest distance* between the source point s and its extent, and any smaller value means to be propagated earlier.

4.3.2 Our problem

Our main problem is to reveal some richer structure of geodesics on $\mathcal{P}$ by describing the wavefront propagation interactively and accurately, where we deal with not only a single geodesic from a source point s but also all geodesics from s at once. At the final moment, we obtain the entire cut-locus $C(\mathcal{P}, s)$ and the source unfolding, provided that the bifurcation event does not occur in the whole process; otherwise, our algorithm stops at some moment after that bifurcation event.

Our algorithm runs in the following time and space complexity:

- Input: a convex polyhedron $\mathcal{P}$ and a point $s \in \mathcal{P}$.
- Output: the cut locus $C(\mathcal{P}, s)$.
- Complexity: $O(n^2 \log n)$ time, $O(n)$ space.

Furthermore, as an option, our algorithm can also support shortest path query using extra space complexity;

- Input: a convex polyhedron $\mathcal{P}$ and a point $s \in \mathcal{P}$.
- Output: the cut locus $C(\mathcal{P}, s)$ and a set of intervals for each face, to be able to find shortest geodesic from s to any given point t on $\mathcal{P}$.
- Complexity: $O(n^2 \log n)$ time, $O(n^2)$ space.

- Input of query: a point t on $\mathcal{P}$.
- Output of query: the shortest path(s) from s to t.

4.3.3 Data structure

The wavefront $W(r)$ is an oriented closed embedded curve on $\mathcal{P}$ consisting of *circular arcs* on faces. For each circular arc A, we introduce the notion of an *enriched interval* $I\,(= I_A)$ as a data structure to express the arc A equipped with some additional data.

Definition 4.3.1. We define an *enriched interval* I as a data structure shown in Table 1. Each item is denoted by $I.[--]$ for notational convention.

I.Face	the oriented face σ which contains the arc A
I.Center	the center p_A of the arc A on the plane H_σ containing σ
I.Edge	the oriented edge e of σ into which A is projected from p_A
I.Extent	the (foreseen) extent e_A associated with A
I.Prev	the enriched interval associated with the previous arc connecting to A
I.Next	the enriched interval associated with the next arc connecting from A
I.Ridge	the ridge point to which A is adjacent as the start point
I.Parent	the enriched interval which generates I

Table 4.1: An enriched interval I

In the previous chapter, we had the I.Depth member in Definition 3.2.1. However, since we are only considering a convex polyhedron in this chapter, we do not need I.Depth to be explicitly stored here (it is always zero).

We also define an *interval loop*

$$\mathbf{I}_r = \left\{ I_0^{(r)}, I_1^{(r)}, \cdots, I_{k(r)}^{(r)} \right\}$$

to be a finite sequence of enriched intervals that satisfy

$$I_i^{(r)}.\text{Prev} = I_{i-1}^{(r)} \text{ and } I_i^{(r)}.\text{Next} = I_{i+1}^{(r)}$$

for $0 \leq i \leq k(r)$, where we put $I_{k(r)+1}^{(r)} := I_0^{(r)}$ and $I_{-1}^{(r)} := I_{k(r)}^{(r)}$.

An enriched interval is similar but different from the notion of an *interval* used in Mount's algorithm [14] and the MMP [12]. Main differences are, e.g.,

- all enriched interval in the wavefront make up an interval loop;
- an interval loop is a circular doubly-linked list: each enriched interval has the previous and the next interval corresponding to adjacency of arcs and orientation of the wavefront;
- our enriched interval depends on r;
- an enriched interval may have the *empty* extent with non-trivial additional data.

Each item in Table 4.1 in Definition 4.3.1 depends on r; those are created at the time when the corresponding arc A is born, and are valid until A disappears. In particular,

- the data in I.Face, I.Center, $I.Edge$ and I.Parent are fixed when A is born;
- the data in I.Extent, I.Prev, I.Next and I.Ridge are updated at every moment where some geometric event involving A happens.

Below we explain the meaning of each item in Table 4.1.

An arc

To begin, let $r_1 > 0$ be fixed. Suppose that a circular arc A participating in $W(r_1)$ lies on a face (oriented triangle) σ of $\mathcal{P}$. Let H_σ denote the affine plane containing σ in $\mathbb{R}^3$, then there is a unique point $p_A \in H_\sigma$ such that A is an arc in the circle on H_σ centered at p_A with radius r_1. Take a point $t \in A$ and the shortest path γ on $\mathcal{P}$ from s to t. We find an unfolding of $\mathcal{P}$ expanded on H_σ, on which γ is represented by a line segment, as shown in Figure 4.9. The 3D coordinates of the point p_A is explicitly obtained from the 3D coordinates of $s \in \mathcal{P}$ by inductively operating certain rotations of $\mathbb{R}^3$ along edges which γ intersects.

Basic data structure for an arc

Suppose that p_A is outside σ, and only one edge of σ, say e_0, cuts any segments between p_A and points of A. The rays from p_A to the arc A meet another edge of σ in opposite side of e_0 with respect to the location of A.

1. Suppose that $I = I_A$ is projected from the center p_A to only one edge e and yields I_1 (see the left picture of Figure 4.10). The direction of e is chosen to be compatible with the orientation of A. We first define the *true extent* associated with A by the subrange $e_A \subset e$ which A will actually pass through, see Figure 4.11. It can be the empty set (see the right picture of Figure 4.11). Notice that the true extent e_A is fixed after all the events involving A have occurred. Therefore, in the middle of the process, what we can do is only to provisionally find a *foreseen extent*, which we denote by $\tilde{e}_A$, and update it just after the next event happens (indeed, this procedure is the heart of our algorithm, which will

