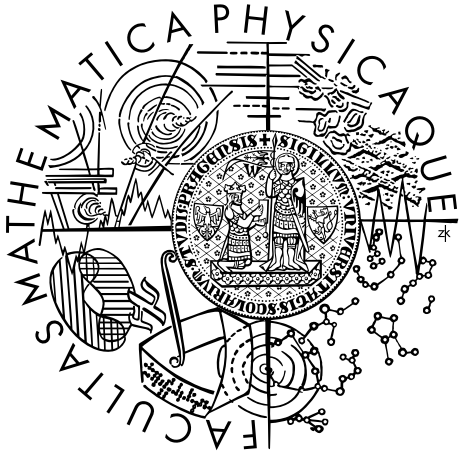
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Martin Maňák

Voronoi Diagrams for Spheres in E^3

Department of Software and Computer Science Education

Supervisor: Doc. Dr. Ing. Ivana Kolingerová

Study Program: Computer Science, Software Systems

2008

I would like to thank my supervisor, *Doc. Dr. Ing. Ivana Kolingerová* for her time, many useful consultations and suggestions, my parents for their endless support and my grandparents for their prays.

I declare that I have elaborated this thesis on my own and listed all used references. I agree with making this thesis publicly available.

Prague, Sep 7, 2008 Martin Maňák

Contents

Název práce: Voronoi Diagramy koulí v E^3
Autor: Martin Maňák
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí diplomové práce: Doc. Dr. Ing. Ivana Kolingerová
e-mail vedoucího: kolinger@kiv.zcu.cz

Abstrakt: Voronoi diagramy (*VD*) pro zadanou množinu objektů popisují jejich prostorové vztahy. *VD* pro množinu bodů v prostoru je velmi dobře prozkoumaná oblast a existuje celá řada algoritmů pro jejich konstrukci. Ačkoliv diagramy pro množinu koulí jsou také již mnoho let známy, popis jejich vlastností a algoritmy pro jejich konstrukci jsou relativně novou záležitostí. Nabývají na významu s výzkumem v oblasti molekulární biologie.

Cílem této práce je prozkoumat teorii *VD* pro množinu koulí v prostoru, implementovat některý z existujících algoritmů pro jejich konstrukci jako knihovnu a vyzkoušet ji v experimentech na realných datech.

Keywords: Voronoi diagramy koulí; aditivně vážené Voronoi diagramy; sledování hrany; proteiny

Title: Voronoi Diagrams for Spheres in E^3
Author: Martin Maňák
Department: Department of Software and Computer Science Education
Supervisor: Doc. Dr. Ing. Ivana Kolingerová
Supervisor's e-mail address: kolinger@kiv.zcu.cz

Abstract: Voronoi diagrams (*VD*) describe spatial relationships among a given set of input sites. The family of *VD* for a set of points is a well-explored domain and effective algorithms for their construction exist. Although the family of *VD* for a set of spheres has been known for many years, properties of these diagrams and algorithms for their construction are a relatively new thing. Their importance grows with the development in the area of molecular biology.

The goal of this work is to survey the theory behind *VD* of spheres, implement one of the existing algorithms for their construction as a library and use the library on a real data, such as proteins.

Keywords: Voronoi diagrams of spheres; additively weighted Voronoi diagrams; edge tracing; proteins

Chapter 1

Introduction

Voronoi diagrams (*VD*) describe spatial relationships among a given finite set of objects. The term was formally defined by Russian mathematician G. F. Voronoi at the beginning of the 20^{th} century [18].

There are many variants of *VD*. The best known is probably *VD(P)* (Voronoi diagram of points) in Euclidean space: We are given a finite set of points S, which are the Voronoi sites (or generators). Each site $s \in S$ has a Voronoi region $V(s)$ consisting of all points that are closer to s than to any other site. Refer to Fig. 1.1 for an E^2 example.

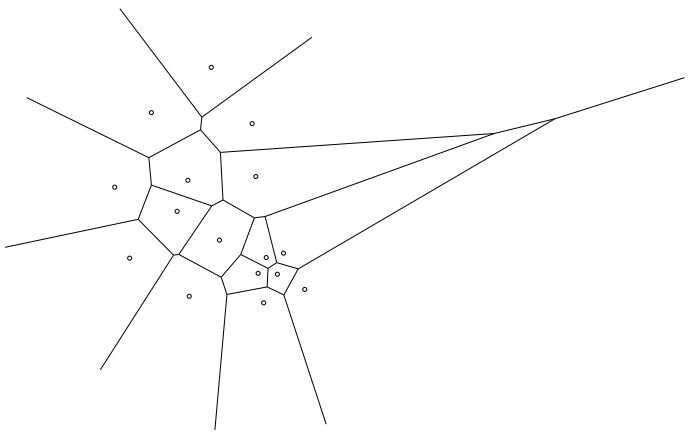


Figure 1.1: Voronoi diagram for a set of points in E^2

The *VD(P)* family of Voronoi diagrams has been studied for many years and its properties and algorithms for their construction are well-known [7, 15].

Depending on the application, we might want to change the family of *VD(P)* into

something else that would better suit our problem's domain. For example, we might want to:

- specify the dimension of the metric space
- employ another metric than the Euclidean distance
- use some different class of generators than points

This usually leads to a different family of diagrams with different properties than *VD(P)*. For example, we might want a VD of line segments in E^3, of circles in E^2 or assign a weight to every generator and change the metric in order to end up with some kind of weighted *VD(P)*.

With the development in the field of molecular biology, some geometric problems have arisen that could be solved more easily if some kind of *VD* for the given set of atoms was available. This motivates the use of *VD(S)* (Voronoi diagram of spheres) because molecules consist of atoms and each atom can be approximated by a sphere. See Fig. 1.2 for an example of this kind of *VD*.

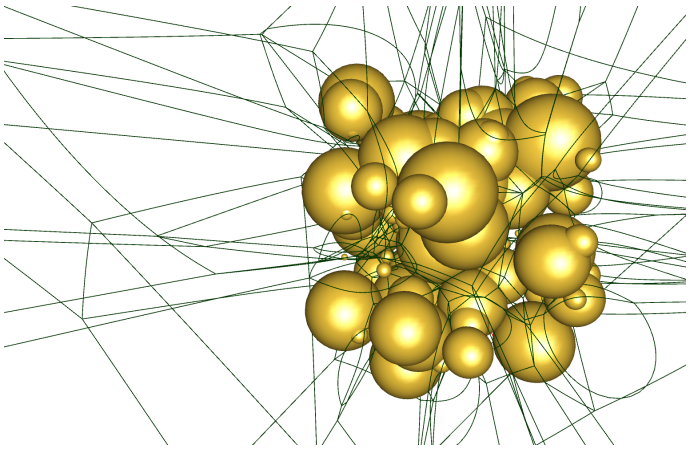


Figure 1.2: Voronoi diagram for a random set of spheres in E^3

Zemek describes an approach to searching tunnels in protein molecules in his thesis [20]. He uses a regular triangulation for their searching. The tunnel analysis is used in protein engineering for the research and modifications of protein molecule behavior.

Kim et al. showed, how *VD(S)* can be used for the computation of surfaces defined on a protein and the extraction and characterization of interaction interfaces between multiple proteins [10].

Gavrilova and Rokne mentioned an application of *VD(S)* to the study of the structure of polydisperse packing of spheres [6]. The diagram can be used for the analysis of empty spaces (voids) in the system of spheres.

At Voronoi Diagram Research Center [1], they have extensively studied Voronoi diagrams of circles and spheres for last few years. In 2004, Kim et al. reported the *edge-tracing algorithm* for the construction of *VD(S)* [8, 9]. The algorithm starts at an arbitrary Voronoi vertex and traces Voronoi edges in order to construct the diagram. In 2005, they reported the *region-expansion algorithm* [12, 13], which in fact turns a *VD(P)* into the *VD(S)* by expanding generators in their radii. They discussed the representation of the topology in [3]. The proposed data structure is a variation of the *radial-edge* or the *partial-entity data structure* but greatly simplified to the problem domain. In 2006, they discussed QT(S) (*quasi-triangulation*) as the dual of *VD(S)* and introduced the *interworld data structure* [11, 16], which can be used to store the topology of the diagram or its dual more compactly than with the previously mentioned structure. They also introduced the *face-tracing algorithm* as the dual to the *edge-tracing algorithm*.

Although *VD(S)* truly represents the spatial relationships among a set of spheres, *VD(P)* is often used instead, because *VD(P)* and its variants are well-explored and effective algorithms for their construction are known.

The goal of this work is to survey the theory behind *VD(S)*, implement one of the existing algorithms for their construction as a library and use the library on a real data, such as proteins. Our priority[1] is to get some diagrams to experiment with and hence we do not address issues regarding numerical stability of the solution nor the performance.

The work is divided into several chapters. Basic definitions and properties of *VD(S)* and its dual structure are in Chap. 3. Differences between *VD(S)* and *VD(P)* are summarized in Chap. 4. Algorithms for the construction of *VD(S)* are briefly summarized in Chap. 5. There is also a discussion to the algorithm chosen for implementation – the *edge-tracing* algorithm. Chap. 6 is dedicated to the computation of geometry of Voronoi vertices and edges. The construction of *VD(S)* by *edge-tracing* together with our modifications and improvements can be found Chap. 7. Details of the algorithm implementation and the related library are in Chap. 8. Our experiments performed on both protein and random data, their results and pictures of real diagrams can be found in Chap. 9.

[1] emphasized by the supervisor

Chapter 2

Preliminaries

In this work, E^n is the Euclidean metric space $(\mathbb{R}^n, d)$ defined over the n-dimensional vector space of real numbers $\mathbb{R}$ with the Euclidean metric $d : \mathbb{R}^n \to \mathbb{R}$. The metric is defined as

$$d(x, y) = \|x - y\| = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

We use the metric to measure the distance from a point x to a sphere $s = (c, r)$ as

$$d(x, c) - r$$

where c is the center and r is the radius of the sphere s. This is shown on Fig. 2.1.

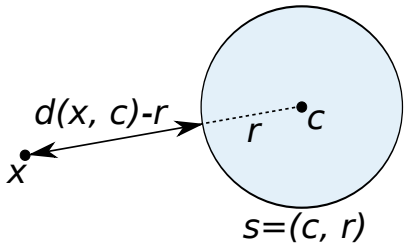


Figure 2.1: The Euclidean distance of the point x from the sphere s.

When we are given a point x and two spheres $s_1 = (c_1, r_1)$ and $s_2 = (c_2, r_2)$, this allows us to compare the distance of x with respect to s_1 and s_2 as

$$d(x, c_1) - r_1 \leq d(x, c_2) - r_2$$

Some figures representing three-dimensional situations, such as Fig. 2.1, 3.2a or 3.4, are drawn in a schematic 2-dimensional view or in a 2-dimensional analogy and

others, such as Fig. 3.3c, are drawn in a 3-dimensional view. Sometimes, we are using these two views interchangeably without any explicit warning. When it does matter, we explicitly specify the lower dimension. When it does not matter, we specify or do not specify the dimension.

Chapter 3

VD(S) and *QT(S)*

3.1 *VD(S)* as a set of Voronoi regions

Definition 3.1. Let $S = \{s_i | i \in \{1, \ldots, n\}\}$ be a set of n spheres in E^3, where $s_i = (c_i, r_i)$ is a *generator sphere* with the center $c_i \in \mathbb{R}^3$ and the radius $r_i \in \mathbb{R}$.

Then for each generator $s_i \in S$, its *Voronoi region* is defined as

$$VR_i = \{x | \forall s_j \in S, j \neq i : d(x, c_i) - r_i \leq d(x, c_j) - r_j\}$$

and for the set S, its *Voronoi diagram* is defined as

$$VD(S) = \{VR_i | s_i \in S\}.$$

For the purpose of diagram construction, we allow generator spheres to intersect each other, but we do not allow any sphere to be fully contained in another. Furthermore, we assume that the spheres are in general positions. The first assumption prohibits empty Voronoi regions and the second assumption prohibits over-constrained Voronoi vertices, edges and faces (constrained by more spheres than it is necessary).

The interesting part about a Voronoi region is its boundary. It consists of lower-dimensional primitives: 2-dimensional faces, 1-dimensional edges and 0-dimensional vertices. Boundaries "connect" regions together.

For the purpose of defining Voronoi faces, edges and vertices, we will define Voronoi regions by an equvalent definition as in Gavrilova's thesis [7].

Definition 3.2. For two different spheres $s_1 = (c_1, r_1)$ and $s_2 = (c_2, r_2)$ in E^3, their *Euclidean bisector* is defined as

$$B(s_1, s_2) = \{x | d(x, c_1) - r_1 = d(x, c_2) - r_2\}.$$

The bisector divides the space into two quasi-halfspaces[1]. The *quasi-halfspace* of s_1 is defined as

$$H(s_1, s_2) = \{x | d(x, c_1) - r_1 \leq d(x, c_2) - r_2\}$$

In Fig. 3.1, the bisector $B(s_1, s_2)$ divides the space into two connected quasi-halfspaces $H(s_1, s_2)$ and $H(s_2, s_1)$. Note that this holds even when the spheres would intersect each other, because $r_1 - r_2$ is constant for all x in the definition of $H(s_1, s_2)$.

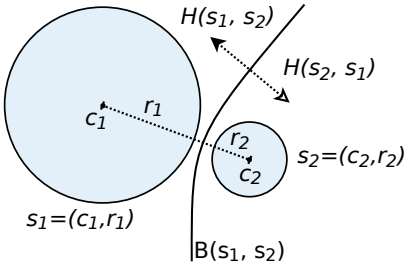


Figure 3.1: Bisector and halfspaces in E^2

Definition 3.3. Let S be a set of generator spheres. For each generator $s_i \in S$, we define its *Voronoi region* as the intersection of all quasi-halfspaces $H(s_i, s_j)$:

$$VR_i = \bigcap_{\substack{s_j \in S \\ j \neq i}} H(s_i, s_j)$$

3.2 Faces, edges and vertices

From Def. 3.3 it immediately follows that the boundary of a Voronoi region consists of 2-dimensional hyperplanes[2] - bisector subsets.

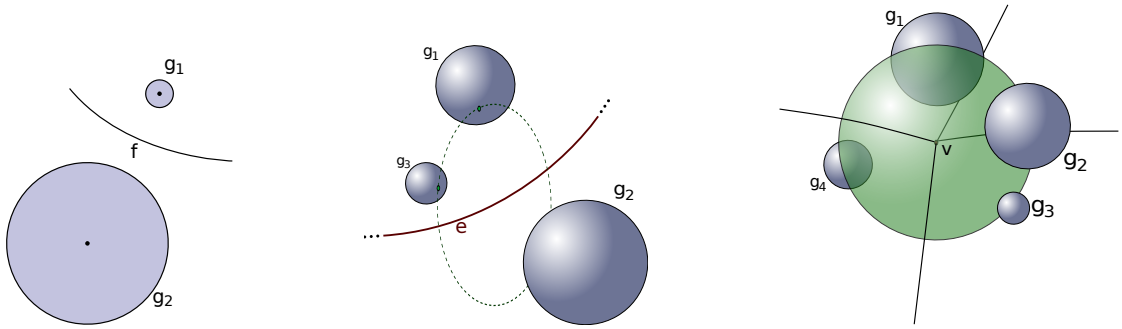
- Each maximal connected 2-dimensional subset is a *Voronoi face*. It is defined by two immediately neighboring generators.
- Each maximal connected 1-dimensional intersection of faces is a *Voronoi edge*. It is defined by three neighboring generators. Edges are oriented[3]
- A *Voronoi vertex* is the 0-dimensional intersection of edges. It is defined by four neighboring generators.

[1] the bisector does not have to be a linear plane in general

[2] they do not have to be linear planes

[3] see Sec. 3.4 for details

We require a subset to be maximal and connected, since a single bisector can participate to the boundary of a region by a number of isolated hyperplanes. Basic *VD(S)* elements and their defining generators are shown on Fig. 3.2.



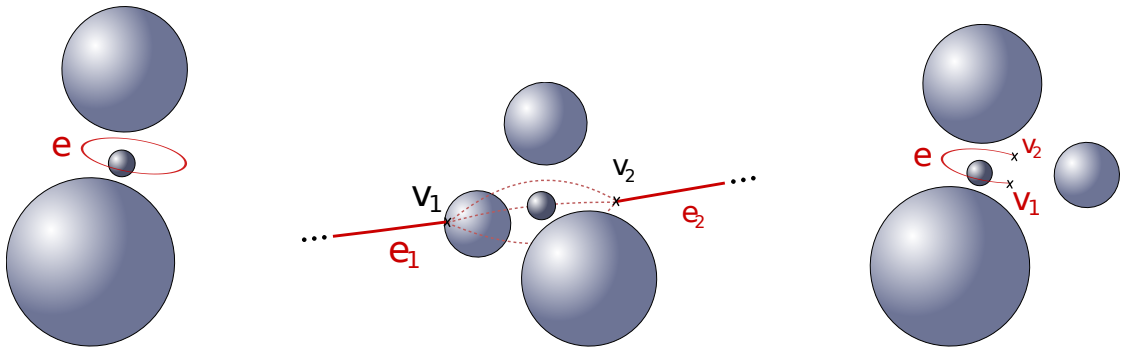
(a) Voronoi face f defined by 2 generators (g_1, g_2) in E^2

(b) Voronoi edge e defined by 3 generators (g_1, g_2, g_3)

(c) Voronoi vertex v as the center of an empty sphere tangent to 4 generators (g_1, g_2, g_3, g_4)

Figure 3.2: Schematic view to faces, edges and vertices in *VD(S)*.

Some differences between *VD(P)* and *VD(S)* are shown on Fig. 3.3. Three generators can form an elliptic edge without any bounding vertex, an edge does not have to be defined uniquely by three generators and similarly a vertex does not have to be defined uniquely by four generators.



(a) Big brothers form an elliptic edge e

(b) Two vertices v_1 and v_2 defined by a single quadruple of generators, two edges e_1 and e_2 defined by a single triplet

(c) Two vertices $v_1 \in e$ and $v_2 \in e$ by four generators

Figure 3.3: Cases that can occur in *VD(S)* but not in *VD(P)*.

A Voronoi face f is always a connected subset of *hyperboloid of two sheets*, each edge e is a *conic section* and each vertex v is a point equidistant to the four defining generators. The position of a Voronoi vertex is given by the center of a sphere

tangent to four generators defining the vertex - a *vertex sphere*. The vertex sphere is called *empty*, since it never intersects nor contains any other generator.

3.3 Separation of geometry and topology

For Voronoi regions, faces, edges and vertices, we distinguish their geometry from their topology:

- The *geometry* of an element is a set of points in the metric space. For example, the geometry of a Voronoi vertex is a single point (its position). The geometry of a Voronoi face is a set of points on its hyperplane.
- The *topology*, on the other hand, describes the constraints of the geometry in terms of their defining generators, and incidence relationship among vertices, edges, faces and regions. For example, the topology of a Voronoi vertex consists of its four defining generators and we may want to know which edges, faces or regions are incident to the vertex.