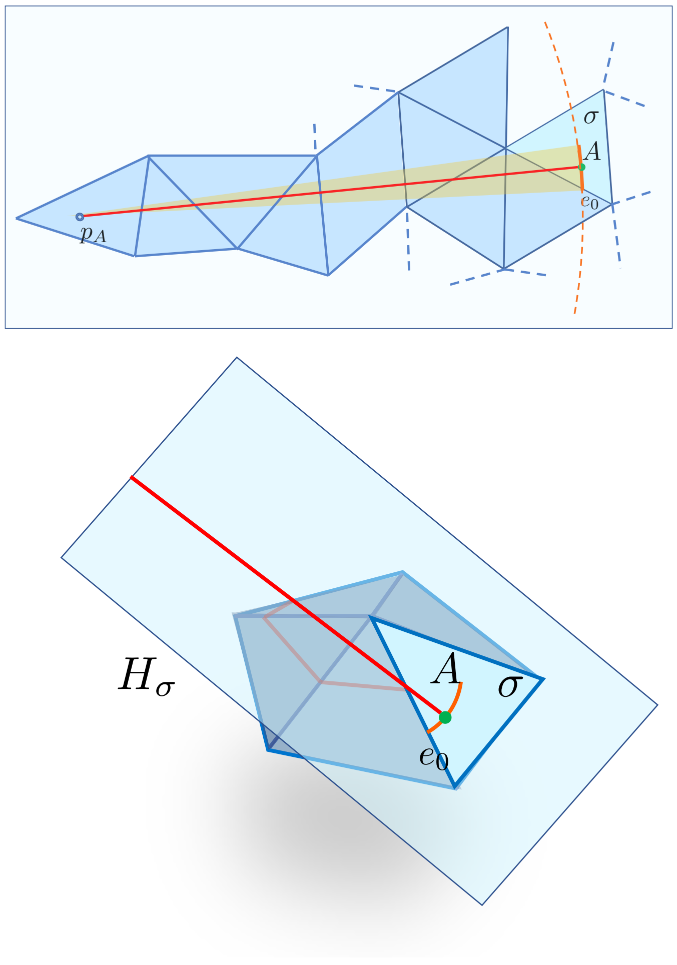


Figure 4.9: A circular arc on a face σ (the upper figure depicts an unfolding on the plane H_σ).

be described in detail later in the following sections). For now, we put

$$I = I_A := (\sigma, e, p_A, \tilde{e}_A),$$

where each is referred to as

$$I.\text{Face} = \sigma, \quad I.\text{Edge} = e, \quad I.\text{Center} = p_A, \quad I.\text{Extent} = \tilde{e}_A$$

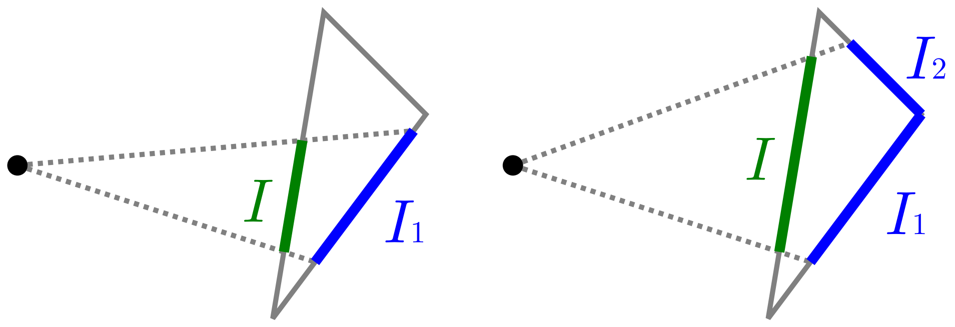


Figure 4.10: Interval I is projected from I.Center to opposite edges.

and we will append some additional data to this data structure and use the same notation I or I_A; we call it an *enriched interval* or simply *interval*.

2. Suppose that $I = I_A$ is projected from p_A to two edges e_1 and e_2 whose order is determined by the orientation of σ and yields I_1 and I_2 (see the right picture of Figure 4.10). Then A is divided into two pieces, say A_1, A_2, projected into e_1, e_2, respectively. If each sub-arc A_i has the non-empty foreseen extent, $\tilde{e}_{A_i} \neq \emptyset \subset e_i$ $(i = 1, 2)$, then we associate to A an ordered pair of two enriched intervals

$$I_1 - I_2 \quad \text{with} \;\; I_i = I_{A_i} := (\sigma, e_i, p_A, \tilde{e}_{A_i}) \quad (i = 1, 2).$$

We say that I_1 and I_2 are twins.

3. Suppose that the source point s is an interior point of σ with edges e_0, e_1, e_2 (anti-clockwise). We then define the *initial loop* by a triple of enriched intervals

$$I_0 - I_1 - I_2 - I_0 \quad \text{with} \;\; I_i := (\sigma, e_i, s, e_i) \quad (i = 0, 1, 2).$$

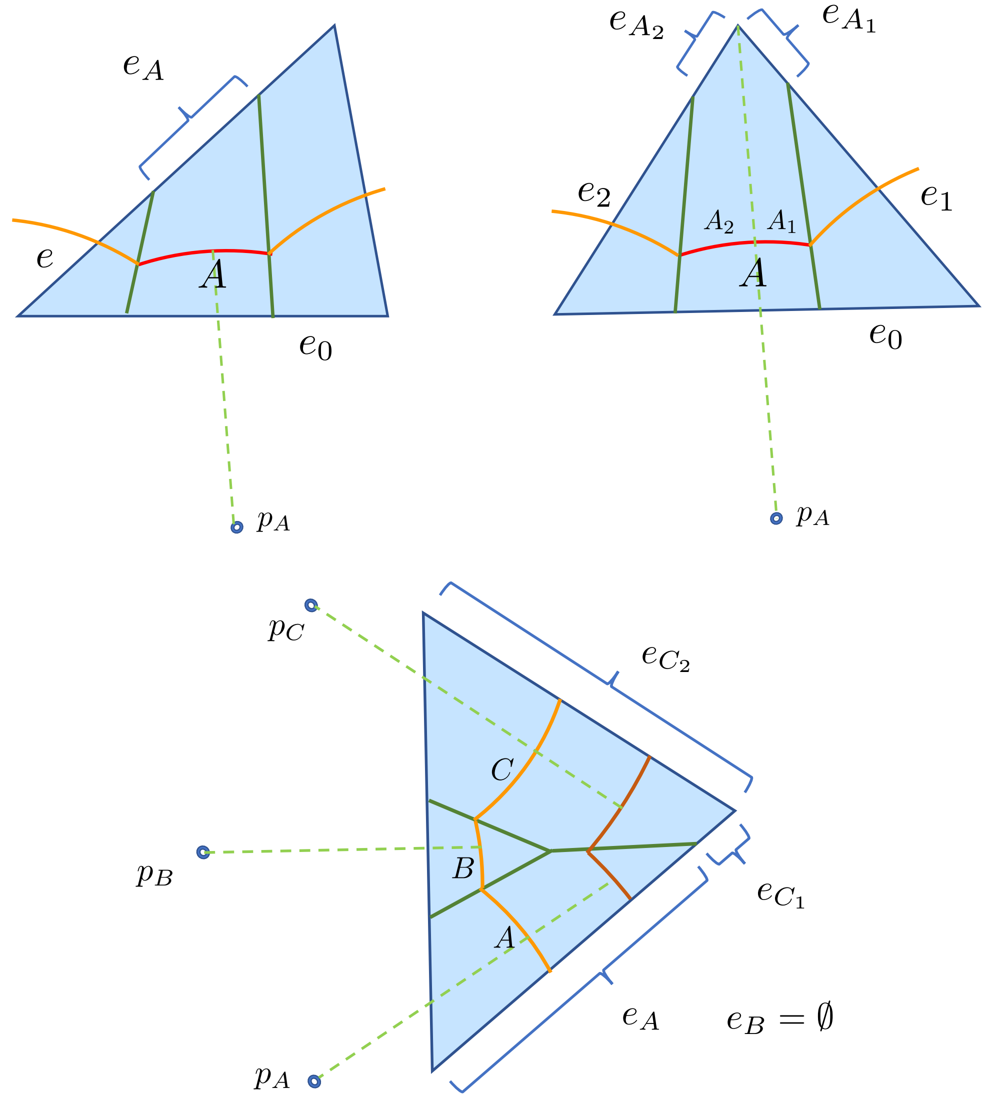


Figure 4.11: True extents

Parent of an arc

Let I_A be an enriched interval associated with an arc A (or one of twins) in a face σ. If the true extent is non-empty, then the arc A will pass through the extent and go into the next face τ sharing the edge with σ. Let A' be the new resulting arc in τ, and $I_{A'}$ the corresponding interval for A' (or $I_{A'_1} - I_{A'_2}$ if A' has two associated enriched intervals). We say that I_A is propagated to $I_{A'}$, and also call I_A its *parent*:

$$I_{A'}.\text{Parent} := I_A, \qquad (\text{or } \ I_{A'_i}.\text{Parent} := I_A \ \ (i = 1, 2)).$$

Throughout this chapter, $A', B', B'', \cdots$ mean the resulting arcs to which their parents $A, B, B', \cdots$, respectively, are propagated.

This item is used as follows. For instance, if an enriched interval I satisfies

$$I.\text{Prev} = I.\text{Parent},$$

then we understand that the parent is non-empty, say A, and the arc represented by I is nothing but the resulting arc A' to which A is propagated. Here, I.Ridge must be empty, for the endpoint of A' is not a ridge point. If neighboring intervals $I - J$ have the same parents

$$I.\text{Parent} = J.\text{Parent},$$

then they are twins (i.e., $I = I_1$ and $J = I_2$).

Previous/next arcs and ridge points

An arc A connects with two other arcs in the same wavefront; according to the orientation of the wavefront, let B be the previous arc and C the next one. Then, for their intervals, we write $-I_B - I_A - I_C-$ and set

$$I_A.\text{Prev} := I_B, \ \ I_A.\text{Next} := I_C.$$

Here B, A or A, C can be a twin. Let a and c be the joint point $B \cap A$ and $A \cap C$, respectively. We make a convention that any information of a (resp. c) will be appended to and stored in I_A (resp. I_C). For instance, if a is a ridge point, we set

$$I_A.\text{Ridge} = a.$$

If not, this item is *Nil*. When some events happen, items I.Prev, I.Next and I.Ridge may be updated.

We recall that there are two patterns of edge events as noted in §2; let $I_A - I_B$ be neighboring intervals where arcs A and B are joined by a ridge point $a = I_B$.Ridge which hits an edge. Then the two patterns can easily be distinguished as follows:

- *Cross event*: both extents of I_A and I_B are non-empty;
- *Swap event*: one of the extents of I_A or I_B is empty.

Furthermore, we call a swap event to be of type CW (resp. CCW) if I_A.Extent (resp. I_B.Extent) is empty (clockwise/counter-clockwise). Note that by definition, at least one extent is non-empty.

4.3.4 Our algorithm

A basic idea is to express the wavefront propagation $W(r)$ by updating interval loops step-by-step

$$\mathbf{I}_{r_0} \Longrightarrow \mathbf{I}_{r_1} \Longrightarrow \mathbf{I}_{r_2} \Longrightarrow \cdots \quad (0 = r_0 < r_1 < r_2 < \cdots).$$

1. **Initial loop**: Suppose that s is in the interior of σ. Then $\mathbf{I}_0$ is given by $I_0 - I_1 - I_2 - I_0$ of directed edges of σ.

2. **Events**: Generic geometric events in §2 are interpreted as change of the data structure of enriched intervals, named *events*.

   In our algorithm, we append two more items to the data structure of each I (Table 4.2). Since an interval I has at most one associated forecast event, we store it as I.Forecast if it exists. Also we use another new item I.IsPropagated to ask whether the arc has been propagated or not (see §4.3.6). The default of I.Forecast is *Nil* (indicating it does not exist), and that of I.IsPropagated is *No*.

I.Forecast	the forecast event for I if exists; otherwise, *Nil*
I.IsPropagated	Yes/No for the inquiry whether or not I has been propagated

Table 4.2: Additional items in an interval I.

3. **Manipulation**: Each $\mathbf{I}_{r_j}$ is updated to $\mathbf{I}_{r_{j+1}}$ at an event – some intervals in $\mathbf{I}_{r_j}$ are removed, some new intervals are inserted, and items of remaining intervals are updated. Each event activates three editing processes named by *Detection*, *Processing* and *Trimming*. The algorithm halts when the final event occurs and the wavefront disappears, or when some exception arises, e.g., a bifurcation event or non-generic event happens.

Pseudocode of our algorithm is described in Algorithm 11. Each process will be described below.

4.3.5 Event Detection

Suppose that we have an interval loop $\mathbf{I}_{r_j}$ which represents the wavefront $W(r_j)$. Let $I = I_A$, I_{A_1} or I_{A_2} be an enriched interval belonging to it, associated with an arc A or

Algorithm 11 (Main algorithm)

```
Create the initial interval loop
while the wavefront exists do
    for each interval $I$ whose adjacency changed in the previous step do
        Trim temporary extents of $I$ and remove redundant intervals
        Detect the forecast and update the event queue
    end for
    Process the top-most event in the event queue
end while
```

twin sub-arcs on a face $\sigma = I_A$.Face.

Detecting the earliest event for an arc

We first detect which geometric event for A will happen *without any consideration on the propagation of other arcs.*

- *Vertex event*

  By the existence of a pair of twins I_{A_1} and I_{A_2}, we foresee that a vertex event will happen at the vertex they meet. We store the event in I_{A_2}.

- *Collision event*

  In case that consecutive I_A, I_B and I_C have the same face and I_B.Extent is empty, we foresee that the arc B will collapse and a collision event will happen at the equidistant point (circumcenter) from three points, I_A.Center, I_B.Center and I_C.Center. We store the event in I_B.

- *Cross event*

  In case that consecutive I_A and I_B have the same edge and both extents are not empty, we foresee that the ridge point between them hits the edge and a cross event will happen at the point I_A.Extent and I_B.Extent meet. We store the event in I_B.