Tab. 3.1 summarizes *VD(S)* topology and geometry in E^3.

- A vertex is always defined by 4 generators and it is incident to exactly 4 edges.
- An edge is defined by 3 generators, it is incident to exactly 3 faces and can be bounded by maximally two vertices. Unfortunately, it can be also unbounded (elliptic or infinite).
- A face is defined by 2 generators and it connects exactly two regions. It can be bounded by a number of edges and can have some topological holes inside.
- A region is defined by 1 generator and can be bounded by a number of faces.

Element	**Topology**	**Geometry**
Region	1 generator	sub-space
Face	2 generators, 2 incident regions	hyperboloid
Edge	3 generators, 3 incident faces	conic section
Vertex	4 generators, 4 incident edges	point

Table 3.1: Summary of *VD(S)* topology and geometry in E^3.

In general, an n-tuple of generators, $n \in \{2, 3, 4\}$, does not have to define exactly one element as it is common in *VD(P)*. A quadruple of generators can define two different vertices, for instance. See Chap. 4 for details and Fig. 4.1 for the summary.

For the computation of topology, only the computation of vertex geometry is inevitable. The remaining geometry of edges and faces can be computed from the knowledge of the topology.

3.4 Edge orientation via angular distance

Motivation

The ability to determine if a point lies on a valid edge segment becomes important at the moment when one needs to compute its geometric representation. Furthermore, the main idea of the edge-tracing algorithm[4] heavily depends on the ability to establish an ordering among points of a possible edge.

Terminology

Given an edge e bounded by one or two vertices, the first vertex is called a *start vertex* and the second is an *end vertex*. Three generators that define the edge are called *gate balls* and they are shared by the vertices. The single remaining generator that defines the start or end vertex is called a *start ball* or an *end ball*, respectively.

An *angular distance* of a point $p \in e$ from the start vertex $v_s \in e$ is the angle $\theta = \angle v_s c_{g1} p$, where c_{g1} is the center of the first[5] gate ball with the minimal radius. $\theta \in (0, 2\pi)$ is a directed distance, positive from v_s in the edge direction.

In Fig. 3.4, there is an edge e oriented from its start vertex v_s to the end vertex v_e with their corresponding vertex spheres S_s and S_e. Only two of the three gate balls defining the edge are shown: b_{g1} and b_{g2}. The angular distance of v_e from v_e is denoted as θ.

Classification of angular distance

Depending on its role with respect to an edge e, we classify an angular distance as *valid*, *unreachable* or *prohibited*, as it is shown on Fig. 3.5.

[4] an algorithm that computes *VD(S)* topology – see Sec. 5.1

[5] in the sense of a global ordering of generators, *e.g.* by their indices

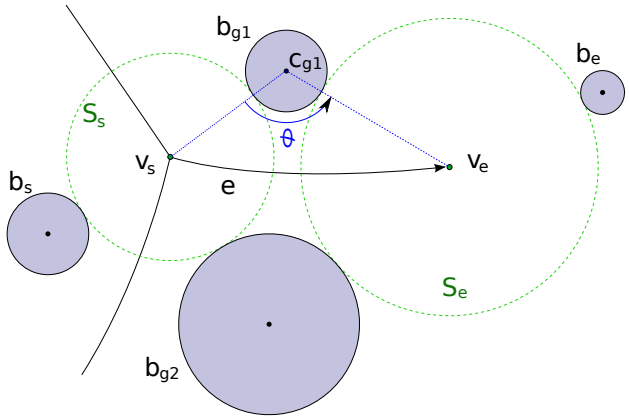


Figure 3.4: Angular distance and edge orientation.

In Fig. 3.5, the edge e has a start vertex v_s and its orientation is indicated by an arrow. Starting from v_s, the angular distance is classified as *valid* until it points to the infinity. Since then, it is classified as *unreachable* – there is no point $p \in e$ that could possibly have any angular distance from this interval. The rest is classified as *prohibited*, since the edge would intersect the fourth generator that defines v_s if it was from this interval.

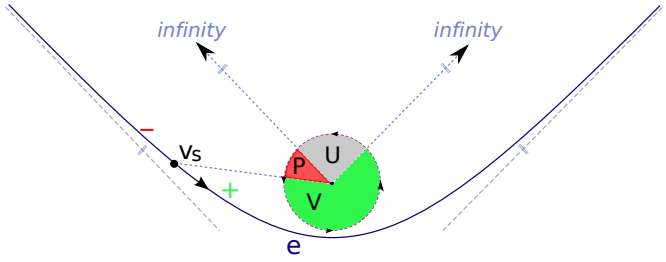


Figure 3.5: Angular distance classified as *valid*, *unreachable* and *prohibited*.

3.5 Quasi-triangulation

A quasi-triangulation is the dual of *VD(S)*[6]. The word *quasi* highlights the difference from Delaunay triangulation (DT) for a set of points. Note however, that $QT(S)=DT$ when all spheres in S are of equal radii. This chapter summarizes [11].

Definition 3.4. Let S be the set of generator spheres in E^3. A *quasi-triangulation* of the set S is defined as

$$QT(S) = \{V^Q, E^Q, F^Q, C^Q\} = Dual(VD(S))$$

where

$$\begin{aligned} V^Q &= \{v_i^Q | i \in \{1, 2, \dots\}\} \\ E^Q &= \{e_i^Q | i \in \{1, 2, \dots\}\} \\ F^Q &= \{f_i^Q | i \in \{1, 2, \dots\}\} \\ C^Q &= \{c_i^Q | i \in \{1, 2, \dots\}\} \end{aligned}$$

denote the sets of vertices, edges, faces and cells in the quasi-triangulation. $VD(S) = \{R^V, F^V, E^V, V^V\}$ is the Voronoi diagram (R^V, F^V, E^V and V^V are the Voronoi regions, faces, edges and vertices). *Dual* is a duality operator defined as follows

- $\forall r^V \in R^V : Dual(r^V) \equiv v^Q$ is the dual vertex. It is a topological point and corresponds to the center of the generator defining r_i.
- $\forall f^V \in F^V : Dual(f^V) \equiv e^Q$ is the dual edge. It is a topological line segment between two dual vertices
- $\forall e^V \in E^V : Dual(e^V) \equiv f^Q$ is the dual face. It is a topological triangle over three dual vertices. Every two vertices form a dual face.
- $\forall v^V \in V^V : Dual(v^V) \equiv c^Q$ is the dual cell. It is a topological tetrahedron over four dual vertices. Every two vertices from c_i^Q form a dual edge.
- $Dual(X) \equiv \{Dual(x) | x \in X\}$ – defines $Dual(VD(S))$ and its parts

In DT, all elements are topologically distinct (defined by different vertices) and each face belongs to a cell. But *QT(S)* differs from DT since *VD(S)* differs from *VD(P)*. There are following anomalies.

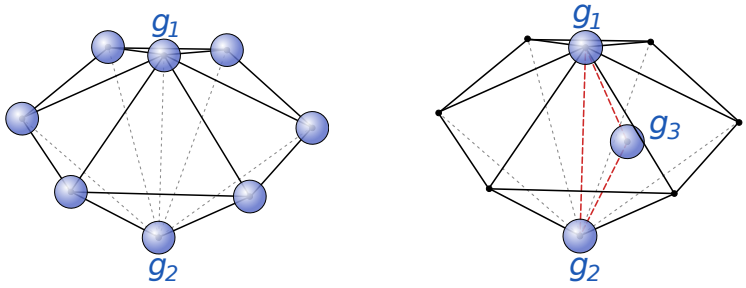
[6] see Def. 3.3 and Sec. 3.2

Multiplicity anomaly is caused by doubled Voronoi vertices[7]. In the dual space, the anomaly is interpreted as two tetrahedra sharing two or more faces. As a consequence, they are defined by the same four vertices.

Degeneracy anomaly is caused by elliptic Voronoi edges[8]. In the dual space, the triangle defining the dual face does not belong to any tetrahedra. It can be looked at as a tetrahedron degenerated to a triangle (such as g_1, g_2 and g_3 on Fig. 3.6b, for instance).

Singularity anomaly is caused by topological holes in Voronoi faces[9].

An example of the singularity anomaly is shown on the Fig. 3.6. An ordinary Voronoi face without any anomaly is shown on Fig. 3.6a. In the dual space, it is an edge between two vertices g_1 and g_2. On Fig. 3.6b is the same Voronoi face but with an elliptic edge causing a very simple topological hole inside the face. In the dual space, it corresponds to the triangle defined by g_1, g_2 and g_3.



(a) Ordinary Voronoi face shown in the dual space

(b) Voronoi face with a hole caused by an elliptic edge

Figure 3.6: Singularity anomaly

[7] see Fig. 3.3b, 3.3c and 4.1h
[8] see Fig. 3.3a and 4.1f
[9] see Fig. 4.1d

Chapter 4

Differences between *VD(P)* and *VD(S)*

The shape of Voronoi regions, and therefore of the entire diagram, is given by the shape of its bisectors[1]. Even when every bisector $B(s_1, s_2)$ in *VD(S)* is a hyperboloid in general, it is a simple linear plane when s_1 and s_2 have equal radii. Therefore, *VD(P)* $\subset$ *VD(S)*. To be more specific, *VD(P)* $=$ *VD(S)* when *all* generators are of equal radii.

Since bisectors in *VD(P)* are linear, their intersection is linear as well. Hence it can be described by a system of linear equations in terms of coordinates of the generator points. When the system is regular[2], it has a unique solution. As a result, elements of *VD(P)* are convex and uniquely defined by their generators. This also means that there are no topological holes inside any element. We define a duality as a Delaunay triangulation DT(P) and benefit from the fact it is a bijective transformation.

In *VD(S)*, on the other hand, bisectors are hyperbolic planes and all the good we used to have from the linearity is lost[3]. Regions do not have to be convex any more. They are star-shaped. Thanks to the requirement that there is no generator sphere fully contained in another and that we do not allow negative nor complex radii, there are *no topological holes in regions*. Unfortunately, this is not true for the remaining elements. There can be an arbitrary number of holes in a face or an edge. Because of its 2-dimensional nature, a single bisector can be split into multiple disconnected faces. Each face can have an arbitrary number of holes inside. An "edge" can be split into multiple disconnected segments and we have to treat each segment as a

[1] see Def. 3.3

[2] given points are linearly independent, *i.e.* the determinant is non-zero

[3] see Fig. 3.3 for an example

different edge element. Furthermore, we can get elliptic edges without any bounding vertex. Given configuration of generators does not have to define any edge or vertex at all or it can define a vertex in two different positions. Hence we can get two different vertices from four generators or a single vertex and two positions, where it could possibly be located.

Fig. 4.1 shows differences between *VD(P)* and *VD(S)* elements in a schematic view.

On Fig. 4.1a and 4.1b, there are *VD(P)* and *VD(S)* regions, respectively. The *VD(P)* region is convex and bounded by linear faces $f_1 \dots f_5$. Its *VD(S)* counterpart is bounded by hyperbolic faces and thus not convex. There are no topological holes.

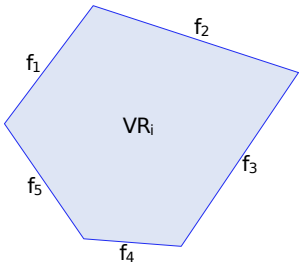
Fig. 4.1c shows a *VD(P)* face f. It is a subset of a linear plane bounded by a set of linear edges (which are not shown, it is just a schematic view). In the sense of topology, it is uniquely defined by two generator points.

Fig. 4.1f shows the *VD(S)* case. There are several isolated faces $f^I \dots f^V$. From the topological point of view, all these faces are defined by the same couple of generators. Hence, now from the geometrical point of view, all $f^I \dots f^V$ are exclusive subsets of the single bisector. In other words, where would be a single face f in *VD(P)*, there are now five faces $f^I \dots f^V$ in *VD(S)*.

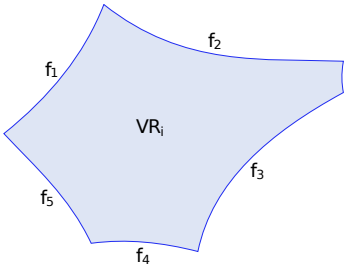
Note that f^{II}, f^{III} and f^V have some inner holes inside. Such a face has still its outer boundary formed by an edge-loop but it has also an inner edge-loop that makes a hole inside the face (in the geometrical sense as well as in the topological sense). Note that there is no edge that would connect the outer loop with the inner loop. These topological holes make the utilization of DT difficult and it is the reason why an inter-world data structure [11] uses an additional array of gates to overcome this problem (viz. Sec. 5.2).

Figures 4.1e and 4.1f show differences between *VD(P)* and *VD(S)* edges, respectively. This is similar to the case of faces but in a lower dimension. A simple *VD(P)* edge is a line segment, it is bounded by two vertices and its three generators define it uniquely. In *VD(S)* on the other hand, both hyperbolic and elliptic edges can be split into several segments: There are five different edges $e_1^I \dots e_1^V$ defined by the same triplet of generators that would otherwise define a hyperbolic edge e_1. It is similar for edges $e_2^I \dots e_2^V$ that would otherwise define an elliptic edge e_2 (imagine e_2 surrounded by five generators, for instance). The last edge e_3 is an elliptic edge with no vertices – this makes difficult to represent the diagram as a graph of edges over vertices.

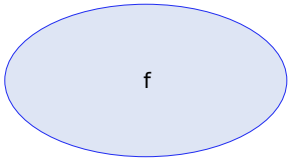
Figures 4.1g and 4.1h show that a vertex v in *VD(P)* is always defined by four generators uniquely, but in *VD(S)*, there can be two such vertices v^I and v^2. For example, these could be the vertices between e_1^I and e_1^{II} if the hole between them would be caused by a single generator.



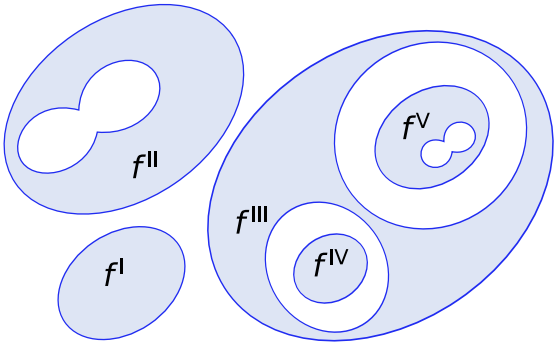
(a) VDP region VR_i bounded by faces $f_1 \dots f_5$ (2-dimensional analogy)



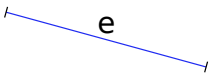
(b) VDS region VR_i bounded by faces $f_1 \dots f_5$ (2-dimensional analogy)



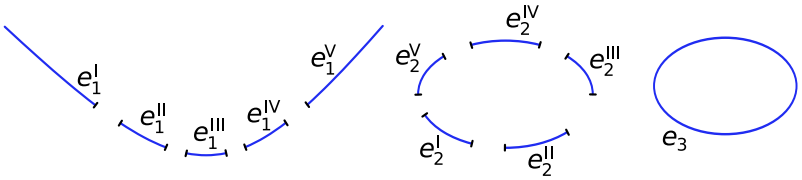
(c) VDP face f (without the geometry of bounding edges)



(d) VDS face f is split into isolated parts $f^I \dots f^V$ (each of them belongs to the same hyperbolic bisector); faces f^{II}, f^{III} and f^V have some topological holes inside

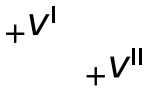


(e) VDP edge e



(f) VDS edges e_1 and e_2 are split into isolated parts $e_1^I \dots e_1^V$ and $e_1^I \dots e_1^V$, respectively; e_3 is an elliptic edge without any vertex

(g) VDP vertex v



(h) VDS vertex v is split in two vertices v^I and v^{II}

Figure 4.1: Topological differences between *VD(P)* and *VD(S)*

Chapter 5

Overview of *VD(S)* algorithms

A schematic top-level view to the problem of *VD(S)* computation is shown on Fig. 5.1. An algorithm gets a set of generators and computes the topology of their Voronoi diagram. From the topology, another algorithm can be used to compute the geometry of Voronoi edges and faces. The problem of topology computation can be solved in quite different ways by different algorithms. The computation of *VD(S)* geometry is separated from them and can be reused. Furthermore, for the computation of topology, only the geometry of Voronoi vertices is required.

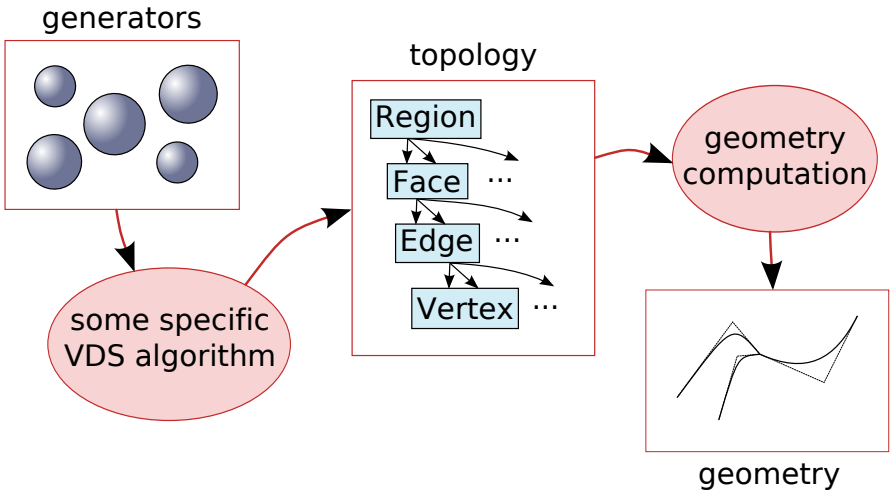


Figure 5.1: Schematic view to *VD(S)* computation

In the following sections, we briefly summarize some algorithms for the construction of *VD(S)* topology. Details to the computation of Voronoi vertices and edges geometry can be found in Chap. 6, details to the topology computation and representations are in Chap. 7.

5.1 Edge tracing

This algorithm constructs the diagram as a graph $G = (V, E)$ of Voronoi edges E over Voronoi vertices V. Starting from an initial vertex v_0, it traces four edges emanating from v_0 in order to find their end-vertices. When an end-vertex of an edge is found, three new edges can be traced. This process repeats as long as there are some edges to be traced. When it finishes, it returns the graph G of vertices and edges. Note that G is just a subgraph of the entire diagram. Edge tracing can not find elliptic edges and isolated subgraphs.

Lets n be the number of generators and $m = |E|$ be the number of edges in G. Without any optimization, edge-tracing has $O(nm)$ time complexity.

There are more details to this algorithm in Sec. 7.1.

5.2 Face tracing

This is in fact the edge-tracing algorithm but reinterpreted in the dual space. Its main benefit is the data structure it uses to represent the topology.