- *Swap event*

  In case that an interval I has the empty extent and the next or previous interval is its parent, we foresee a swap event will happen. There are two types of swap events, CW and CCW, depending on its parent is the previous or next of I. We store the event in I.

For each event, the predicted time can exactly be calculated from the point at which the event will happen.

Priority queue

Since the earliest forecast event cannot be modified by other events and occurs next for certain, we use a priority queue to schedule all forecast events by their predicted times and choose the earliest one to be processed; we call it the *event queue*.

A pseudocode for Detection of the forecast for each enriched interval of the loop is described in Algorithm 12. Referring to this queue, we can find the earliest event among the forecast events of all enriched intervals belonging to $\mathbf{I}_{r_j}$.

4.3.6 Event Processing

As the result of Detection process, now we have the earliest forecast event of the loop $\mathbf{I}_{r_j}$. First we delete several intervals of $\mathbf{I}_{r_j}$ involved in that event, and then create and insert new intervals, and make some changes of items in remaining intervals of $\mathbf{I}_{r_j}$.

Algorithm 12 (Detection at each enriched interval I)

```
if $I$.Extent $= \emptyset$ then
    if $I$.Prev.Face $=$ $I$.Face $=$ $I$.Next.Face then
        A collision event will occur; calculate the predicted time
    end if
    if $I$.Prev $=$ $I$.Parent and $I$.Next.Extent $\neq \emptyset$ then
        A CW swap event will occur; calculate the predicted time
    end if
    if $I$.Next $=$ $I$.Parent and $I$.Prev.Extent $\neq \emptyset$ then
        A CCW swap event will occur; calculate the predicted time
    end if
else (i.e. $I$.Extent $\neq \emptyset$)
    if $I$ and $I$.Prev are twins then
        A vertex event will occur; calculate the predicted time
    end if
    if $I$.Edge $=$ $I$.Prev.Edge then
        A cross event will occur; calculate the predicted time
    end if
end if
```

Below we describe Processing for each type of events in typical situations (in fact, it is often to need to consider several divided cases, but we avoid a messy description here).

Recognition of propagated arcs

If an arc A has a non-empty true extent, $e_A \neq \emptyset$, then let ξ_A be the one closer to A of the two endpoints of e_A (if both endpoints have equal distance, take one of them).

Afterwards, just when A arrives at the point ξ_A, some event (vertex, cross/swap) happens at that point and A starts to propagate to the next face. At this moment, we update the item I_A.IsPropagated to be Yes (from No, the default) and create a new interval $I_{A'}$ representing the resulting arc A' and insert it to the interval loop. Here, of course, it can happen that A starts to propagate to multiple faces, and it creates multiple new intervals, say, $I_{A'}, I_{A''}, \cdots$. Afterwards, the arc A arrives at the other endpoint of e_A and some other event happens at that point. At this moment we recognize that all the propagation of A has been done; namely, we remove I_A from the interval loop. We remark again that we do not take any attention to a 'partial propagation' caused by tangency of A with some edge (Remark 4.2.2). That makes the algorithm much simpler.

Temporary extents

In Processing at an event, each newly-created interval I_A may be assigned incomplete data at some items. For instance, if an arc A is resulted by propagating an arc (= the parent of A), the data structure I_A is created at that moment and the item I_A.Extent is temporarily filled in by the image of its parent's extent I_A.Parent.Extent via the projection from the center $p_A = I_A$.Center (Figure 4.12). Such a temporary extent for A may have overlaps with extents of previous/next intervals. The next process Trimming corrects the overlaps and produces a foreseen extent $\tilde{e}_A$ (§4.3.7). Afterwards, at every event which is related to A, I_A.Extent is updated by new $\tilde{e}_A$, and finally, if no more related event occurs, then the latest $\tilde{e}_A$ means the true extent e_A. In Figure 4.13, we explain a consecutive process updating the extents; the detail of manipulation at each event will be described below.

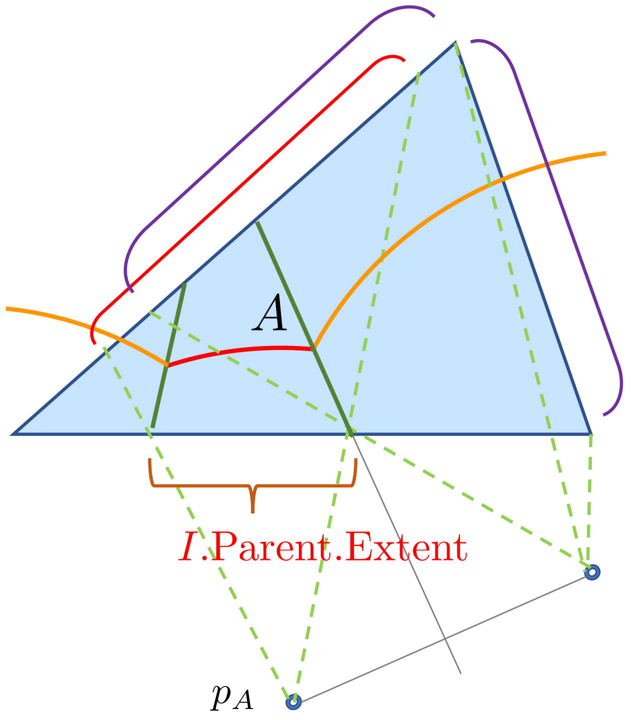


Figure 4.12: Temporary extents in Processing; overlaps are resolved in Trimming

Cross event

Consider the cross event such that both neighboring arcs joined by the ridge point, say A and B in order, have *non-empty true extents* just before the event happens. The edge (blue) is locally divided into the two extents. Each of arcs A and B has two patterns according to whether it has already been propagated or not yet; this information (Yes or No) is stored in I.IsPropagated. There are four patterns, see Figure 4.14. In the first and second ones, we simply remove the interval I_B (resp. I_A) from the interval loop, and instead, suitably insert a new interval $I_{A'}$ (resp. $I_{B'}$) to produce the new