It deals with a quasi-triangulation *QT(S)* by using an *inter-world data structure* [11], abbreviated as *IWDS*[1]. The structure is similar to the well-known Delaunay triangulation for points *DT(P)* but it can handle anomalies that can occur in *VD(S)*. Instead of explicitly representing Voronoi regions, faces, edges and vertices, it deals with the corresponding vertices, edges, faces and cells in the dual space. A vertex in the dual space is represented as a simple tetrahedron and the remaining elements are represented implicitly, *i.e.* they can be derived from the tetrahedron. Furthermore, the concept of a *big-world* and *small-worlds* is used to deal with disconnected subgraphs. Big world represents the main triangulation and small worlds the disconnected subgraphs. These worlds are disconnected in the sense of Voronoi edges but connected in the sense of Voronoi faces. An array of *gates* is used to represent this relationship. A gate is the entry from a world to another world via a Voronoi face (or edge in the dual space).

The algorithm first discovers the big-world as an ordinary edge-tracing would do. Then it finds isolated generators and identifies the elliptic edges, or constructs small-worlds from them in the similar manner as the big-world. These small-worlds are then connected with their neighboring worlds via an array of gates.

[1] there is also *eIWDS* [16] which represents edges and faces explicitly – *e* stands for *extended*

5.3 Region expansion

Recall that *VD(P)* $\subset$ *VD(S)*. For a set of generator spheres, *VD(P)* = *VD(S)* when radii are equal, *e.g.* zero.

The *region expansion* algorithm [13, 12] benefits from an observation that the Voronoi diagram of generator centers, which is a *VD(P)*, often is topologically very close to the *VD(S)* we are looking for. The algorithm expects that a diagram of generator centers is already computed – it can be obtained by an arbitrary algorithm for *VD(P)*. Then it expands generators one-by-one in their radii in order to obtain the entire *VD(S)*. Local changes in the topology can occur during an expansion. The algorithm drives the expansion of a region trough these topological changes (it uses an event-based approach) and updates the topology in order to keep the diagram a valid *VD(S)*. Region expansion could be considered as a top-down algorithm – it works on the entire hierarchy from regions to vertices.

A 2-dimensional analogy of an expanded region is shown on Fig. 5.2. In most cases, the topology did not change. Note that only one edge has disappeared as well as its start and end vertices, and a new one just originated together with two new vertices.

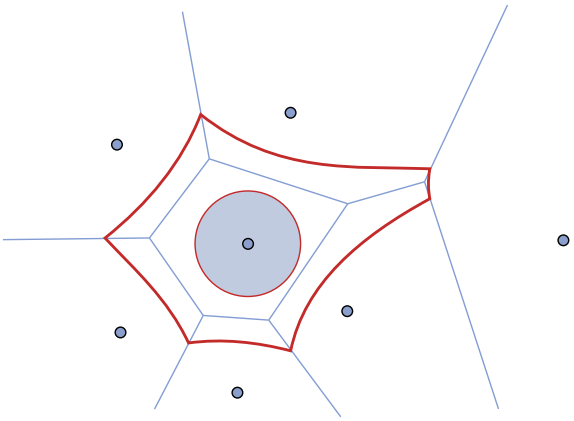


Figure 5.2: Expanding region

Pros and cons

Because the algorithm starts from a *VD(P)*, which is expected to be very close to the desired *VD(S)*, the main benefit is the running time – with a little bit of imagination, it could be looked at as if the linear part was computed by an efficient algorithm for *VD(P)* and the quadratic part by expanding regions. Furthermore, topological

changes involving elliptic edges are calculated in a similar way as changes involving non-elliptic edges.

The main drawback could be the requirement of an efficient and easily modifiable hierarchical structure with many (six) levels of hierarchy and the numerical calculation involved, such as finding roots of a quartic polynomial with coefficients that has been already burdened by some non-trivial calculation.

5.4 Selected solution

From the previously mentioned algorithms, we have implemented the *edge-tracing* algorithm. In our implementation, we *do not* address issues regarding the numerical stability. We *do not* search for isolated subgraphs, *ignore* elliptic edges (they are rare in proteins anyway) and *did not optimize* the algorithm in the searching for end-vertices. The data structure for a diagram is the naïve one as it is shown on Fig. 7.1. The idea was to get some diagrams in an easy way and try to experiment with them, instead of dealing with sophisticated data structures, numerical stability, running time, etc. Furthermore, this allowed us to implement our new idea of finding an initial vertex for edge-tracing. The algorithm is described in Sec. 7.1 and its implementation is elaborated into more detail in Sec. 8.6.

Although region-expansion should be significantly faster, it requires a heavy topological structure (see Sec. 7.2.1) and involves more complex numerical calculations - we could not afford to avoid issues regarding the numerical stability in the calculation of topological events (finding roots of a quartic polynomial).

Face tracing would be probably a better choice than edge-tracing. It would mean to use the inter-world data structure (see Sec. 7.2.2) for topology representation. This would not improve the time complexity but it would provide us complete diagrams (as a quasi-triangulation).

Chapter 6

Geometry computation

In this chapter, we briefly summarize how to compute the geometry of Voronoi vertices and edges as it is described in [9].

In the case of edge geometry, we have to find the center of a circle inscribed to three circles which is The Apollonius 10^{th} Problem. In the calculation of vertex geometry, we need to find the center of a sphere inscribed to four generator spheres. This is the same problem but formulated in a higher dimension (we are given four spheres in 3D instead of three circles in 2D). Gavrilova and Rokne described a solution for the general n-dimensional case in [6]. We use their approach in the following sections.

6.1 Vertices

The geometry of a Voronoi vertex v is the center of an empty sphere $S(v)$ inscribed to four generator spheres. Let us denote them as s_1, s_2, s_3 and s_4 respectively. We want to find the center of $S(v)$.

Let us denote $s_i = (x_i, y_i, z_i, r_i)$, where (x_i, y_i, z_i) is the center and r_i is the radius of the sphere s_i. Without the loss of generality, we assume that s_4 is the sphere with the minimum radius.

At first, we transform s_i so that s_4 becomes the origin:

$$s_i := s_i - s_4$$

We will compute the tangent sphere (x, y, z, r) in this new system. The solution is

hidden in equations:

$$
\begin{aligned}
(x-x_1)^2+(y-y_1)^2+(z-z_1)^2 &= (r-r_1)^2 && (6.1)\\
(x-x_2)^2+(y-y_2)^2+(z-z_2)^2 &= (r-r_2)^2 && (6.2)\\
(x-x_3)^2+(y-y_3)^2+(z-z_3)^2 &= (r-r_3)^2 && (6.3)\\
x^2+y^2+z^2 &= r^2 && (6.4)
\end{aligned}
$$

We expand Eq. 6.1, 6.2 and 6.3, subtract Eq. 6.4 from them and get:

$$
\left(\begin{array}{cccc|c}
x_1 & y_1 & z_1 & r_1 & (x_1^2+y_1^2+z_1^2-r_1^2)/2\\
x_2 & y_2 & z_2 & r_2 & (x_2^2+y_2^2+z_2^2-r_2^2)/2\\
x_3 & y_3 & z_3 & r_3 & (x_3^2+y_3^2+z_3^2-r_3^2)/2
\end{array}\right) \qquad (6.5)
$$

This is a system of three linear equations in four variables x, y, z and r. We solve it in terms of *one of them* as a free variable. By substituting the result into Eq. 6.4, we get a quadratic equation in the free variable. We solve it for its real roots, get the remaining unknowns and transform them back to the original system by adding the s_4. We are interested only in spheres with real, non-negative radii.

Depending on configuration of the four generators, we can end up with *none*, *one*, *two* or *infinite* number of tangent spheres [6].

How to choose a free variable

We want to mention here that the choice of a free variable in 6.5 is important. In other words, solving the system in terms of r all the time is not as wise as it is convenient.

Let D_r be the determinant of a linear system in unknown variables x, y, z and r defined as

$$
D_r = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}
$$

Imagine the case when centers of three generators are co-linear or nearly co-linear (the vertex belongs to an edge that is almost a circle) and we choose r to be the free variable. Then the determinant D_r will be zero or almost zero and the solution will be therefore lost or we will run into numeric issues.

We propose here to select a variable $q \in \{x, y, z, r\}$ to be free when it maximizes the absolute value of the determinant D_q after normalization of its columns. In other words, we want the most orthogonal system.

6.2 Edges

Because a Voronoi edge is always a conic section, it can be represented as a *quadratic rational Bézier curve*

$$\mathbf{B}(t) = \frac{w_0(1-t)^2 P_0 + 2w_1(1-t)tP_1 + w_2 t^2 P_2}{w_0(1-t)^2 + 2w_1(1-t)t + w_2 t^2}, \tag{6.6}$$

where P_0, P_1 and P_2 are the control points, w_1, w_2 and w_3 are their weights.

Suppose that we are given a conic section. We can represent it as a quadratic rational Bezier curve in the form given by Eq. 6.6 if we know these five parameters: *two different points*, *tangent vectors at both these points* and a *passing point* (another point on the conic section) [5, Section 12.6].

From this information, we are able to calculate coordinates of the middle control point P_1 and its weight w_1. The points we know tangent vectors for, they define the first control point P_0 and the last control point P_2 with their weights set to 1.

This is illustrated on Fig. 6.1. The five parameters of an edge e are shown on Fig. 6.1a. The middle control point P_1 can be calculated easily from them. Note that the passing point Q lies on the opposite edge segment e'. In Fig. 6.1b, several modifications were made to the weight $w_1 > 0$ of the middle control point P_1. There holds $0 = w_1^{III} < w_1 < w_1^I < w_1^{II} = 1$. Weights $w_1^{VI} < w_1^V < w_1^{IV}$ are all negative. Furthermore, $w_1^V = -w_1$ and this weight defines e'.

When we get a Voronoi edge bounded by two vertices, the position of the start vertex becomes P_0, the position of the end vertex becomes P_2 and $w_0 = w_2 = 1$. The problem is to calculate coordinates of P_1 and the weight w_1.

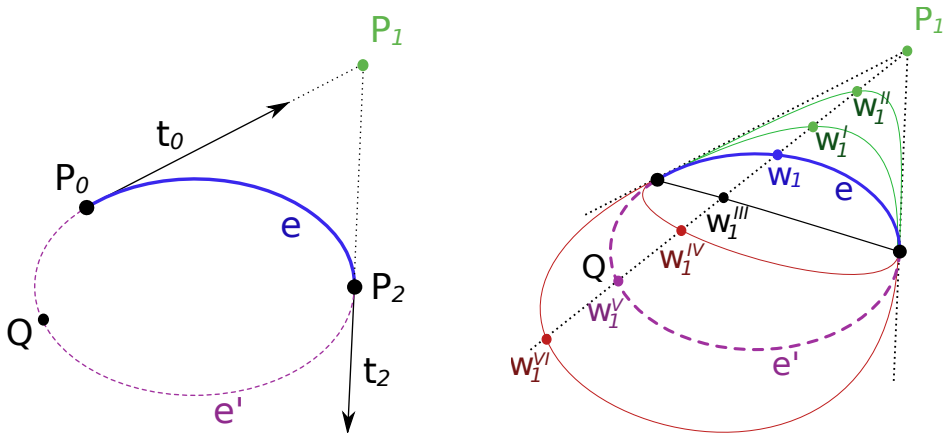
Coordinates of P_1 can be calculated as the intersection of two rays passing through P_0 and P_1 in the direction of the respective tangent vectors as it is shown on Fig. 6.1a. The weight w_1 is then calculated from barycentric coordinates (τ_0, τ_1, τ_2) of the passing point[1] with respect to the triangle (P_0, P_1, P_2) as

$$w_1 = \frac{\tau_1}{2\sqrt{\tau_0 \tau_2}} \tag{6.7}$$

Detailed explanation to Eq. 6.7 including its derivation can be found in [5, Sec. 12.5].

The sign of w_1 has to be reversed if the passing point does not lie on a valid edge segment, *i.e.* when its angular distance is smaller than the distance of the end-vertex. Otherwise, we would end up with a curve representing the opposite segment of the conic section as it is shown on Fig. 6.1b – there we get w_1^V from the passing point Q and we need to revert its sign in order to obtain w_1 because $Q \in e'$.

[1] they do not need to be normalized; $\tau_0 \tau_2 > 0$ since the passing point is on the curve



(a) Five parameters: end-points P_0 and P_1, their tangents t_0, t_1 and a passing point Q

(b) The role of the weight w_1

Figure 6.1: Edge e as an elliptic segment and its parameters

A passing point can be obtained by this procedure:

1. project three generators of the edge to a plane passing through their centers
2. find the center of a circle inscribed to the three projected generators[2]
3. transform the center back to the original system

Computation of a passing point p is shown on Fig. 6.2. There is an edge e defined by spheres s_1, s_2 and s_3. The spheres are projected to the plane passing trough their centers and p is found as the center of an empty tangent circle.

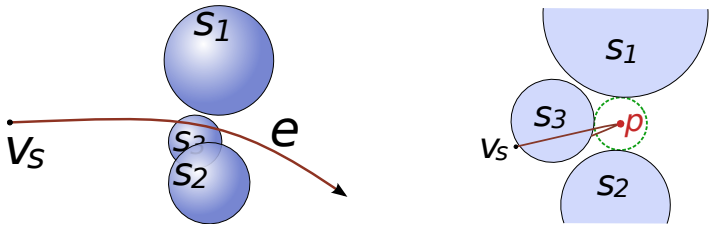


Figure 6.2: Finding a passing point p

The tangent vector at a vertex v can be calculated as an angle trisector from its position to the centers of three gate balls defining the edge. Recall that the vertex

[2]this can be done the same way as we did in Sec. 6.1

sphere $s(v)$ is tangent to the gate balls at some tangent points tp_1, tp_2 and tp_3. The tangent vector is the normal of the plane passing through tp_1, tp_2 and tp_3.

Although there is no need for the tangent vector to be in any particular orientation for the calculation of P_1, it is useful to have it oriented in the direction of the edge for purposes of angular distance comparison.

Rendering of 3-dimensional quadratic rational Bezier curves may seem to be problematic. Sec. 8.6.2 describes how to render them as ordinary 4-dimensional Bezier curves.

Chapter 7

Topology construction and representation

VD(S) topology is shown on Fig. 7.1: A diagram consists of topological elements - regions, faces, edges and vertices. They form a topological structure over a set of generators. Every topological element is defined by a specific number of generators (Gn is an n-tuple of references to generators). For example, every edge is defined by exactly three generators. Hierarchical relations are represented as incidence/boundary links. For example, a vertex has always four incident edges. An edge can be bounded by two vertices. The first vertex is the start vertex. Lets call vertices and edges as *lower topology* and regions with faces as *higher topology*.

Geometry references are shown as well. For the computation of topology, it is inevitable to maintain only the geometry of generators and vertices.

Fig. 7.1 shows only a very rough model which can be implemented in different ways. It does not show how to represent relations among elements in the same topological level (topological holes in faces or incidence ordering among edges, for instance). Different algorithms prefer different model implementation – edge tracing runs on lower topology, region expansion maintains the whole diagram and face-tracing does not use it at all (it works with a dual representation).

Sec. 7.1 describes a slightly modified version of the edge-tracing algorithm [9] for the construction of *VD(S)* topology – we present a new approach to finding an initial vertex and slightly modify the search for end-vertices. Sec. 7.2 describes data structures for topology representation.

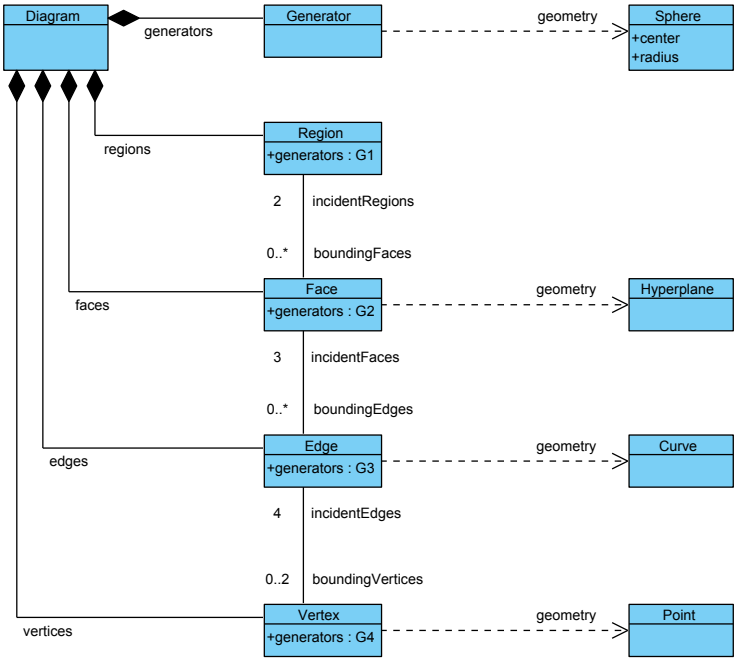


Figure 7.1: Basic VD(S) model - topology and geometry

7.1 Edge tracing

The idea of this algorithm is quite simple. Given a vertex, there are always four edges incident to the vertex. The algorithm traces these edges in order to discover their end vertices. When a new vertex is found, three more edges can be traced.

This can be interpreted as a bottom-up approach (from vertices to edges). The algorithm runs on the lower topology. It constructs a graph $G = (V, E)$, where V is a set of Voronoi vertices and E is a set of Voronoi edges. The higher topology can be constructed from the lower topology without much effort.

Tracing edges from an initial vertex gives us a maximal set of vertices reachable from the vertex and all incident edges. Note that this is just a subgraph of the whole lower topology in general! When two graphs are connected only by faces, there is *no way* how to get from the first graph to the second by tracing edges. Even if the second graph would be reachable, there can still be elliptic edges that are not reachable, since they do not have any vertex.

Algorithm 1: Edge tracing

Data: Set of generator spheres.
Result: Graph $G = (V, E)$ of vertices and edges, $G \subset VD(S)$

```
1  v_0 ← find an initial Voronoi vertex
2  Push 4 edges emanating from v_0 into a stack (v_0 is their start vertex)
3  while stack is not empty do
4      e ← stack.pop()
5      if e is already computed then
6          ignore e (goto 4)
7      Find the end vertex v_end of e by computing an empty sphere tangent to
       three gate balls and a fourth generator. If it is already computed, use that
       one instead of creating a new one to complete the edge definition.
8      Push 3 edges emanating from v_end into the stack.
9  end
10 return (vertices, edges)
```

7.1.1 Finding the very first Voronoi vertex

Algo. 1 can search for Voronoi vertices but it needs an initial vertex to start from. In [9], they presented two approaches how to find the very first Voronoi vertex:

Naïve brute-force For each quadruple of generators, compute a tangent sphere. If the sphere is empty, a Voronoi vertex has been found. This is $O(n^5)$ and therefore not recommended.

Explicit vertex workaround Create an explicit initial vertex by adding four additional generators to the original set. Run the algorithm and wait until it discovers an end vertex defined by four *original* generators. Then restart the algorithm, this time with the known initial vertex.

The second approach does not change the asymptotic time complexity of the algorithm, however some questions would come to the mind:
Can we just add another four generators to the given array, even for a while? How big and how far from each other should they be, *i.e.* how to choose the radius of the vertex sphere? Where to put them? We want to find some vertex and do not want

to violate the assumption that any sphere can not be fully contained in another. Rather than answering these questions, we have devised another algorithm:

Near vertex search algorithm Algo. 2 first locates the nearest face of some region. Then it locates the nearest edge on the face and the nearest vertex on the edge. The distance is measured as a radius of an empty tangent sphere.

Algorithm 2: Near vertex search

Data: Set of generator spheres G.
Result: A Voronoi vertex as (g_0, g_1, g_2, g_3, sv), where g_0, g_1, g_2 and g_3 are its defining generators and sv is the vertex sphere.

```
1  begin
       // for some region
2      for g_0 ∈ G do
           // find the nearest face of the region
3          g_1 ← g ∈ G \ {g_0} that minimizes MTS(g_0, g).
           // find the nearest edge of the face
4          g_2 ← g ∈ G \ {g_0, g_1} that minimizes MTS(g_0, g_1, g).
           // find the nearest vertex of the edge
5          g_3 ← g ∈ G \ {g_0, g_1, g_2} that minimizes MTS(g_0, g_1, g_2, g).
6          if g_0, g_1, g_2, and g_3 define a vertex then
7              return (g_0, g_1, g_2, g_3, sv)
8      end
9      Fail, since no vertex has been found.
10 end
   /* MTS(g_0, ..., g_{n-1}) is the minimal tangent sphere for the given
      set of n generators, in the sense of its radius.                  */
```

The problem of finding MTS in E^3 for given $n \leq 4$ can be solved as an $(n-1)$-dimensional sub-problem. For $n = 2$, it reduces to the problem of finding the nearest neighbor. For $n = 3$, it can be solved in a plane defined by centers of the three generators and it is the problem of finding a circle inscribed to three another circles. For $n = 4$, it is the problem of finding a sphere inscribed to four other spheres.

Time complexity	
Worst case	Expected
$O(n^2)$	$O(n)$ or even $O(1)$ if optimized

Finding the nearest face, edge and vertex is $O(n)$, but it could be optimized in the same way as the edge tracing algorithm (see [9, 4]). In the worst case, algorithm 2 iterates $O(n)$ times to find a good region where a vertex can be found, thus we end up with $O(n^2)$. Note that *it does not have to find any vertex* even when there is some in the true diagram.

These worst cases occur when all regions – or at least most of them – have their nearest faces unbounded or with "wrong" nearest edges (even unbounded or elliptic).

Except the elliptic edges, this could be avoided if the entire set of generators would be closed into an outer tetrahedron (all the previously opened regions become closed). We hope that elliptic edges can be safely ignored, because edge tracing could not find them anyway.

On the other hand, we could argue that the algorithm finds the proper region in $O(1)$ in most cases (elliptic edges are rare, especially on nearest faces). This gives us expected $O(n)$ or even $O(1)$ if it would be optimized in the same way as edge tracing.

Why and how algorithm 2 works It turned out that we have used the nearest-neighbor property from additively weighted Voronoi diagrams: If a sphere $s_i \in S$ is the nearest neighbor of $s_j \in S$, then Euclidean regions of s_i and s_j have a common facet. See Gavrilova's Ph.D. thesis [7, part 11] for details to this theorem and its proof. The theorem is applied to g_0 in the first step of the algorithm as the search for the nearest face. This gives us two regions in terms of their defining generators g_0 and g_1. Then, the theorem is applied from both g_0 and g_1 at the same time in order to obtain g_2. Generators g_0, g_1 and g_2 define the nearest edge in the context of the face given by g_0 and g_1. The same rule is used to find a vertex, *i.e.* using the theorem from three generators at the same time to find the fourth that defines the vertex.

This can be viewed in another way. Suppose, that we are given a sphere that is flexible in its radius. We put this sphere on the hull of a generator g_0, set its radius to zero and let it grow but it has to stay empty and tangent to all generators it touches. At first, it rolls over the first generator at which we have put it (g_0) and it can grow, because it stays empty. At some moment, it gets big enough and touch another generator g_1. Since then, it can not roll over g_0 any more, since it has to stay tangent to both g_0 and g_1. All it can possibly do is to rotate around the axis of their centers. So it rotates and grows. At some moment, it touches another generator g_3 and has to grow on the edge trajectory until it reaches the fourth generator g_4 defining a vertex. Because the trajectory of the sphere is empty and the sphere is tangent to four generators, a Voronoi vertex was just found.

7.1.2 Searching for end vertices

To trace an edge, we are given its start vertex v_s, three gate balls defining the trajectory (bg_1, bg_2, bg_3) and the fourth generator b_s that completes the definition of the start vertex and determines the direction of the trajectory.

The task is to find the next vertex v_e on the edge in the given direction. This finishes the edge and v_e becomes its end vertex. The task reduces to find only one generator b_e and the center of the vertex sphere $S(v_e)$, because the remaining generators defining v_e are bg_1, bg_2 and bg_3. The ordering is given by the angular distance (viz. Sec. 3.4).

We could imagine that we are given a vertex sphere $S(v_s)$. It is empty and touches the four generators defining v_s. Then we push this sphere in the edge direction. We want it to remain empty and touch only the edge generators, hence we allow it to be flexible in its radius. When it hits a fourth generator, it gives us the end vertex.

An example of finding the end vertex is shown on Fig. 7.2. We trace an edge defined by gate balls (bg_1, bg_2, bg_3). We start at the known start vertex v_s and select candidates from all generators except the three gate balls. The generator b_i would define a vertex v_i but it does not, since its tangent sphere S_i is not empty – it is intersected by another generator b_j. The generator b_j would define a vertex v_j but it does not. In this case, the tangent sphere s_j is empty, but there is another generator b_e that defines an empty tangent sphere which closer to v_s in terms of the angular distance θ. Because there is no other tangent sphere closer to v_s, b_e defines the end vertex v_e and finishes the edge e.

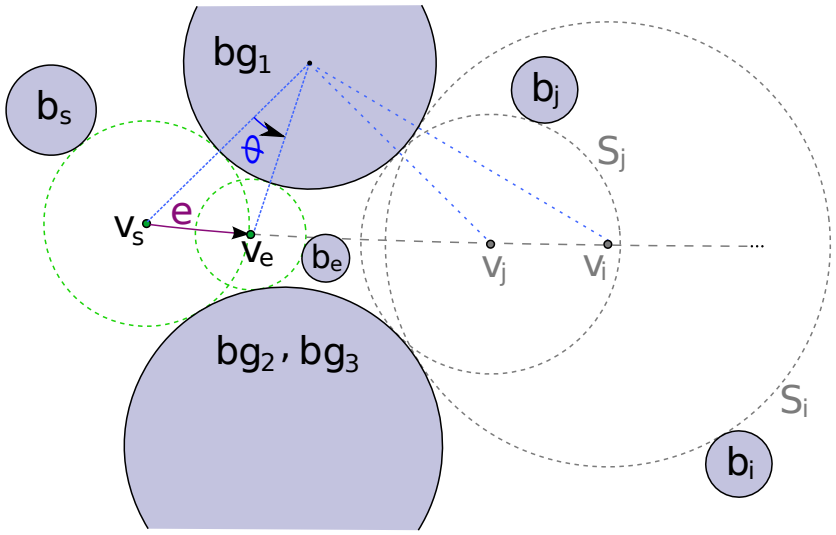


Figure 7.2: Finding the end vertex

The algorithm presented in [9] *almost* works. It will run into problems with unbounded edges leading to the infinity. In this case, it could incorrectly choose the end vertex from the prohibited part of the edge. A simple workaround can avoid this problem and we present the improved version here as Algo. 3.

Algorithm 3: Find end vertex

Data: Start vertex v_s defined by four generators bg_1, bg_2, bg_3 and b_s.The edge trajectory is defined by gate balls (bg_1, bg_2, bg_3). K is the set of candidate generators (all generators except the gate balls) and s_∞ is an imaginary infinite generator. $S(v_s)$ is the vertex sphere of v_s.

Result: A pair (S, b), where S is a sphere and $b \in K \cup \{b_\infty\}$. If $b = b_\infty$, the edge leads to infinity. Otherwise, bg_1, bg_2, bg_3 and b_s define an end vertex v_e and the center of S can be directly used as its position.

`// select the first candidate`

if (bg_1, bg_2, bg_3) *define an elliptic edge* **then**
    $b_1 \leftarrow b \in K$
else
    $b_1 \leftarrow b_\infty$
$S_1 \leftarrow$ tangent sphere from bg_1, bg_2, bg_3 and b_1, such that $S_1 \neq S(v_s)$. If there are more solutions to S_1, prefer the one with the minimal angular distance from v_s.

`// minimize the angular distance`

$(S, b) \leftarrow (S_1, b_1)$.
foreach $b_k \in K \setminus \{b_1\}$ **do**
    $S_k \leftarrow$ tangent sphere from bg_1, bg_2, bg_3 and b_k.
    if S_k *is closer to* v_s *than* S **then**
        $(S, b) \leftarrow (S_k, b_k)$
end
return (S, b)

A tangent sphere from bg_1, bg_2, bg_3 and b_∞ is the common supporting plane of bg_1, bg_2, bg_3. In the angular comparison, we use its normal vector. This precaution establishes an upper bound to the valid angular distance, avoiding prohibited and unreachable. See Sec. 3.4 for more details.

7.1.3 Checking if a vertex is already computed

When an end vertex is computed, it is necessary to find out if it already exists, since existing vertices have some topological information associated with them. To achieve this, a dictionary of computed vertices can be used, called as VIDIC (Vertex Index Dictionary).

A vertex is defined by four generators. We can represent them as a quadruple of sorted indices into an array of generators[1]. However, this quadruple *does not* identify the vertex uniquely. An additional information must be added to make a unique key. It could be a flag that says if vertex coordinates are from the first or from the second solution of Eq. 6.4 or the coordinates directly.

The ordering of generators in the computation of vertex coordinates is important – different ordering would give us a slightly different results due to floating point precision loss.

7.1.4 Time complexity

The tracing-loop is iterated as many times as the number of edges, which is $O(n^2)$ in the worst case and $O(n)$ on the average, where n is the number of generators. Finding an end vertex takes $O(n)$ steps. Checking if the end vertex is already computed takes expected $O(1)$ when a hash table is employed. Therefore, the expected running time is $O(n^2)$.

The expected running time can be greatly reduced if some kind of geometric filtering is used in the searching for end vertices. See [9, 4] for details.

[1] $G4$ in Fig. 7.1

7.2 Data structures

For *VD(P)* in E^3, there are two common ways of its topology representation:

(a) *directly* as the hierarchy of regions, faces, edges and vertices

(b) *indirectly* by using a dual structure – Delaunay tetrahedralization *DT*

DT is often preferred over the direct representation of *VD(P)*, because it introduces only one level of hierarchy – an array of tetrahedra. It is a very compact representation. Each tetrahedron corresponds to a Voronoi vertex – it consists of four generator indices defining the vertex and four links to neighboring tetrahedra. These links explicitly represent Voronoi edges in loops bounding Voronoi faces. The higher topology of the diagram can be obtained easily.

Unfortunately, *VD(S)* is a non-manifold that can have elliptic edges and holes in faces as it was shown on Fig. 4.1. Its vertices, edges and faces do not have to be defined uniquely by their generators. The dual of *VD(S) is not* a *DT*, but a quasi-triangulation [11, 12].

The topology of a *VD(S)* could be directly represented by an appropriate structure for non-manifold models, such as *RES*[2] or *PES*[3] [14]. However, these structures are designed to handle a general case and thus too complex. *PES* consists of 10 levels of hierarchy, for instance, and it has several pointers at each level. Note that *VD(S)* is not so general. Therefore, these structures can be simplified in the problem domain.

A simplified data structure based on RES was introduced by Cho and Kim in [3]. The structure consists of only 6 levels of hierarchy. An *IWDS*[4] for storing quasi-triangulation was introduced later on [11]. These are briefly summarized in following two chapters.

In our implementation of edge-tracing, we did not use any of them and rather used a naïve model as it is shown on 7.1. It is more simple than the previously mentioned approaches, edge-tracing runs on the lower topology and the higher topology can be built separately. We want to mention here that *IWDS* could be a better choice.

7.2.1 Simplified radial edge structure

The model is shown on Fig. 7.3. There still are regions, faces, edges and vertices, but also two supplementary entities: an *edge-loop* and a *partial-edge*.

[2] radial-edge structure
[3] partial-entity structure
[4] inter-world data structure

- A region is bounded by a number of faces and each face is incident to exactly two regions.
- A face can have an arbitrary number of edge-loops. The first one represents the outer boundary and the following represent topological holes in the face.
- A loop consists of partial edges – it keeps just a single reference to one of them.
- Partial edge is a supplementary structure for an edge. It references the edge and contains its state in the context of a loop (*i.e.* in the context of a face). Partial edge represents two cycles – a *loop-cycle* and a *radial-cycle*. The first one realizes the corresponding edge-loop and the second loop connects all the three partial edges around their common referenced edge.
- An edge contains two references to its vertices (*i.e.* its start- and end-vertex).
- A vertex references all its four incident edges.

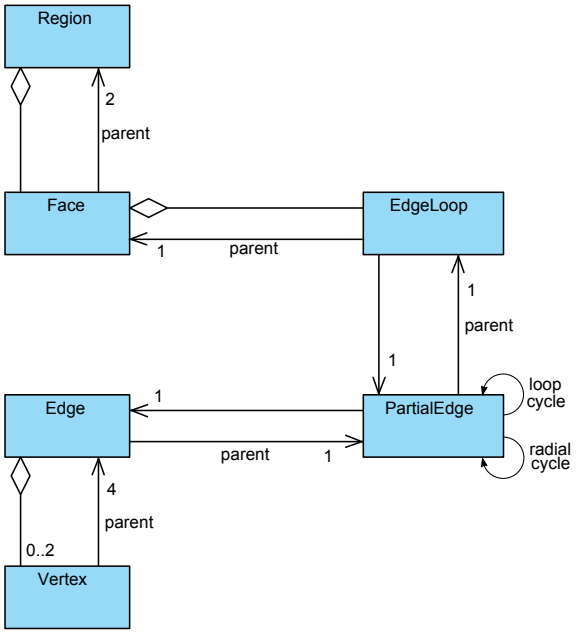
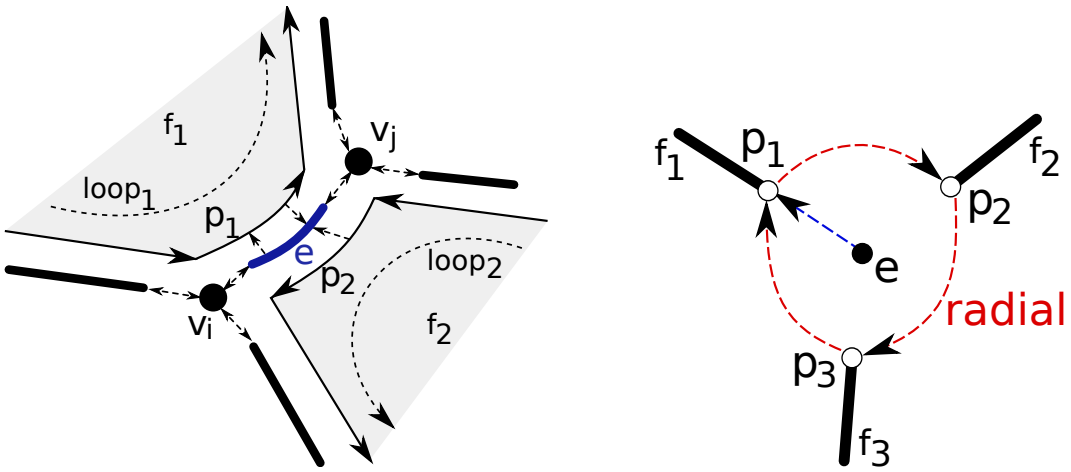


Figure 7.3: Simplified radial edge structure

Fig. 7.4 shows partial edges and their role in a loop. On Fig. 7.4a, there is an edge e between two vertices v_i and v_j. The edge references both these vertices and one of its partial edges – p_1. Note that there are three partial edges p_1, p_2 and p_3 around e but only the first two are shown on Fig. 7.4a. The partial edge p_1 is part of a loop cycle denoted as $loop_1$. The loop belongs to the face f_1. Similarly, partial edge p_2 is

part of a loop cycle $loop_2$ of the face f_2. Fig. 7.4b shows partial edges p_1, p_2 and p_3 around an edge e in a radial cycle. The edge references only one of its partial edges. Remaining partial edges can be found by searching through this cycle.



(a) Partial edges in loop-cycles (f_3 is not shown) (b) Partial edges in a radial-cycle

Figure 7.4: Partial edges and loops

References to defining generators

Even if it would be possible to explicitly store the references to the defining generators at each level of the hierarchy as it is shown on Fig. 7.1, it should be avoided because it brings more redundancy to the model. For this purpose, we classify topological elements by the number of defining generators into G_n classes and introduce topological derivation and integration as a way of inferring among these classes.

Lets denote G_n as an n-tuple of indices to generators. If an element is defined by exactly n generators, it belongs to the G_n class.

Voronoi vertices belong to G_4, edges to G_3, faces to G_2 and regions to G_1. Furthermore, they can be ordered by some global ordering, *e.g.* ascending ordering by generator indices.

Lets define a *topological derivation* as the process of generating an instance of G_n from an instance of G_{n+1} by leaving out one reference from the $(n+1)$-tuple at the specified *derivation index* $i \in \{1 \dots n+1\}$. Lets define the opposite process of assembling an instance G_n from an instance of G_{n-1} as a *topological integration*.

Topological derivation does not change the ordering of components but it needs an index to derive by. Topological integration does not need any index to be specified

explicitly but it has to infer it from the ordering of components. An example of topological derivation and integration is shown on Fig. 7.5.

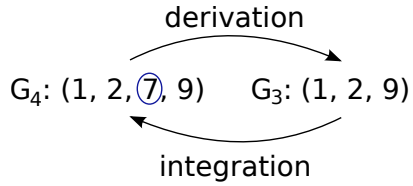


Figure 7.5: Topological derivation and integration

When the hierarchical structure of the topology is known, it is sufficient to store only G_1 at each region. Lower hierarchical levels can integrate their G_n from incident regions by following the parent-links. This is compact for the entire model and convenient for the upper topology. But it is less convenient for the lower topology. Topological integration is more complex than topological derivation – It has to keep the ordering among the components of G_n. Furthermore, when we integrate over several hierarchical levels, we get more fields $m > n$ than we need for a G_n instance, because incident elements share some generators. Superfluous fields needs to be removed.

It is also possible to keep G_4 at vertices and use a topological derivation for getting G_n at higher hierarchical levels. This is not so compact for the entire model but it is very convenient for the lower hierarchy. Furthermore, topological derivation is easy – it keeps the ordering among the components of G_n and it just needs an index to derive by. The index can be implicitly represented by the ordering of parent-links. Unfortunately, there is a problem with elliptic edges. They have no vertex from which a G_3 could be obtained. This could be handled by adding an artificial vertex.