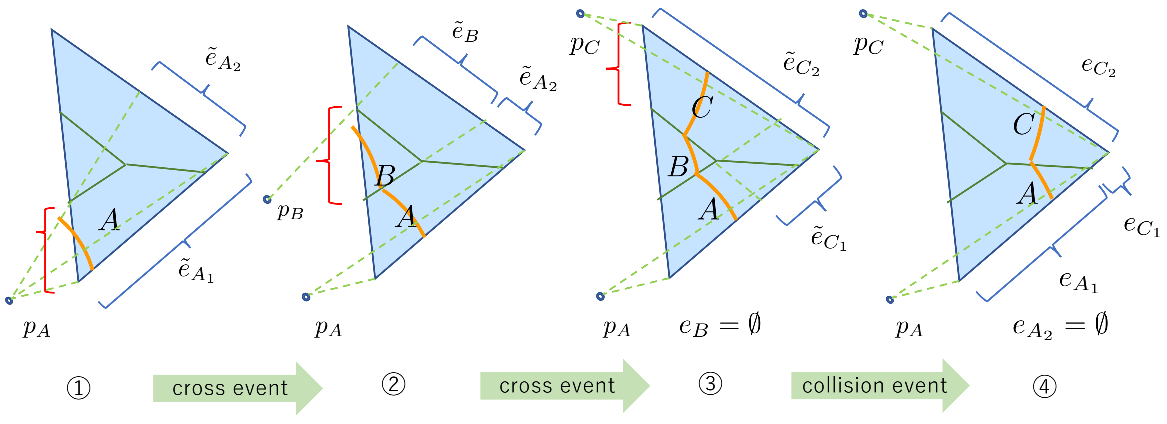


Figure 4.13: An example of updating I.Extent at consecutive several events. (1) An arc A is born on this face (propagated from its parent) and temporary extents for a twin is made; those do not need to be edited yet. (2) A cross event has just created a new arc B; Processing puts a temporary extent in I_B.Extent, but soon after, Trimming updates I_{A_2}.Extent and I_B.Extent by resolving overlaps. (3) Another cross event has created the third arc C; after Trimming, it turns out that I_B.Extent $= \emptyset$. Since A and C are not neighboring, foreseen extents of A_i and C_i $(i = 1, 2)$ may have overlaps, and do not need to be edited yet. (4) The next is a collision event; Processing deletes I_B and Trimming edits all extents and deletes I_{A_2} (resolve the redundant twin). There remains $-I_{A_1} - I_{C_1} - I_{C_2}-$ in the interval loop. Finally, I_{A_1} will soon be deleted at the coming cross event.

loop, e.g., in case of (No-Yes), we do the manipulation

$$-I_A - I_B - I_{B'}- \quad \Longrightarrow \quad -I_A - \boxed{I_{A'}} - I_{B'}-$$

For the third pattern, just before the cross event happens, both arcs A and B have not yet arrived at ξ_A and ξ_B, respectively, thus both I.IsPropagated are still being 'No'. If both arcs have already been propagated, that is the fourth one.

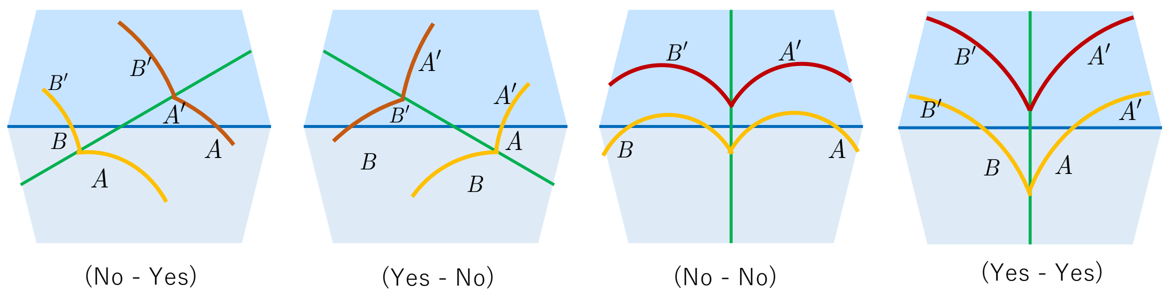


Figure 4.14: There are four patterns for cross events where the cut-locus (green) intersects with an edge (blue), referring to (I_A.IsPropagated - I_B.IsPropagated).

Swap event

At a swap event, let A, B be the neighboring arcs joined by a ridge point which hits an edge (Figure 4.15). Then, one of them has already been propagated, say it A; just before the event, A' is joined with B and the extent of A' is empty. Just after the event, A' disappears and an arc B' newly arises. Namely, arcs A' and B' are swapped. The manipulation on enriched intervals is as follows:

$$-I_A - I_{A'} - I_B- \quad \Longrightarrow \quad -I_A - \boxed{I_{B'}} - I_B - \quad (CW)$$

$$-I_A - I_{B'} - I_B- \quad \Longrightarrow \quad -I_A - \boxed{I_{A'}} - I_B - \quad (CCW)$$

Vertex event

Suppose that an arc A in σ meets a vertex v of σ and the cut-locus is created in another face τ. The arc A is represented by twin arcs, say A_1, A_2, whose extents are joined at v. For example, look at the left picture of Figure 4.16 which is an unfolding around v on the plane H_τ. Just before the event happens, the twins have already been propagated, and moreover A_2' has also been propagated, so intervals $I_{A_1'}$, $I_{A_2'}$ and $I_{A_2''}$ exist. Then

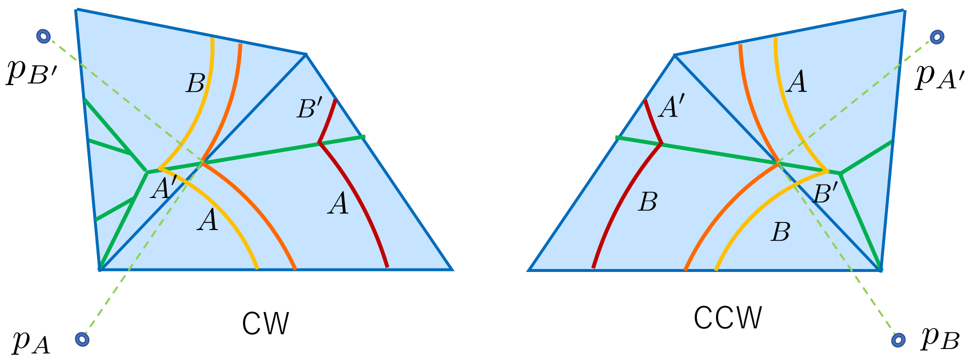


Figure 4.15: CW/CCW swap events

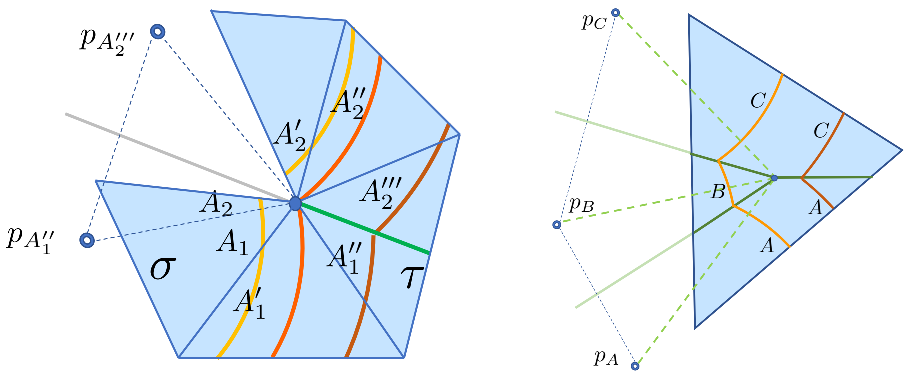


Figure 4.16: Vertex event and collision event

we do the manipulation

$$-I_{A_1'} - I_{A_1} - I_{A_2} - I_{A_2'} - I_{A_2''} - \quad \Longrightarrow \quad -I_{A_1'} - \boxed{I_{A_1''} - I_{A_2'''}} - I_{A_2''} -$$

Remark 4.3.2. Our algorithm may belatedly detect a vertex event which has actually occurred in the past. This delay is due to our simplification rule to ignore the tangency of the wavefront and an edge. In Figure 4.17, the arc A gets to be tangent to the edge e, and soon after, it meets a vertex v, but we do not recognize this 'partial propagation' of A, because I_A.IsPropagated is still being 'No'. When A reaches the endpoint ξ_A of

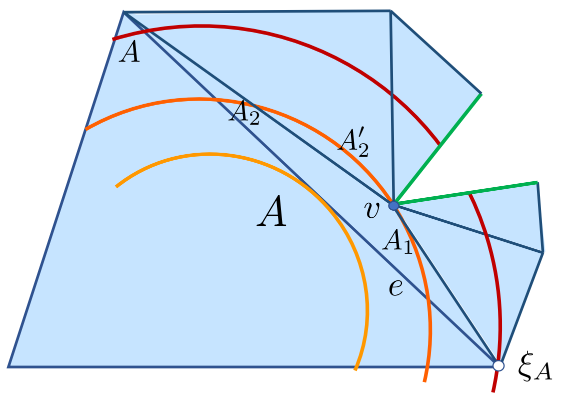


Figure 4.17: Delayed vertex event

its extent $e_A \subset e$, we update I_A.IsPropagated to be 'Yes', and only then new twins $I_{A_1} - I_{A_2}$ are recognized. The next Detection step now detects this vertex event at v. Processing and Trimming perform and draw the cut-locus created at v belatedly.

Collision event

A collision event happens when three consecutive arcs, say A, B, C in order, lie on the same face and B.Extent is empty (Figure 4.16, the right). Note that the final event is detected as three collision events that occur at the same point. If $W(r)$ consists of only three arcs and intervals, the collision event is the final event: just stop the algorithm. Otherwise, the manipulation is simply to delete I_B:

$$-I_A - I_B - I_C- \quad \Longrightarrow \quad -I_A - I_C-$$

4.3.7 Trimming

After Processing is finished, some of temporary extents of new/remaining intervals need to be corrected. This editing process is based on a similar one called *trimming* in the

MMP algorithm, but slightly modified. For each pair of neighboring non-twin intervals sharing the same face created in Processing, we check whether there is an overlap or not; if so, we correct it and update their items I.Extent. Let $I - J$ be such a pair of intervals. We can divide into two possible cases according to whether I.Edge is equal to J.Edge or not. The former case is the same as described in the MMP algorithm, but the latter case is our original generalization. In the former case, we calculate the ridge point which hits the common edge of I and J, and set it as the end point of I and the starting point of J. In the latter case, we calculate the possible ridge point hitting each edge of I and J. Namely, if the ridge point hits I.Edge, set it as the end point of I and set J.Extent to be empty, and if the ridge point hits J.Edge, set it as the starting point of J and set I.Extent to be empty.

In the process, it can happen that at least one of twin intervals has the empty extent. We say that they are *redundant twin*. By definition, each of twin intervals must have non-empty extent, thus we need to resolve the redundant twin. If only one of their extents is empty, remove the interval, and if both extents are empty, remove one of them, e.g., let the first interval remain (notice that any enriched interval with the empty extent is still in use in the expression of the wavefront). Finally we produce the new interval loop.

4.4 Computational Complexity

4.4.1 Theoretical Upperbound

We assume that the source point s is generic and the wavefront collapses to the farthest point without any bifurcation events during the propagation. Let n be the number of vertices of $\mathcal{P}$. Then the number b of (undirected) edges is $3(n-2)$ and the number c

of faces is $2(n-2)$; indeed, we have $2b = 3c$ and $n - b + c = 2$ (the Euler characteristic of the 2-sphere).

Lemma 4.4.1. The number of the vertex events is n. The number of the collision events is $n-2$ (here the final event is counted as one collision event).

Proof. The number of ridge point on the wavefront increases by one at a vertex event, decreases by one at a collision event except the final event, and decreases by three in the final event. Other types of events do not change the number. $\square$

Lemma 4.4.2. At any given time r, the number of the ridge points is $O(n)$.

Proof. The number of the ridge points in $W(r)$ is equal to or less than the number of vertex events happened, so it is $O(n)$. $\square$

Lemma 4.4.3. The sum of the numbers of the edge (cross/swap) events is $O(n^2)$.

Proof. It is equal to how many times the cut-locus C intersects the edges. For each edge e, let a an intersection point of C and e. a determines a sub-tree of C, which consists of the ridge points going to a. Those sub-trees are disjoint, therefore the number of possible intersection $C \cap e$ is at most n (since every ridge originated from at least one vertex). Thus the number of all intersections is bounded by nb, therefore $O(n^2)$. $\square$

Lemma 4.4.4. All vertex events take $O(n)$ time in total to be processed. Each of other events takes $O(1)$ time per event to be processed.

Proof. The total number of calculation caused by all vertex events is estimated to be $2b\,(= 6(n-2))$, which is the number of directed edges. A cross event or swap event takes $O(1)$ time, because it has at most two intervals to be propagated or deleted. A collision event takes $O(1)$ time, because it has only one interval to be deleted and two adjacent intervals to be updated. $\square$

Theorem 4.4.5. Our algorithm takes $O(n^2 \log n)$ time and $O(n)$ space.

Proof. While each event takes $O(1)$ time on average to be processed, it requires $O(\log n)$ time to be scheduled using a priority queue. The overall number of the events is $O(n^2)$, thus the time complexity is $O(n^2 \log n)$. There are $O(n)$ intervals in the interval loop at any given time. Because at most one event is associated with an interval, there are $O(n)$ events in the event queue at any given time. thus the space complexity is $O(n)$. $\square$

Our algorithm can support the shortest path query using extra space complexity:

Theorem 4.4.6. Our algorithm takes $O(n^2 \log n)$ time and $O(n^2)$ space for supporting the shortest path query.