Both approaches have their tradeoffs. Keeping G_1 at regions is be better suited for region-expansion, keeping G_4 at vertices is better for edge-tracing.

7.2.2 Inter-world data structure

The dual structure for *VD(S)* is a quasi-triangulation[5] *QT(S)*. There are three anomalies (multiplicity, degeneracy and singularity anomaly) that make it difficult to store it as a simple array of tetrahedra as it is common for the Delaunay triangulation of points in E^3. An *inter-world data structure* [11], abbreviated as *IWDS*, can handle these anomalies. It benefits from the observation that the occurrence of singularity anomalies is quite rare for real data, such as proteins.

[5] see Chap. 3.5

Suppose that we are given a set of spheres S containing an infinity generator s_∞ and we have created its Voronoi diagram. We could divide the lower topology of the diagram into several exclusive parts. Each part corresponds to a maximal connected set of Voronoi edges. These parts are called as *worlds* and they are connected only by Voronoi faces (edges in the dual representation). In order to represent this relation, *IWDS* employs a supplementary data structure called a *gate*. Gates connect worlds together and they overcome topological holes in Voronoi faces. For the ordering of worlds and gate orientation, please refer to [11].

IWDS model

Fig. 7.6 shows a model of *IWDS*. The data structure consists of three arrays. There is an array of generators representing vertices in the quasi-triangulation, an array of tetrahedra and there is also an array of gates. A tetrahedron is the basic building block. It references its four defining vertices and four neighboring tetrahedra. The references can be realized by indices to the appropriate arrays. A gate connects two worlds together. It is associated with an edge in the quasi-triangulation, implicitly represented as a reference to its two vertices. Furthermore, the gate has a reference to one of the tetrahedra from the source world sharing the edge and to one tetrahedron per one destination world sharing the edge.

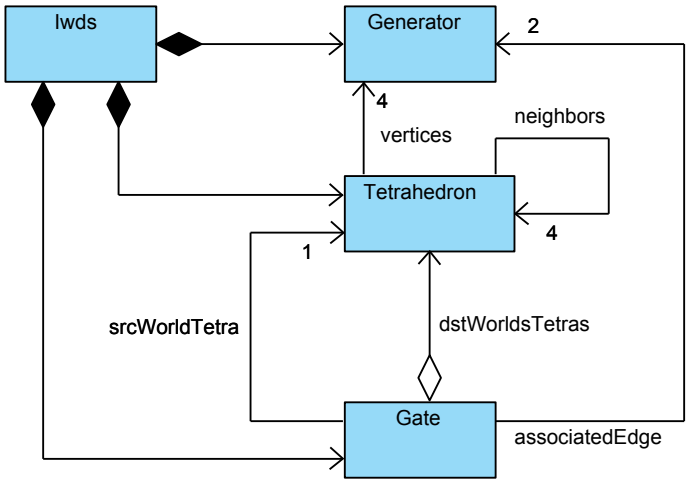


Figure 7.6: Inter-world data structure – UML model

Handling multiplicity anomaly

The case when two tetrahedra share two or more faces can be easily recognized by looking at the references to neighbors in such a tetrahedron. If there are two or more valid references that are equal, the tetrahedron causes the multiplicity anomaly.

Handling degeneracy anomaly

A tetrahedron degenerated to a triangle can be easily represented by invalidating one of its vertex references and all neighbors references (they are no more needed).

Handling singularity anomaly

A gate can be formally defined as $g = (\{s_i, s_j\} \rightarrow (\tau, \{\Delta_1, \Delta_2, \dots\}))$ where s_i and s_j are indices of two vertices defining an implicit edge in the quasi-triangulation, τ is the tetrahedron from the source world that has the implicit edge $\{s_i, s_j\}$ and Δ_k are the tetrahedra of destination worlds – one Δ per each world.

Note that the gate g connects exactly one source world to one or more destination worlds through tetrahedra that are part of the Voronoi face connecting these worlds.

The information that an edge in a tetrahedron is a gate from a source world to a destination world, *i.e.* that the tetrahedron is the τ-part of a gate, can be stored by increasing the indices of the respective vertices in the tetrahedron, such as $s_i \leftarrow s_i + |S|$ and $s_j \leftarrow s_j + |S|$.

The opposite direction, *i.e.* that an edge of a tetrahedron is a gate from a destination world back to the source world, *i.e.* that the tetrahedron is the Δ-part of a gate, can be stored by decreasing indices of the respective vertices in the tetrahedron, such as $s_i \leftarrow s_i - |S|$ and $s_j \leftarrow s_j - |S|$.

When a tetrahedra with modified vertex indices is encountered during the traversal through *IWDS*, it means that there is a gate to another world at some of its edges. The modification made to the vertex indices can be easily transformed back to the original system in order to get the real vertex indices.

Chapter 8

Implementation

8.1 Project structure

Fig. 8.1 shows a basic structure of the entire project. It consists of some libraries (*VdsLib* and *Algebra*), some external libraries and two applications (*VdsCmd* and *Viewer*). These all are in C#. There is also a simple conversion script (*Pdb2Txt*). Arrows symbolize dependencies among these parts.

VdsLib is a library for *VD(S)* computation. It is the most important part of the project. The library defines the structure of a Voronoi diagram, implements edge-tracing algorithm for its construction (Sec. 7.1), computes the geometry of Voronoi vertices and edges, and offers a basic serialization of the diagram.

Algebra is a basic algebraic library. It offers quaternions, two-, three- and four-dimensional vectors together with their algebraic operations. It can solve determinants, quadratic equations, etc.

VdsCmd is a command-line application. It reads an array of generators, computes their Voronoi diagram and serializes the output into an XML document.

Viewer is a GUI application for *VD(S)* visualisation. It reads a diagram from an XML document, creates a scene from it and displays the diagram. The user has a view to the scene managed by a camera. He or she can control the camera to change the view or select a filter to the diagram with respect to a selected generator.

Pdb2Txt is a simple conversion script for conversion of protein data from the PDB format to a simple text file. It is written in Visual Basic.

Log4Net is a logging library for the .NET platform.[1]

TaoFramework provides bindings of OpenGL, Cg, etc. from their native libraries to the .NET and Mono platforms.[2]

ShaderLoopControl is a component written in *C#* that makes easy to configure and use an OpenGL viewport. It is a non-standard library and we have made some minor modifications to it in order to serve our purposes.[3]

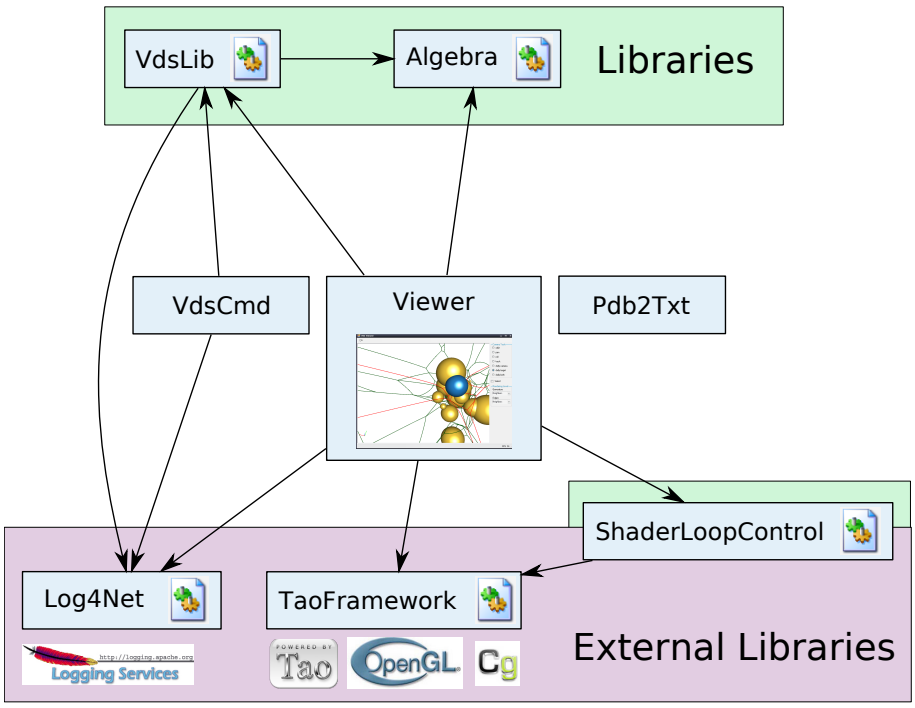


Figure 8.1: Project structure

8.2 Data flow

Fig. 8.2 shows the intended data-flow between applications. There is a PDB text file. These files can be downloaded from Protein Data Bank[4]. PDB file describes a protein molecule. All what the Pdb2Txt script does is that it reads the molecule

[1] `http://logging.apache.org/log4net/`
[2] `http://www.taoframework.com/`
[3] `http://cgg.ms.mff.cuni.cz/~marsalek/code/ShaderLoopControl.zip`
[4] `http://www.rcsb.org/`

and outputs each atom as a line with the position and atomic radius. Then VdsCmd comes into play, reads the text file as a set of spheres and with the help of VdsLib, it computes the Voronoi diagram and serializes it as an XML document. The document can be then read by Viewer and displayed to the user, or used by another application.

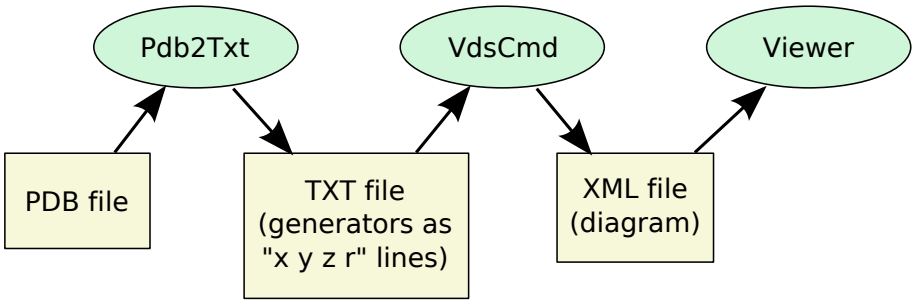


Figure 8.2: Dataflow between application

8.3 Implementation considerations

C# as the programming language

The most important part of our project is the library VdsLib. On the other hand, it would be incredibly hard to implement the library without any decent application that would display the resulting diagram (Viewer). We have decided to implement both these parts in C# rather than in C++. As a consequence, also Algebra and VdsCmd are in C# and we have used the specified external libraries.

It turned out that it is hard to make a reusable DLL library from a code written in C++, especially when C++ templates are used, such as STL or Boost. Even if it would be possible, we would need to write also a binding to the .Net platform because of Viewer (written in C#). Instead of dealing with these issues, we have decided for C# as the programming language, dropping the performance aspects of C++, but gaining better debugging ability, trouble-free library making, support for XML serialization and avoiding interoperability issues.

Restrictions

We have decided to implement edge-tracing algorithm without any support for elliptic edges and isolated sub-graphs, without any optimization for searching end-

vertices and without handling issues regarding the numerical stability of the algorithm (see Sec. 5.4). These issues could be addressed in the future.

Diagram serialization into XML

Instead of keeping the diagram in a plain text or binary file, we keep it as an XML document and describe its format in an XSD schema. This allows us to easily validate a file against the format. Furthermore, C# classes can be generated from the schema that makes serialization and deserialization more convenient. The price to pay is the greater size of the file and more time needed by an XML parser.

8.4 VdsLib structure

VdsLib is divided into several sections:

Algorithm contains the implementation of edge-tracing.

Geometry defines geometrical entities, such as sphere or Bezier curve. There is also an implementation of angular distance comparer, code for Voronoi vertices and edges geometry computation, geometric predicates, etc.

IO cares about reading generators from a text stream, serializing and de-serializing the diagram, its XSD schema and pre-generated classes.

Topology defines generators and the model of *VD(S)* topology.

8.5 Computing a diagram with VdsLib

An example of computing a Voronoi diagram for a set of spheres follows. A list of generators is read in the first step. Then an outer tetrahedron is computed and added to the list in order to bound infinite regions (and regions that are too large). This is achieved by computing an approximate bounding sphere of the original set of generators, magnifying its radius (1.3 times in our case) and computing the minimal equilateral tetrahedron that contains the sphere. Next step is creating an object representing the edge-tracing algorithm and defining the input set. After that, the construction of the diagram is started. It can run pretty long depending on the input size and it can even throw an exception caused by some numerical error. When the algorithm fails to find an initial vertex to start from, it returns false. Releasing

the diagram releases also internal structures kept by the algorithm. When the construction is finished, higher topology of faces and regions is created from the lower topology. As the last step, edge geometry is computed.

```
// get some generators
List<Generator> generators = ReadGenerators();

// add outer tetrahedron (four more generators, 130% bounding sphere)
Generator[] outerTetra;
Generator.ComputeOuterTetrahedron(generators, 1.30, out outerTetra);
generators.AddRange(outerTetra);


// compute Voronoi diagram of spheres
ETDiagram diagram = null;
EdgeTracing algo = new EdgeTracing();
algo.DefineGenerators(generators);
if (algo.ConstructDiagram())
   diagram = algo.ReleaseDiagram();

// create the topology for faces and regions
diagram.CreateHigherTopology();

// compute the geometry of edges
Factory geometryFactory = new Factory(diagram);
foreach (ETEdge edge in diagram.Edges)
   edge.Curve = geometryFactory.ProduceGeometry(edge);

// ... use diagram.Generators, Vertices, Edges, Faces, Regions ...
```

8.6 Implementation details

8.6.1 Edge-tracing algorithm

Follows our implementation of edge-tracing. This is more specific than Algo. 1 and the implementation underneath this top-level code is slightly different from the algorithm presented in [9]. For the representation of topology we use the naïve model from Fig. 7.1. We store generator indices at vertex level only and derive them at higher levels as it is shown on Fig. 7.5. Each vertex has four slots for incident edges and each edge references two vertices. Our algorithm builds only the lower topology.

```
// find the first vertex and push edge candidates
VertexCandidate tmp;
if (!FindFirstVertexCandidate(out tmp))
   return false;
ETVertex firstVertex = CreateNewVertex(tmp);
PushEdgeCandidates(firstVertex);

// trace edges
while (IsThereAnyEdgeCandidateToProcess)
{
   EdgeCandidate ec = PopEdgeCandidate();
   if (ec.IsEdgeAlreadyDefined)
      continue;

   VertexCandidate endVertexCandidate;
   if (!FindEndVertexCandidate(ec, out endVertexCandidate))
      continue;

   // get the existing end-vertex or create a new one if not found
   ETVertex endVertex = FindExistingVertex(endVertexCandidate);
   if (endVertex == null)
      endVertex = CreateNewVertex(endVertexCandidate);

   DefineEdge(ec, endVertex);
   PushEdgeCandidates(endVertex);
}

// diagram construction is finished
return true;
```

The algorithm uses vertex and edge candidates. These are simple structures[5] which may eventually become real vertices or edges. A candidate may represent an element that already exists in the diagram. A vertex candidate may represent a configuration of four spheres that will never become a vertex. Edge candidates reference their start vertices but they do not have any end-vertex yet.
Recall Fig. 7.2 – there is a real vertex v_s and some edge e that used to be an edge candidate. Three vertex candidates v_i, v_j and v_e were compared, v_e became the end-vertex and defined the edge.

At first, the algorithm finds some candidate for the very first Voronoi vertex. It uses our approach described by Algo. 2. Four edge candidates are stored for tracing and edge tracing loop begins. In our implementation we use a queue of edge candidates instead of an edge-stack. We believe that a breadth-first discovery will define edges earlier than a depth-first approach.

[5] C# structures are value-types – they are allocated on the stack rather than on the heap

In the loop, some old edge candidates are ignored – their edges are already defined. This is achieved by looking at incident edges of the start-vertex – when the corresponding edge slot is not empty, the edge already exists and the candidate has to be ignored.

From the edge candidate, a search for an end-vertex candidate is performed. It is implemented as a linear search through almost all generators and hence it is the main performance bottleneck and the source of numerical stability issues. The end-vertex candidate does not have to be found. This is the case when an edge goes to the infinity and such edges are ignored. We use Algo. 3 for searching end vertices.

To determine if the end-vertex candidate represents an existing vertex, we use a dictionary – VIDIC. Our dictionary maps four generator indices to a pair of vertices. Indices are sorted by the ascending ordering. If there is an entry for the given candidate, we compare their positions for equality. Note that the coordinates are floating-point numbers and we test them for an exact match. This works because a vertex sphere is always computed from four generators in exactly the same way thanks to the ascending ordering of vertex generator indices.

When an existing vertex is found in VIDIC, it is used directly as the end-vertex. Otherwise, a new vertex is created, put into VIDIC and used as the end-vertex. After that, a new edge is defined from the edge candidate and it is bound to the start- and end-vertex. This is a possible source of errors: Imagine that a wrong end-vertex has been found due to some numerical error in angular distance comparison. The end-vertex may already have some edge in the corresponding slot. When the algorithm tries to bind a new edge to this slot, an exception is thrown, the algorithm fails an the user gets no diagram at all.

8.6.2 Visualization of Voronoi edges in OpenGL

Voronoi edges can be represented as quadratic rational Bezier curves. To render them, we could either write a specialized shader or we could try to use some evaluator which seems to be much more convenient. Unfortunately, OpenGL does not support rational Bezier curves.

The trick that will help us here is a conversion of a d-dimensional *rational Bezier* curve to a $(d+1)$-dimensional *ordinary Bezier* curve. In other words, we get a *rational Bezier*, put it into homogeneous coordinates and treat is as it was an *ordinary Bezier*.

In our case, we get a control point $P_i = (x_i, y_i, z_i)$ and its weight w_i. All we have to do is to send it as $(x_i w_i, y_i w_i, z_i w_i, w_i)$ and let OpenGL to treat is as an ordinary Bezier curve. The perspective division evaluates it to the rational form.

```
stride = 4, // 4D
order = 3;  // quadratic + 1

glEnable(GL_MAP1_VERTEX_4);
glMapGrid1f(numOfEdgeSegments, t0, t1);
glMap1f(GL_MAP1_VERTEX_4, t0, t1, stride, order, controlPoints4D);
glEvalMesh1(GL_LINE, 0, numOfEdgeSegments);
glDisable(GL_MAP1_VERTEX_4);
```

Chapter 9

Experiments and results

Different sets of spheres have different Voronoi diagrams. The topology as well as the geometry of a diagram depends on the distribution of spheres in a set in terms of their positions and radii. The resulting diagram can be very similar to the well-known Voronoi diagram of a point set. For some data, the diagram can be quite different. To investigate these cases, we have used our implementation of edge-tracing algorithm to find diagrams in some classes of input data and measured their characteristics. Recall that our implementation (Sec. 8) can find only a subgraph of the real diagram. It is quite slow and it does not care much about the numerical stability. We believe that it is sufficient for experiments and hence we have measured its characteristics as well.

9.1 Input data

We have focused on two classes of input data - *proteins* and *random spheres*.

9.1.1 Proteins

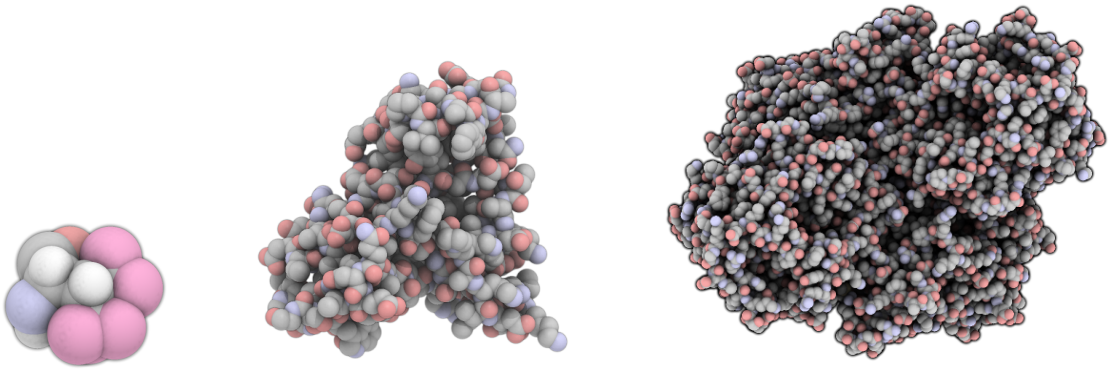
Proteins are large organic compounds made of chains of amino-acids. Therefore, proteins consist of only a few types of atoms - hydrogen (H), carbon (C), nitrogen (N), oxygen (O) and sulfur (S).

For our purposes, a protein is just a set of atoms and we interpret each atom as a sphere (see Fig. 9.1). We use Van Der Waals atomic radii shown on Tab. 9.1. The usual unit of length is *one angstrom*, $1\text{Å} = 1 \times 10^{-10}\text{m}$.

Element symbol	H	C	N	O	S
VDW radius [Å]	1.20	1.70	1.55	1.52	1.80

Table 9.1: Van Der Waals atomic radii [19]

Some proteins are simple - they consist of just a small number of atoms. Another proteins can be quite large - about ten thousands of atoms and more. See Fig. 9.1a, 9.1b, and 9.1a for examples.



(a) 1EVD has about 10 atoms

(b) 2RE7 has about 1000 atoms

(c) 2FXJ has about 10 000 atoms

Figure 9.1: Proteins, rendered by *QuteMol*

There are about 50 000 protein models available at RCSB PDB[1]. Fig. 9.2 shows a part of their histogram with respect to the number of atoms.

For our implementation of edge-tracing, it would be impossible to compute Voronoi diagrams of all these proteins in a reasonable time because of the time complexity. Therefore, we have chosen 292 random proteins from some intervals of the histogram as it is shown in Tab. 9.2. The first table column represents the size of a protein as the number of its atoms – it is an interval. The second column represents the rough number of proteins in the interval (after some filtering through the database). The number of samples from these proteins is given by the third column. Furthermore, we have computed diagrams for another 500 proteins 101M–1BVV.

[1]RCSB Protein Data Bank at `http://www.rcsb.org/pdb/home/home.do`

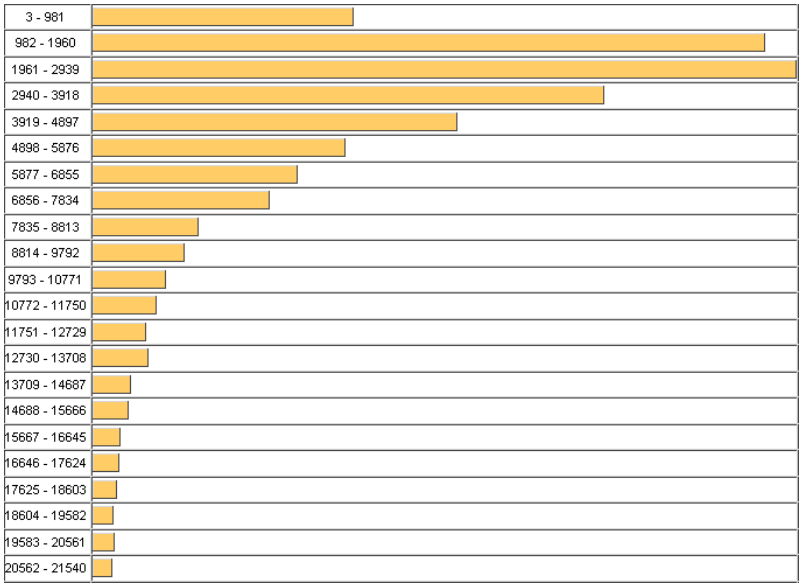


Figure 9.2: Distribution of PDB structures with respect to the number of atoms [2].

Atoms Count	Proteins Count	Selected Proteins Count
3-981	2 000	200
982-1960	8 000	50
1961-2939	8 500	25
2940-3918	6 000	10
3919-4897	4 400	5
12730-13708	600	2

Table 9.2: Random proteins selection.

9.1.2 Random spheres

A set of random spheres should provide diagrams that are more exotic than these obtained from proteins because differences among their radii can be greater and their positions can be somehow "wilder". Note that in proteins the radii are from 1.20Å to 1.80Å and the atoms are in chains because of peptide bonds.

A set of random spheres can not be arbitrary in order to get an interesting diagram, *i.e.* different from *VD(P)*. It needs to fulfill two requirements:

1. Neighboring spheres need to be close enough to each other with respect to their radii.

2. No sphere is fully contained in another sphere.

Spheres need to be close to each other in order to get some influence from the difference in their radii. Consider the case when two spheres have some constant radii r_1 and r_2. The more distant they are, the less impact the difference $r_1 - r_2$ have to the shape of their bisector. The second requirement is the requirement from Chap. 3.

The first requirement is easy to accomplish by generating random coordinates from an interval $I \subset \mathbb{R}^3$ of the specific size, but the second one is a little more difficult.

A naïve approach that generates random spheres in a 3-dimensional box will start having problems when they cover most of the box, because the next sphere will be probably eaten by some existing sphere or eat some existing one. This approach could be improved by adding a logic that does not allow these cases.

We introduce a policy for the generation of a set of random spheres:

"Do not eat your best friend!"

We say that a sphere s has been *eaten* by another sphere q when $s \subset q$. Suppose that a sphere s_i is a point, *i.e.* its radius r_i is zero, and it starts growing in r_i. The first sphere s_j eaten by s_i becomes the *best friend* of s_i.

The policy is formalized by Algo. 4. At first, the algorithm generates random positions for all spheres. Then it randomly generates their radii, carefully avoiding eating best friends. The searching for the best friend can be optimized by creating the Delaunay triangulation for the points in the initialization phase, aggregating friends for each vertex as the neighboring vertices and searching best friends among these candidates. This improves the expected time complexity from $O(n^2)$ to $O(nlog(n))$.

Algorithm 4: Generate Random Spheres

```
1 S ← random set of N spheres with zero radii
2 foreach s ∈ S do
3     s_best ← find the best friend of s
4     r_max ← the minimal radius needed by s to eat s_best
5     r ← random radius, but not greater than r_max
6     set r as the radius of s
7 end
8 return S
```

9.2 Measured characteristics

We have measured these *global characteristics* per each diagram construction:

CPU time as the amount of time spent by a computation process on CPU by diagram construction.

Generators count as the size of the input, *i.e.* the number of generator spheres.

Isolated generators count as the number of Voronoi regions that the edge-tracing algorithm did not found. When this number is zero, the algorithm have found the complete diagram. Otherwise, there were some isolated subgraphs or elliptic edges that the algorithm could not find from the initial starting vertex.

Vertices count as the total number of Voronoi vertices found by the algorithm.

Edges count as the total number of Voronoi edges found by the algorithm.

Collecting these results for some random instances of an input data class, such as proteins, gives us an approximated characterization of the entire class.

9.3 Experiment model

Our experiments look like the schema on Fig. 9.3. There are several input files, each one describes a set of generator spheres. We send these files to VdsCmd. It uses VdsLib to compute the diagram and to get its representation as an XML document. It serializes the diagram and logs possible errors. Furthermore, VdsCmd collects the characteristics and logs them to a statistics log file – one record per diagram. These statistics are then converted to an XML file are further processed.

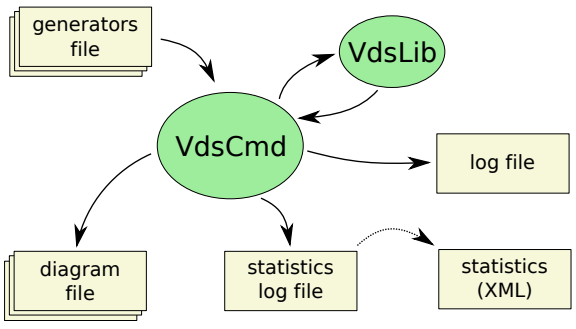


Figure 9.3: Context of an experiment - the data flow.

VdsCmd adds another four generators to each input set. These generators form an outer tetrahedron which stops edges going to the infinity as well as edges going to Voronoi vertices that are too far away. This should connect some parts of the diagram that would be otherwise disconnected, solve some numerical issues and it is convenient for the visualization of Voronoi edges.

For angular distance comparisons in edge-tracing, we use Shewchuk's library for geometric predicates [17]. The library defines several predicates for orientation tests among points in 2- and 3-dimensions. These tests are based on the sign of a determinant. This should solve some issues regarding numerical stability of the angular distance comparison but it does not solve all numerical issues of the algorithm. The library is provided by VdsCmd to VdsLib explicitly. VdsLib does not use it by default and does not depend on it.

9.4 Results

The experiments were performed on an AMD Athlon XP 2100+, 1.73 GHz, 1.50 GB RAM. The results are available in enclosed CD.

9.4.1 Running time

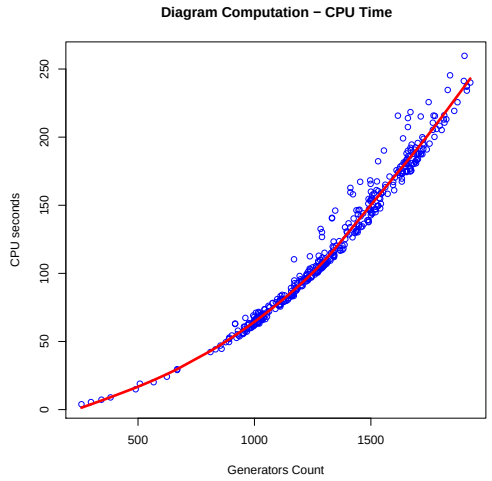
Fig. 9.4 shows the running time of our edge-tracing implementation for different data sets (proteins and random spheres). The horizontal axis represents the size of the input set (*i.e.* the number of generators). The vertical axis shows how much time did a process spent on CPU in the phase of diagram computation. Each mark represents a diagram.

Recall that our implementation has the time complexity $O(mn)$, where n is the number of generators and m is the number of edges in the diagram. The number of edges is expected to be $m \approx O(n)$. Therefore, the expected running time should be $O(n^2)$. Fig. 9.4 confirms that and shows some real examples.

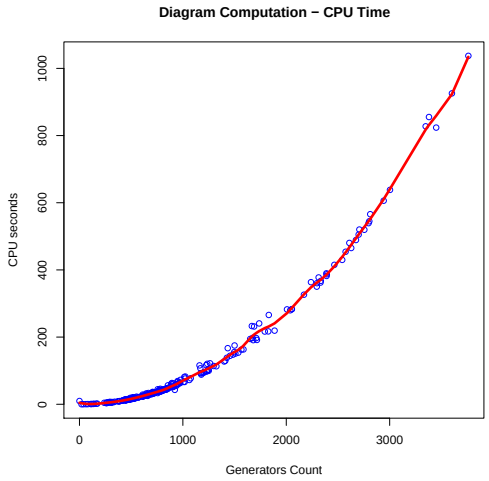
9.4.2 The number of edges and vertices

The number of Voronoi edges and vertices in a diagram is expected to be linear with respect to the size of the input set.

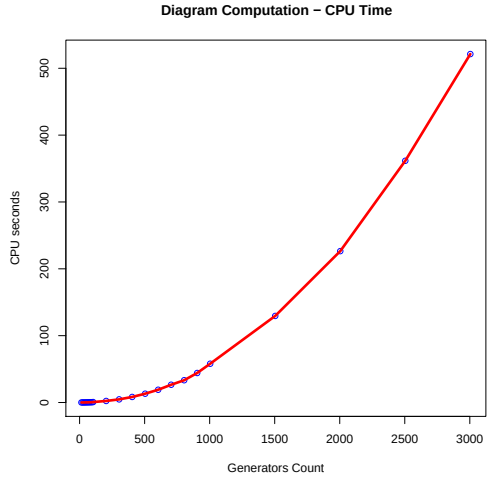
Fig 9.5 shows results from our experiments for both protein data (Fig. 9.5a) and random spheres (Fig. 9.5b) – it is apparent that they both fulfill this expectation. In the case of random spheres, isolated generators were subtracted before doing the



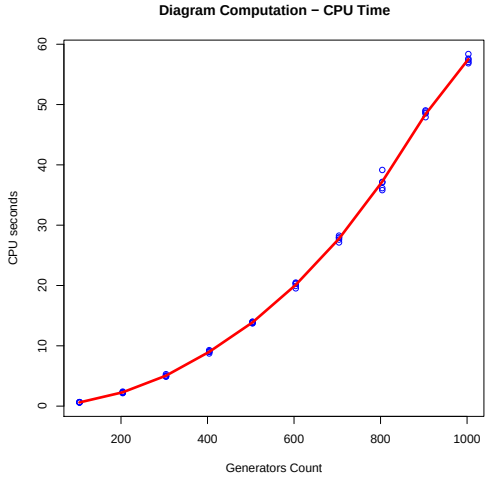
(a) 500 proteins



(b) 292 proteins



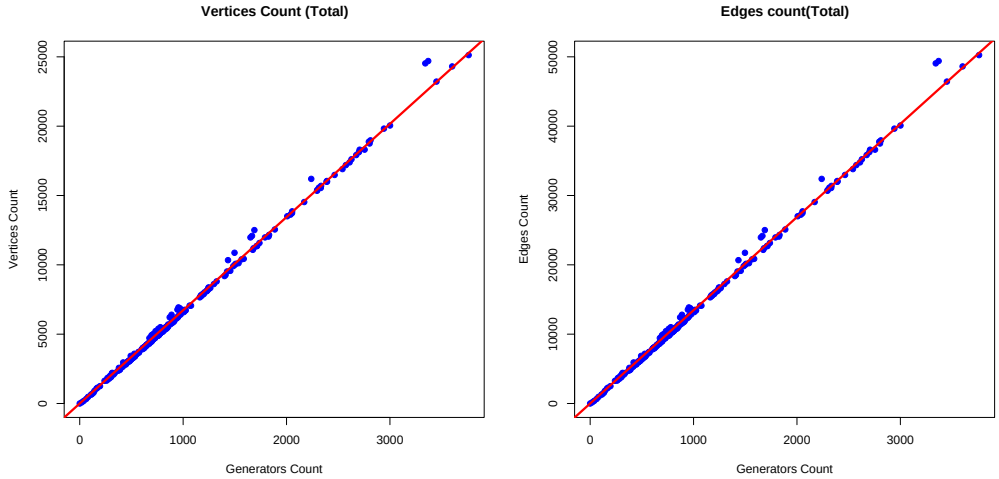
(c) 23 sets of random spheres



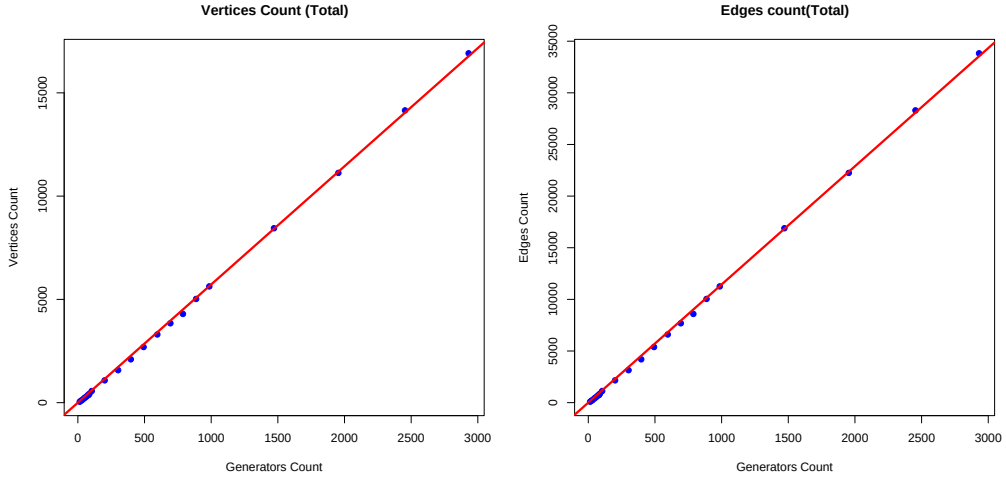
(d) 50 sets of random spheres

Figure 9.4: Running time of our edge-tracing implementation – the reality.

statistics. Note that each figure has a line that approximates the linear trend of the respective quantities. The line represents the linear model of the data. Coefficients of these models are listed in Tab. 9.3. This could be used to predict the total number of vertices and edges in a Voronoi diagram of spheres for random or protein data only from the knowledge of the number of atoms.



(a) 292 proteins; linear model



(b) 23 sets of random spheres; linear model

Figure 9.5: The number of edges and vertices with respect to the input size.

	Vertices	Edges	Input data
Proteins	6.72	13.44	292 proteins
	6.69	13.38	500 proteins
	6.60	13.20	292 proteins, equal radii
Random spheres	6.52	13.04	23 sets, equal radii
	5.72	11.44	23 sets
	5.53	11.01	50 sets

Table 9.3: Coefficients of linear models for edges and vertices

9.4.3 The number of isolated generators

Recall that our implementation of edge-tracing searches only for the maximal connected set of edges from an initial start vertex and it can not handle elliptic edges. Unfortunately, the true diagram can consist of several disconnected subgraphs (connected only by faces). The fact that the algorithm did not find an edge of such a disconnected subgraph is revealed by the presence of isolated generators – these are the generators that do not define any region, because they do not participate to any face, any edge and any vertex. Isolated generators can be identified easily from the set of vertices or edges.

Our experiments show that the number of isolated generators in a diagram[2] is almost always zero for protein data and very small for sets of random spheres (about 2% from the size of the input set).

Fig. 9.6 shows the number of isolated generators for protein data. There are some outliers - for example, 1NIL has 65 and 1COD even 180 isolated generators which is pretty much relative to their size. Rendering their models in JMol has shown obvious z-fighting among spheres representing their atoms. We suspect these models to be wrong.