Proof. To do this, all intervals that are generated and removed from the wavefront during the algorithm running are required to the path query. They must be retained in the memory. For each edge, intervals are separated by the intersections with the cut-locus. Therefore there are $O(n^2)$ intervals overall and the space complexity is $O(n^2)$. The time complexity does not change. $\square$

4.4.2 Experimental Result

An experimental result regarding computational complexity is shown below. We took the recursively subdivided surfaces of a regular octahedron using the Loop subdivision scheme [10] (Figure 4.18).

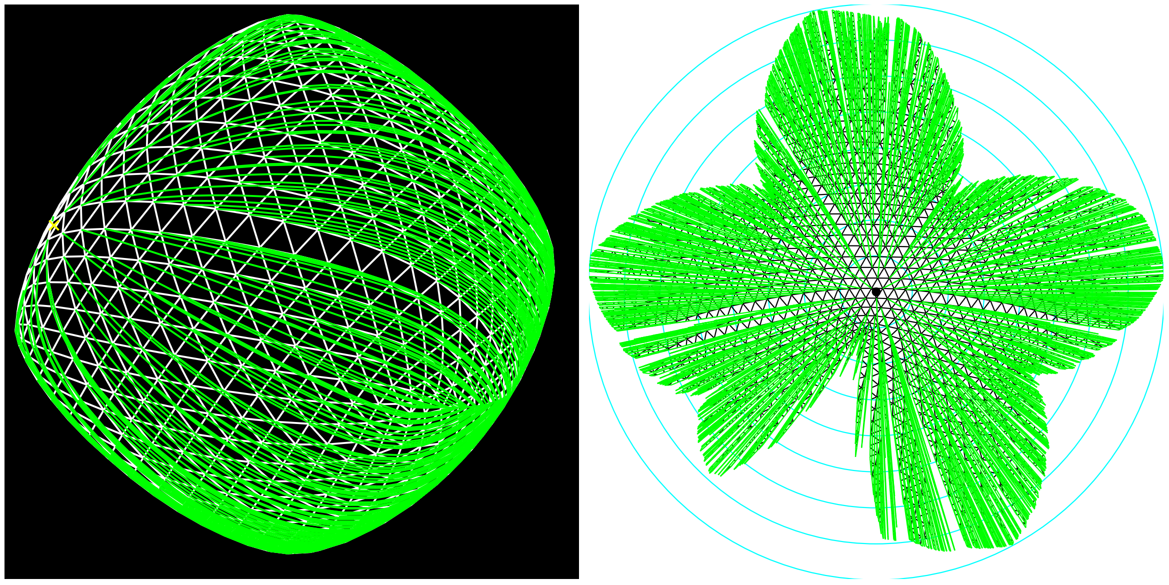


Figure 4.18: Level-4 subdivided surface of an octahedron and its source unfolding

The table below shows the *level* (how many times the subdivision performed from the initial octahedron), the number of vertices, the number of faces, the overall computation time, the memory usage, the number of total processed events, and the computation time per event. Here the $O(n^2)$ space variant (supporting the shortest path query) is used.

level	vertices	faces	time (sec.)	memory (MB)	events	μs/event
4	1026	2048	0.051	11	15737	3.2
5	4096	8192	0.506	71	125350	4.0
6	16386	32768	4.315	481	889247	4.8
7	65538	131072	40.214	3788	7158370	5.6

Like an experimental result of the MMP algorithm by Surazhsky et al. [17], experimental performance of our algorithm is sub-quadratic, both in terms of time and space. This is due to the fact that estimation of the number of edge events given by

Lemma 4.4.3 is too pessimistic and sub-quadratic in practice. Note that the memory usage is measured by the runtime, and in fact, contains the 3D mesh data as well as other miscellaneous things that make our program actually works. We can see that the computation time per event is clearly linearly correlating with $\log n$, and considering that, an experimentally-estimated complexity in this example is given by $O(n^{1.47} \log n)$ time and $O(n^{1.47})$ space, since the number of events is estimated to be $O(n^{1.47})$.

Chapter 5

Conclusion and Future Work

In Chapter 3, we have formulated the single-source geodesics enumeration problem and presented two data structures for the problem: the complete geodesic interval tree and the reduced geodesic interval tree. Although we could not provide their worst-case bounds in terms of the input size, experiments suggested that the reduced geodesic interval tree is practically more efficient on a nonconvex polyhedron in terms of running time and memory consumption. On the assumption of reasonable complexity of the input mesh, our method allows geodesics to be computed in reasonable time, which opens up possibility of finding useful non-shortest geodesics which traditional shortest path algorithms cannot find. We have given the complexity of our algorithms in terms of the size of the output geodesic interval trees. However, it largely depends on the geometry of the polyhedron, and a (non-trivial) complexity estimation is likely to include not only the input size (such as the number of vertices) and R, but also geometrical characteristics of the polyhedron (such as discretized curvatures). Therefore, rigorous analysis of complexity may involve mathematical study of geodesics themselves on a polyhedron. Geodesics yielded by our method can be used to approximate geodesics on a smooth surface, although more accurate or efficient algorithms tailored for smooth surfaces could be developed.

In Chapter 4, we have proposed a novel generalization of the MMP algorithm; it produces an interactive visualization of the wavefront propagation and the cut-locus on a convex polyhedral surface $\mathcal{P}$, and finally provides a nice planar unfolding of $\mathcal{P}$ without any overlap, instantaneously and accurately. Here we consider *generic* source points, that is sufficient for our practical purpose, and indeed, that enables us to classify what kind of geometric events arises in the wavefront propagation and makes the algorithm simple enough to be treated. A main idea is to introduce the notion of an *interval loop*, which is a new data structure representing the wavefront. It is propagated as the time (distance) r increases. The computational complexity of our algorithm is the same as the original MMP, while our actual use is supposed for polyhedra with a reasonable size of number n of vertices. We have successfully implemented our algorithm to computer – it works well as expected and we have demonstrated a number of outputs. There is still large room for further development. Extension of our method to non-convex polyhedra might be straightforward. However, a major downside of our method is that it cannot deal with bifurcation events, which are practically common to occur and more so in non-convex cases. In principle, brute-force detection of the bifurcation events should be possible with an increased time complexity of $O(n^3 \log n)$ by checking every pair of arcs existing on a common face. As we discussed in Section 4.1, we can compute cut-loci on a convex polyhedron in $O(n^2 \log n)$ time using Voronoi diagrams, even with existence of a bifurcation event. However, this strategy does not work on a non-convex polyhedron because, on a non-convex polyhedron, the intersection of a cut-locus and a face is not always a Voronoi diagram. In fact, it is not even an additively-weighted Voronoi diagram in general. Therefore, it would be valuable if one could develop an algorithm of $O(n^2 \log n)$ time complexity capable of work on non-convex polyhedra with existence of bifurcation events.