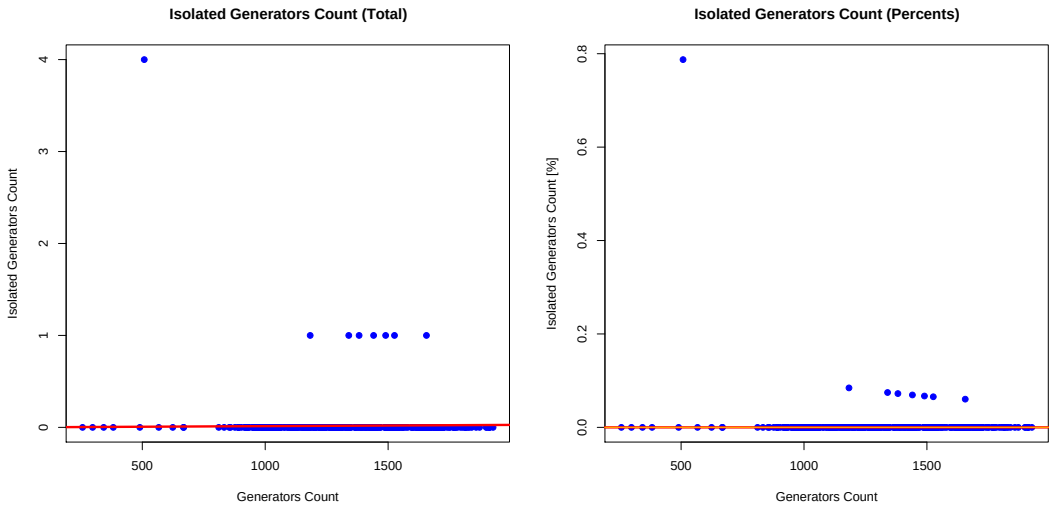
Fig 9.7 shows the number of isolated generators for random spheres (generated by Algo. 4). It seems that their count linearly depends on the size of the input set and hence we express it in percents and use a median to get the expected value. According to our experiments, approximately 2% of an input set are isolated generators.

The values measured from our testing sets are summarized in Tab. 9.4.

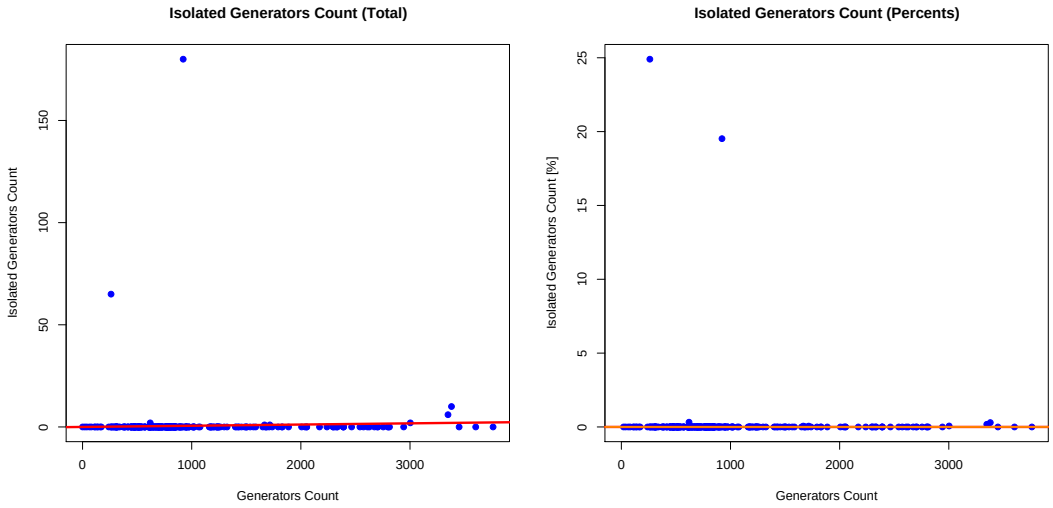
	median (% isolated)	mad (% isolated)	Input data
Proteins	0	0	292 proteins
	0	0	500 proteins
Random spheres	1.881	0.911	23 sets
	1.948	0.806	50 sets

Table 9.4: The percentage of isolated generators in a diagram – characterized by median and its absolute deviation.

[2] we always add four generators that form an outer bounding tetrahedron

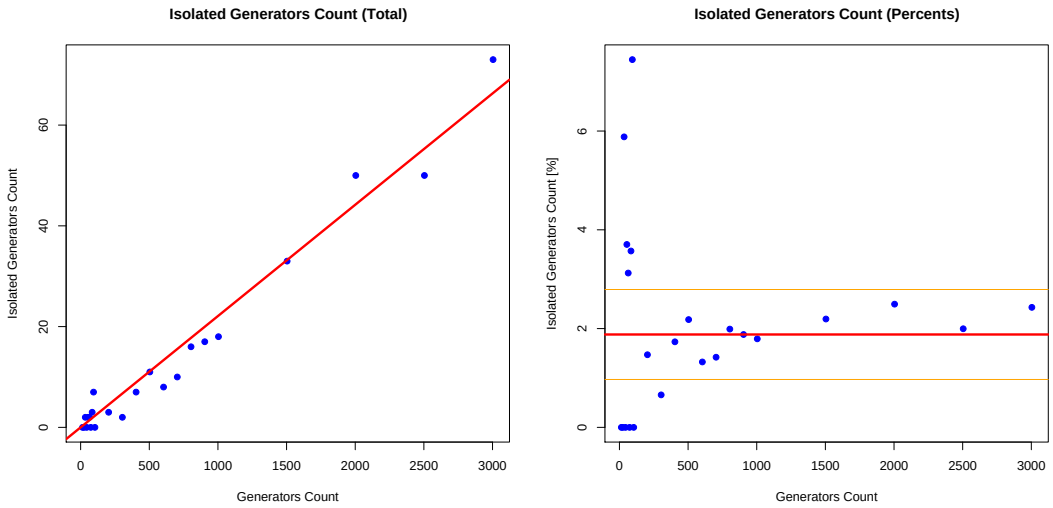


(a) 500 proteins; left – linear model; right – percents and their median

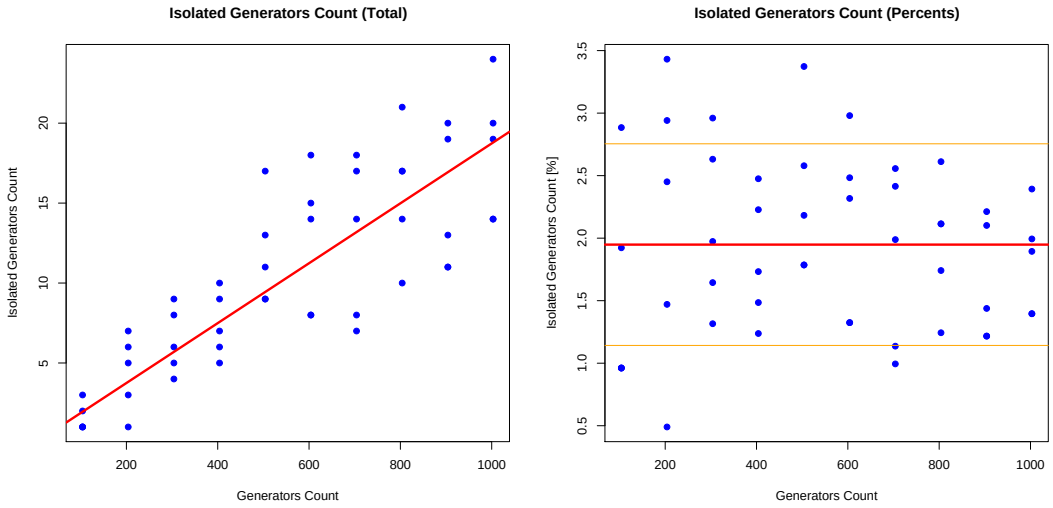


(b) 292 proteins; left – linear model; right – percents and their median

Figure 9.6: The number of isolated generators for protein data.



(a) 23 sets of random spheres; left – linear model; right – percents and their median



(b) 50 sets of random spheres; left – linear model; right – percents and their median

Figure 9.7: The number of isolated generators for random spheres.

9.4.4 Summary

Our experiments showed that the number of isolated generators for protein data is almost zero and about 2% from the input size for pseudo-random data. The experiments also confirmed that the number of vertices and edges is linear with respect to the input size. To be more specific, the total number of vertices seems to be 6.5 times greater and the total number of edges seems to be 13.5 times greater than the total number of atoms in a protein. Diagrams for the input size of about 1000 generators are still achievable for our implementation of edge-tracing, which is not so low when looking at the histogram of protein size available at RCSB PDB.

Chapter 10

Conclusion

We have explored Voronoi diagrams for a set of spheres. From the known algorithms for *VD(S)* construction we implemented *edge-tracing* as a library. In the implementation we successfully use our new algorithm for finding an initial Voronoi vertex (Algo. 2). With the help of this library and related applications, we performed experiments on protein data as well as on random data and provided the results in Chap. 9. We also proposed an algorithm for generating a pseudo-random set of spheres avoiding any full-containment (Algo. 4).

The main purpose of the library is to provide some data for experiments and its practical usability is limited. The library is written in C#, the expected time complexity of the algorithm is $O(n^2)$, it does not handle issues regarding numerical stability and the current implementation ignores elliptic edges and isolated sub-graphs that may occasionally occur in *VD(S)*. On the other hand we have not found any library for computation of *VD(S)* publicly available.

Despite these facts we successfully used the library in our experiments. They showed that elliptic edges and isolated subgraphs are almost absent in protein data and also rare (about 2%) in random data (generated by Algo. 4). The number of vertices and edges is linear in the size of the input set and we experimentally obtained the constants of the respective linear models.

10.1 Future work

The main drawback of our implementation is the expected $O(n^2)$ time complexity. The running time of edge-tracing is dominated by the time spend on searching for end-vertices which is $O(n)$ in the current implementation. This behavior can

be improved by utilizing some kind of geometric filtering. It could be worth to try Delaunay triangulation of sphere centers for prediction of the end-vertices, especially for protein data.

It seems that *VD(S)* for protein data is similar to the respective *VD(P)* for the respective atom positions. This could be further investigated by comparing the topology of these diagrams on the same data. Maybe the differences are big enough to give better results in the tunnel-analysis in proteins.

Bibliography

[1] Voronoi diagram research center, `http://voronoi.hanyang.ac.kr`. [cited at p. 9]

[2] RCSB Protein Data Bank. `http://www.rcsb.org/pdb/home/home.do`. [cited at p. 56]

[3] Youngsong Cho, Donguk Kim, and Deok-Soo Kim. Topology representation for the voronoi diagram of 3d spheres. *International Journal of CAD/CAM*, 5(3), 2005. [cited at p. 9, 40]

[4] Youngsong Cho, Donguk Kim, Hyun-Chan Lee, Joon Young Park, and Deok-Soo Kim. Reduction of the search space in the edge-tracing algorithm for the voronoi diagram of 3d balls. In *ICCSA (1)*, pages 111–120, 2006. [cited at p. 36, 39]

[5] Gerald E. Farin. *Curves and Surfaces for CAGD: A Practical Guide*, volume 5. Morgan Kaufmann, 2001. [cited at p. 29]

[6] M. L. Gavrilova and J. Rokne. Updating the topology of the dynamic voronoi diagram for spheres in euclidean d-dimensional space. *Computer-Aided Geometric Design*, 20:231–242, 2003. [cited at p. 9, 27, 28]

[7] Marina Gavrilova. *Proximity and Applications in General Metrics*. PhD thesis, The University of Calgary, Dept. of Computer Science, Calgary, AB, Canada, MAY 1998. [cited at p. 7, 12, 36]

[8] Deok-Soo Kim, Youngsong Cho, and Donguk Kim. Edge-tracing algorithm for euclidean voronoi diagram of 3d spheres. In *Proceedings of the 16th Canadian Conference on Computational Geometry (CCCG'04)*, pages 176–179, 2004. [cited at p. 9]

[9] Deok-Soo Kim, Youngsong Cho, and Donguk Kim. Euclidean voronoi diagram of 3d balls and its computation via tracing edges. *Computer-Aided Design*, 37:1412–1424, 2005. [cited at p. 9, 27, 32, 34, 36, 38, 39, 50]

[10] Deok-Soo Kim, Donguk Kim, and Youngsong Cho. Euclidean voronoi diagrams of 3d spheres: Their construction and related problems from biochemistry. In *IMA Conference on the Mathematics of Surfaces*, pages 255–271, 2005. [cited at p. 8]

[11] Deok-Soo Kim, Donguk Kim, Youngsong Cho, and Kokichi Sugihara. Quasi-triangulation and interworld data structure in three dimensions. *Computer-Aided Design*, 38(7):808–819, 2006. [cited at p. 9, 18, 21, 24, 40, 43, 44]

[12] Donguk Kim, Cho, and Deok-Soo Kim. Region expansion by flipping edges for euclidean voronoi diagrams of 3d spheres based on a radial data structure. In *ICCSA (1)*, pages 716–725, 2005. [cited at p. 9, 25, 40]

[13] Donguk Kim and Deok-Soo Kim. Region-expansion for the voronoi diagram of 3d spheres. *Computer-Aided Design*, 38:417–430, 2006. [cited at p. 9, 25]

[14] Sang Hun Lee and Kunwoo Lee. Partial entity structure: a compact non-manifold boundary representation based on partial topological entities. In *SMA '01: Proceedings of the sixth ACM symposium on Solid modeling and applications*, pages 159–170, New York, NY, USA, 2001. ACM. [cited at p. 40]

[15] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial tessellations: concepts and applications of Voronoi diagrams*. John Wiley & Sons, Inc., 1992. [cited at p. 7]

[16] Jeongyeon Seo, Donguk Kim, Youngsong Cho, Joonghyun Ryu, and Deok-Soo Kim. Beta-shape and beta-complex for the structure analysis of molecules. *International Journal of CAD/CAM*, 7(1), 2007. [cited at p. 9, 24]

[17] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust predicates for computational geometry, `http://www.cs.cmu.edu/~quake/robust.html`. [cited at p. 59]

[18] G. Voronoi. Nouvelles applications des paramétres continus á la théorie des formes quadratiques. *Journal fur die Reine and Angewandte Mathematic*, 133:97–178, 1907. [cited at p. 7]

[19] Wikipedia. Atomic radii of the elements, `http://en.wikipedia.org/wiki/Atomic_radii_of_the_elements_(data_page)`. [cited at p. 55]

[20] Michal Zemek. Dělení prostoru pro rozsáhlá a měnící se data. Master's thesis, FAV, University of West Bohemia, Pilsen, 2007. [cited at p. 8]

Appendix A

Screenshots

We rendered some diagrams obtained by the experiments discussed in Chap. 9. Our tool for *VD(S)* visualization is shown on Fig. A.1. Following pictures are real examples of diagrams computed by our library and rendered by our Viewer.

Fig. A.2 shows two diagrams. The first one is a complete *VD(S)* for a protein consisting of about 800 atoms. As it was mentioned before, the number of isolated generators for protein data is almost always zero hence it is expectable that the diagram computed for a protein will be complete. The second diagram is a diagram computed for a set of 2500 random spheres generated by Algo. 4. Note that it is not a complete *VD(S)* because it has 50 isolated generators (2% of the size of the input set) Fig. A.3 shows a dense set of 70 random spheres. The diagram is complete.

Fig. A.4 shows an importance of adding an outer tetrahedron to the original set of generators. The role of the four spheres of the tetrahedron is to stop edges going to the infinity or creating distant vertices that are too far away to be important but too far away to cause numerical stability issues. When this outer tetrahedron is absent, regions at the convex hull boundary of the original input set can be incomplete.

In Fig. A.5, there are some Voronoi regions for protein data. At the first look, they seem to be quite linear, such as regions in *VD(P)*. This is not any surprising discovery because atoms in proteins have similar radii and they are not so close to each other to have very significant impact on the shape of bisectors. Interesting non-linear edges can be found in regions near the boundary of the protein. In other words, an edge needs to be long enough to be bent enough and for this it needs some space (on the convex hull boundary, for instance).

Next Fig. A.6 shows some regions for a random set of spheres. There things get more interesting, because generator spheres can have greater differences in their radii. The first two figures show regions with topology impossible for *VD(P)* and hard-to-find

in proteins. The next figure shows a small sphere between two great spheres ("big brothers"). This configuration would result in an elliptic edge, but there is another sphere that breaks the ellipse and creates two vertices on the same four generators. Next figure is similar – there is also an elliptic edge segment. But we want to show a different point here and it is that the "Do-no-eat-your-best-friend-policy" followed by Algo. 4 really works...

Circular configurations of atoms as they are shown on Fig. A.7 are quite frequent in proteins. These configurations produce almost identical edges. This could be problematic for edge-tracing – it fails on an attempt to define a wrong vertex.

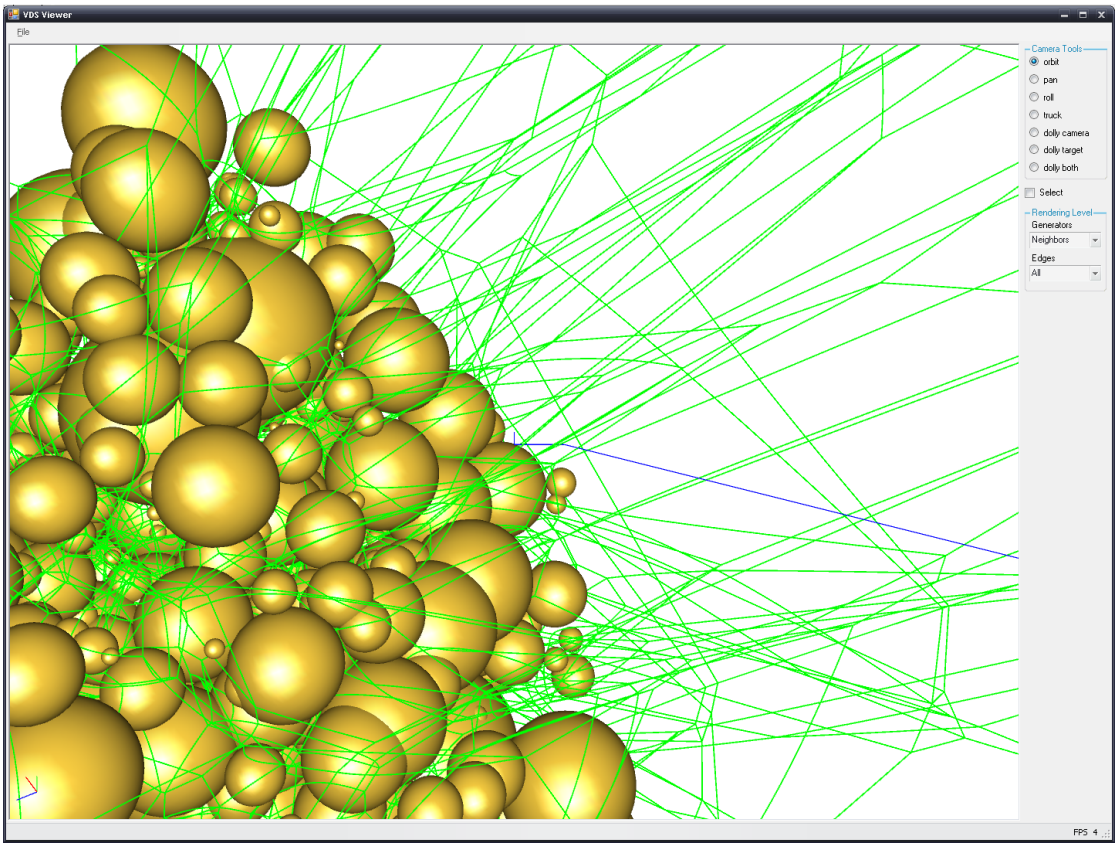
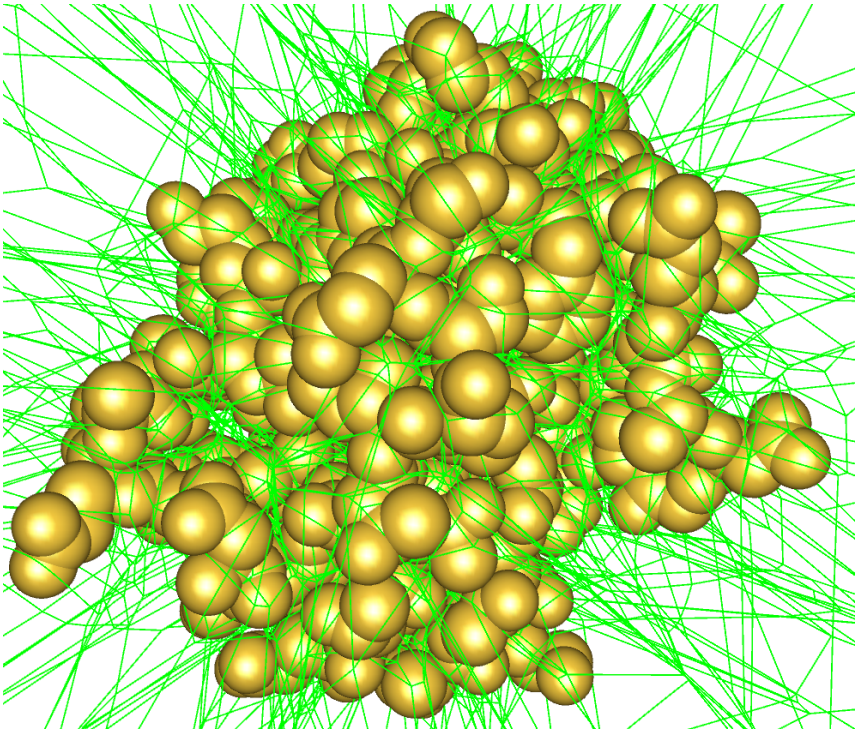
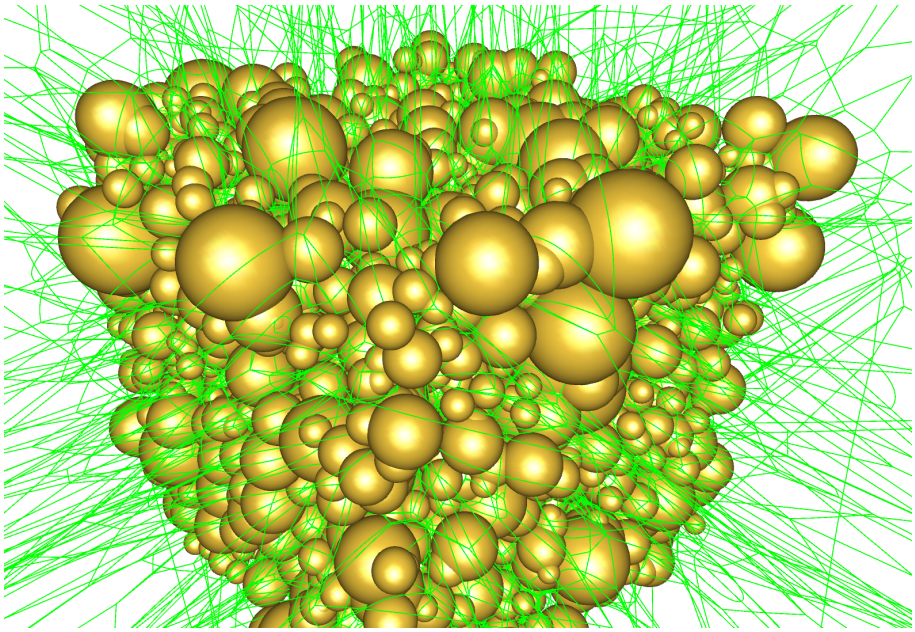


Figure A.1: Viewer and the diagram of a set of random spheres



(a) Full *VD(S)* for a protein with PDB id 1AAJ (about 800 atoms)



(b) *VD(S)* for a random set (about 2500 spheres, 50 isolated generators)

Figure A.2: Voronoi diagrams for different input sets

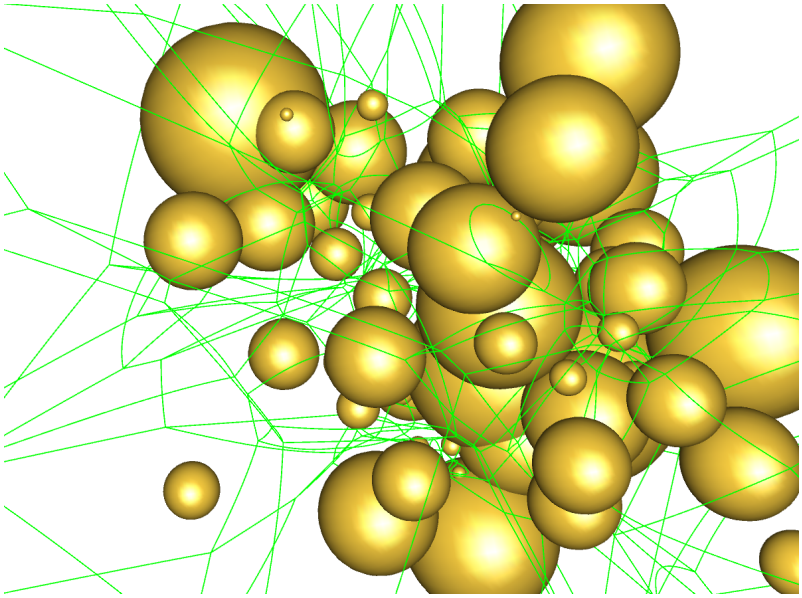
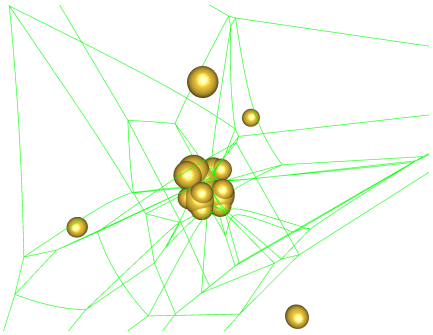
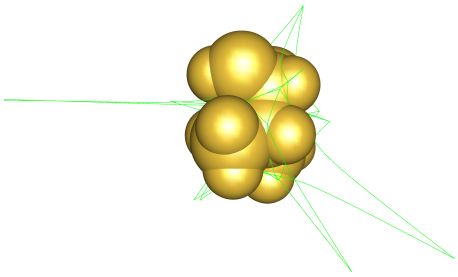


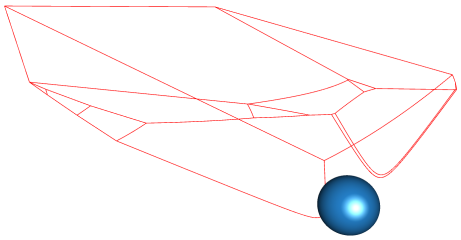
Figure A.3: Full *VD(S)* for a dense set of random spheres (70 generators)



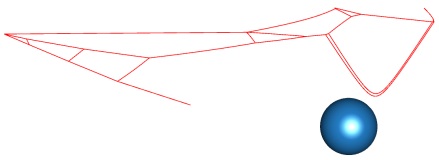
(a) Small protein 1EVD with an outer tetrahedron added to the set of atoms



(b) Small protein 1EVD without any outer tetrahedron

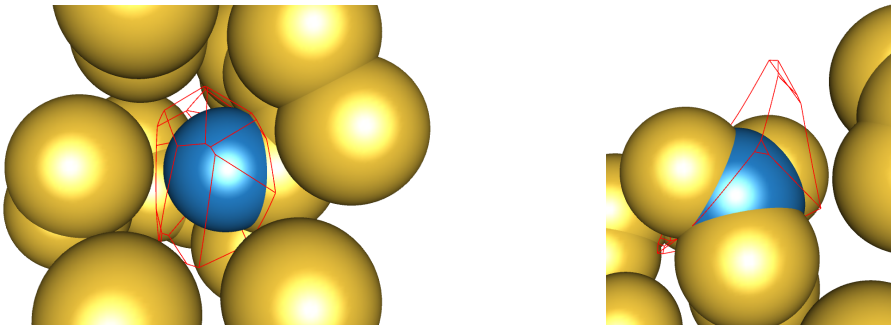


(c) Valid Voronoi region for an atom on the convex hull boundary (protein 1TKQ)

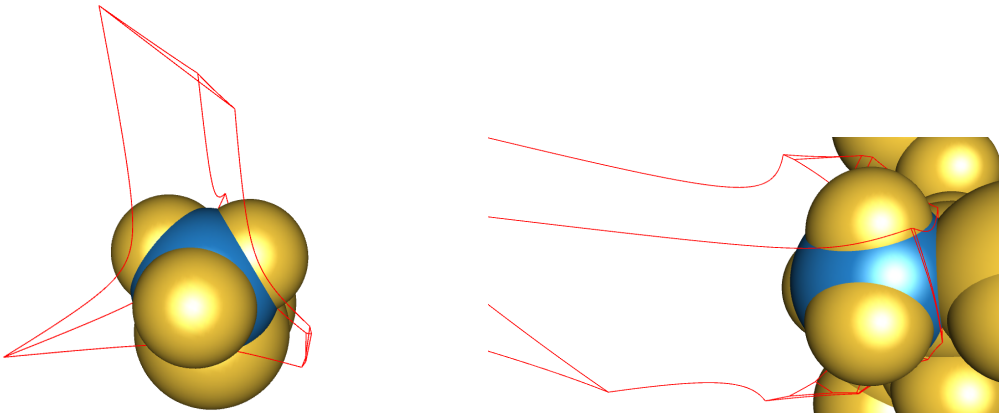


(d) Incomplete Voronoi region (caused by the absence of the outer tetrahedron)

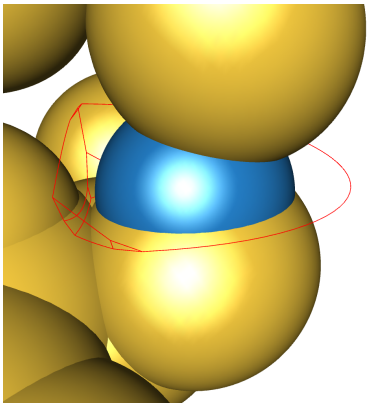
Figure A.4: It is important to add an outer tetrahedron to the input data set.



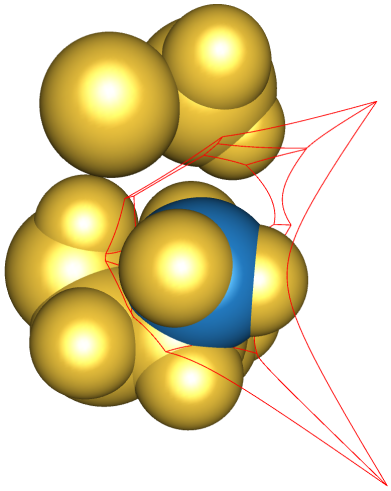
(a) Some ordinary Voronoi regions (protein 1AAJ)



(b) Voronoi edges are less linear for regions close to the convex hull boundary (proteins 1AUM and 1TKQ)

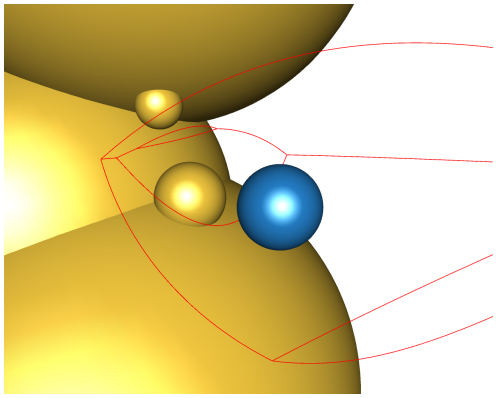


(c) A region containing an elliptic edge segment (protein 1TKQ)

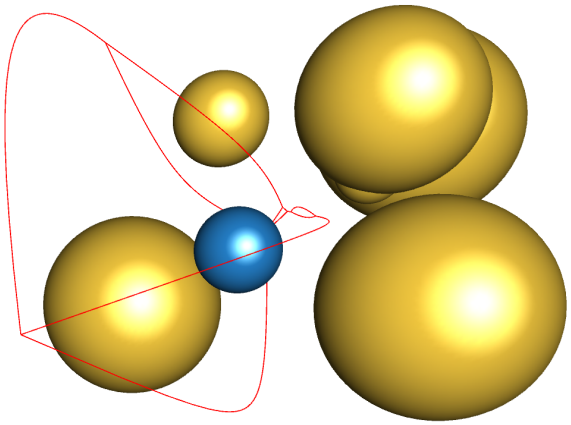


(d) A region in the protein 1LVQ

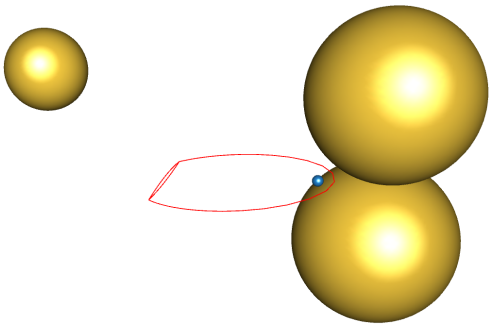
Figure A.5: Voronoi regions for protein data



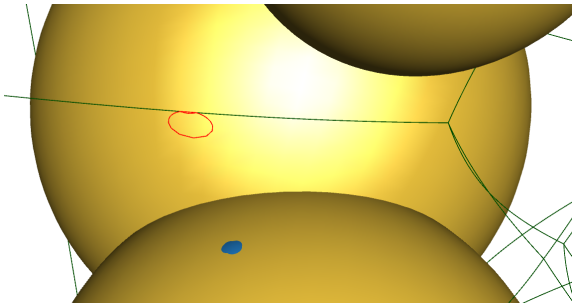
(a) Two edges are defined by the same three generators



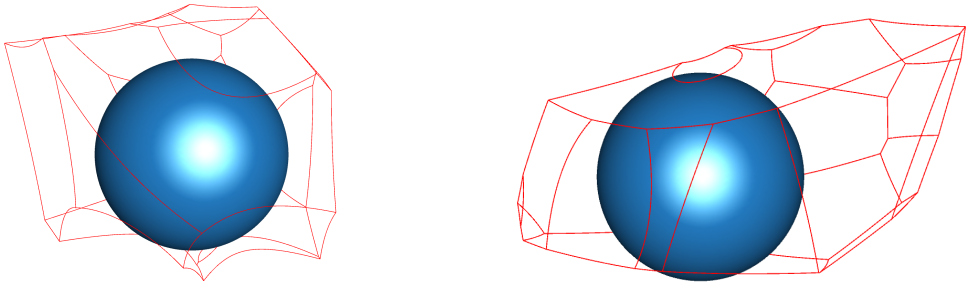
(b) A region in the random set of spheres



(c) A small generator between two "big brothers"

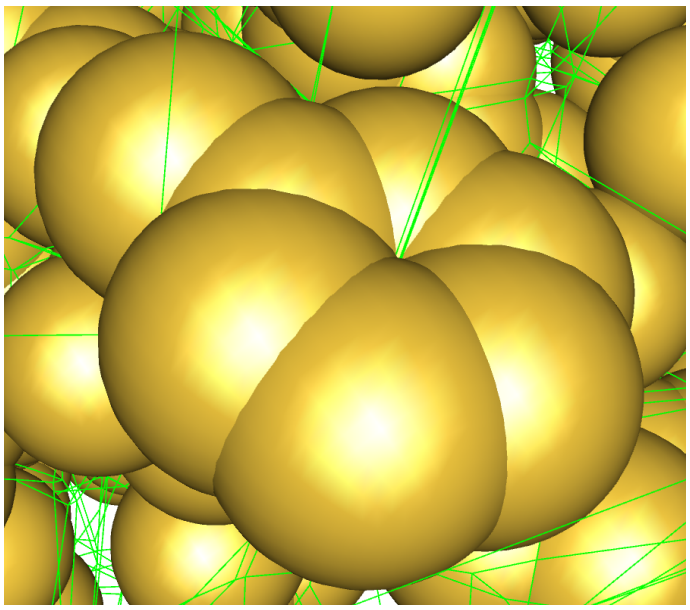


(d) A sphere has been almost "eaten" but it still defines part of a small elliptic edge segment

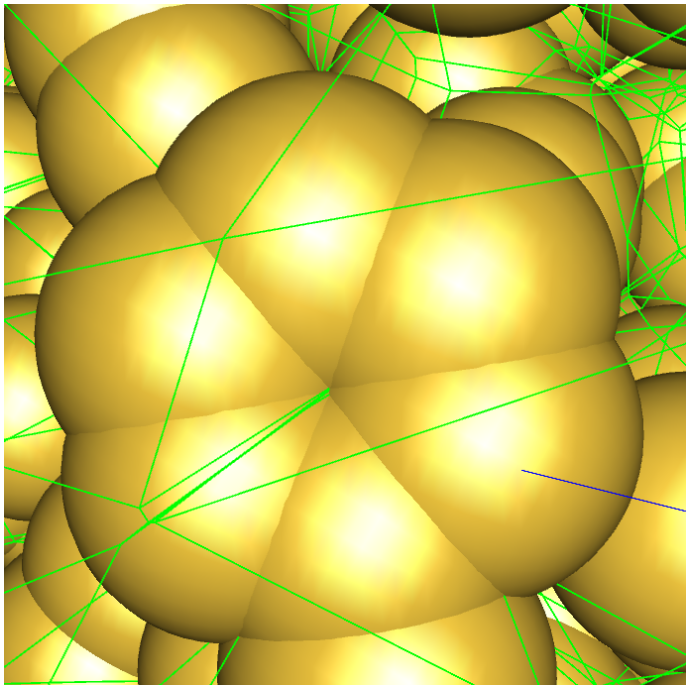


(e) Two regions shown without their neighbors

Figure A.6: Voronoi regions from random spheres



(a) Protein 1C2A



(b) Protein 1AUM

Figure A.7: Problematic configuration of atoms in proteins – almost identical edges are a possible source of numeric stability issues