As we discussed in Chapter 2, there are often multiple similar geodesics on a polyhedron due to the existence of spherical vertices (see Figures 2.3 and 2.4). When we see the polyhedron as the exact description of the geometry of our interest, they are certainly distinct geodesics, which is the viewpoint we took in this paper. However, we sometimes have to see the polyhedron as an approximate description of a smooth surface of our interest. From this viewpoint, this property of polyhedral geodesics is unfavorable, because these similar geodesics often correspond to a single geodesic on the smooth surface. This distinction is crucial to deal with a cut locus of a 3D triangular mesh, as every spherical vertex generates a branch of the cut locus on a polyhedron (see Figures 4.3, 4.6 and 4.18). As a result, when we take a sequence of polyhedra that converges to a smooth surface, the cut loci of the polyhedra usually do not converge to the cut locus of the smooth surface. In this sense, cut loci of a polyhedron and a smooth surface are different in nature. In fact, we can unfold a convex polyhedron into a plane by cutting along a cut locus (see Figure 4.5, 4.6 and 4.18), whereas this feature is not available on a smooth surface. It is challenging to compute a cut locus of a 3D mesh as a smooth surface. Firstly, the problem is not mathematically well-defined unless we fix an interpolation method to produce a smooth surface from the mesh, while it may be practically sufficient to supply a method to filter the polyhedral cut-locus into more scarce cut-locus of the corresponding smooth surface. Secondly, computing geodesics on a smooth surface is not as easy as on a polyhedron, because we cannot unfold the surface into a plane. Thus, it could be another future work.

Acknowledgements

I appreciate Professor Toru Ohmoto. He accepted me as a doctoral student even though I hadn't decided on a research subject at that time, and provided a topic about cut loci, which led me to my first academic paper. When I had difficulty writing the paper, he wrote a prototype for me even though the topic was different from his main research area. Also, he patiently listened to me even when I ran into difficulty to get a fruitful result. Lastly, he also gave me advice about my second academic paper on geodesics enumeration. I appreciate Professor Hiroki Arimura. During the entrance interview, he convinced the judges that the enumeration of geodesics was possibly a new research topic. I might not be able to enter the course without his convincement. After Professor Ohmoto moved to Waseda University, he accepted me as a student and taught me various teachings about being a researcher until late at night, such as the importance of respecting and mentioning previous research. Lastly, he gave me advice about my second academic paper on geodesics enumeration.

I appreciate Professor Zin Arai, who was my supervisor during my master's course. When I could not decide on a research subject at that time, he suggested a topic about shortest paths and geodesics on a polyhedron, which led me to my current research subject.

I appreciate Professor Takashi Horiyama and Professor Yasuhide Numata for being my vice-chairs on my dissertation committee and giving me advice about the dissertation. I sincerely appreciate all the people who helped and advised me.

Lastly, I appreciate my parents for physically, mentally, and financially supporting my personal life. I couldn't complete my doctoral course without their assistance.

Bibliography

[1] A. D. Alexandrov. *Convex polyhedra*, volume 109. Springer, 2005.

[2] V. I. Arnol'd. *Catastrophe theory*. Springer Science & Business Media, 2003.

[3] P. Bose, A. Maheshwari, C. Shu, and S. Wuhrer. A survey of geodesic paths on 3D surfaces. *Comput. Geom.*, 44(9):486–498, 2011.

[4] J. Chen and Y. Han. Shortest paths on a polyhedron. In R. Seidel, editor, *Proceedings of the Sixth Annual Symposium on Computational Geometry, Berkeley, CA, USA, June 6-8, 1990*, pages 360–369. ACM, 1990.

[5] P. Cheng, C. Miao, Y. Liu, C. Tu, and Y. He. Solving the initial value problem of discrete geodesics. *Comput. Aided Des.*, 70:144–152, 2016.

[6] K. Crane. Discrete differential geometry: An applied introduction. *Notices of the AMS, Communication*, pages 1153–1159, 2018.

[7] K. Crane, M. Livesu, E. Puppo, and Y. Qin. A survey of algorithms for geodesic paths and distances. *CoRR*, abs/2007.10430, 2020.

[8] E. D. Demaine and J. O'Rourke. *Geometric folding algorithms - linkages, origami, polyhedra*. Cambridge University Press, 2007.

[9] J. Itoh and R. Sinclair. Thaw: A tool for approximating cut loci on a triangulation of a surface. *Exp. Math.*, 13(3):309–325, 2004.

[10] C. Loop. Smooth subdivision surfaces based on triangles, master's thesis. *University of Utah, Department of Mathematics*, 1987.

[11] C. Mancinelli, M. Livesu, and E. Puppo. Practical computation of the cut locus on discrete surfaces. *Comput. Graph. Forum*, 40(5):261–273, 2021.

[12] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou. The discrete geodesic problem. *SIAM J. Comput.*, 16(4):647–668, 1987.

[13] D. M. Morera, L. Velho, and P. C. P. Carvalho. Computing geodesics on triangular meshes. *Comput. Graph.*, 29(5):667–675, 2005.

[14] D. M. Mount. On finding shortest paths on convex polyhedra. Technical report, MARYLAND UNIV COLLEGE PARK CENTER FOR AUTOMATION RESEARCH, 1985.

[15] K. Polthier and M. Schmies. Straightest geodesics on polyhedral surfaces. In J. W. Finnegan and D. Shreiner, editors, *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2006, Boston, Massachusetts, USA, July 30 - August 3, 2006, Courses*, pages 30–38. ACM, 2006.

[16] N. Sharp and K. Crane. You can find geodesic paths in triangle meshes by just flipping edges. *ACM Trans. Graph.*, 39(6):249:1–249:15, 2020.

[17] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. J. Gortler, and H. Hoppe. Fast exact and approximate geodesics on meshes. *ACM Trans. Graph.*, 24(3):553–560, 2005.

[18] The CGAL Project. *CGAL User and Reference Manual.* CGAL Editorial Board, 5.5.2 edition, 2023.

[19] C. C. L. Wang. Cybertape: an interactive measurement tool on polyhedral surface. *Comput. Graph.*, 28(5):731–745, 2004.

[20] S. Xin and G. Wang. Efficiently determining a locally exact shortest path on polyhedral surfaces. *Comput. Aided Des.*, 39(12):1081–1090, 2007.

[21] S. Xin and G. Wang. Improving Chen and Han's algorithm on the discrete geodesic problem. *ACM Trans. Graph.*, 28(4):104:1–104:8, 2009.

[22] X. Ying, X. Wang, and Y. He. Saddle vertex graph (SVG): a novel solution to the discrete geodesic problem. *ACM Trans. Graph.*, 32(6):170:1–170:12, 2013.