Voronoi diagrams in CGAL

Menelaos I. Karavelas*

Abstract

In this paper we describe a generic `C++` adaptor[1], that adapts a 2-dimensional triangulated Delaunay graph and to the corresponding a Voronoi diagram, represented as a doubly connected edge list (DCEL) data structure. Our adaptor has the ability to automatically eliminate, in a consistent manner, degenerate features of the Voronoi diagram, that are artifacts of the requirement that Delaunay graphs should be triangulated even in degenerate configurations. Depending on the type of operations that the underlying Delaunay graph supports, our adaptor allows for the incremental or dynamic construction of Voronoi diagrams and can support point location queries. Our code will appear in the next public release of CGAL.

1 Introduction

A Voronoi diagram on the plane is typically defined for a set of planar objects, also called sites in the sequel, and a distance function that measures the distance of a point x in $\mathbb{R}^2$ from an object in the object set. Let $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ be our set of sites and let $\delta(x, S_i)$ denote the distance of a point $x \in \mathbb{R}^2$ from the site S_i. Given two sites S_i and S_j, the set V_{ij} of points that are closer to S_i than to S_j with respect to the distance function $\delta(x, \cdot)$ is simply the set: $V_{ij} = \{x \in \mathbb{R}^2 : \delta(x, S_i) < \delta(x, S_j)\}$. We can then define the set V_i of points on the plane that are closer to S_i than to any other object in $\mathcal{S}$ as $V_i = \bigcap_{i \neq j} V_{ij}$. The set V_i is said to be the *Voronoi cell* or *Voronoi face* of the site S_i. The locus of points on the plane that are equidistant from exactly two sites S_i and S_j is called a *Voronoi bisector*. A point that is equidistant to three or more objects in $\mathcal{S}$ is called a *Voronoi vertex*. A simply connected subset of a Voronoi bisector is called a *Voronoi edge*. The collection of Voronoi faces, edges and vertices is called the *Voronoi diagram* of the set $\mathcal{S}$ with respect to the distance function $\delta(x, \cdot)$, and it is a subdivision of the plane.

We typically think of faces as 2-dimensional objects, edges as 1-dimensional objects and vertices as 0-dimensional objects. However, this may not be the case for several combinations of sites and distance functions (for example points in $\mathbb{R}^2$ under the L_1 or the L_∞ distance can produce 2-dimensional Voronoi edges). Moreover, the cell of a site can in general consist of several disconnected components (e.g., in the multiplicatively weighted Euclidean Voronoi diagram). In this paper we are going to restrict ourselves to Voronoi diagrams that have the property that the Voronoi cell of each site is a simply connected region of the plane. We are going to call such Voronoi diagrams *simple Voronoi diagrams*. Examples of simple Voronoi diagrams include the usual Euclidean Voronoi diagram of points, the Euclidean Voronoi diagram of a set of disks on the plane, the Euclidean Voronoi diagram of a set of disjoint convex objects on the plane, or the power (Laguerre) diagram for a set of circles on the plane. In fact every instance of an *abstract Voronoi diagram* in the sense of Klein [2] is a simple Voronoi diagram in our setting. In the sequel when we refer to Voronoi diagrams we refer to simple Voronoi diagrams.

2 Adapting triangulated Delaunay graphs

In many applications we are not really interested in computing the Voronoi diagram itself, but rather its dual graph, called the *Delaunay graph*. In general the Delaunay graph is a planar graph, each face of which consists of at least three edges. Under the non-degeneracy assumption that no point in the plane is equidistant to more than three sites, the Delaunay graph is a planar graph with triangular faces. In certain cases this graph can actually be embedded with straight line segments in which case we talk about a triangulation (e.g., the Euclidean Voronoi diagram/Delaunay triangulation of points, or the power diagram/regular triangulation of a set of circles). Graphs of non-constant non-uniform face complexity can be undesirable in many applications, so we typically end up triangulating the non-triangular faces of the Delaunay graph.

Choosing between computing the Voronoi diagram or the (triangulated) Delaunay graph is a major decision while implementing an algorithm. It heavily affects the design and choice of the different data structures involved. Although in theory the two approaches are entirely equivalent, it is not so straightforward to go from one representation to the other. The objective for our adaptor is to provide a generic

*Applied Mathematics Department, University of Crete; `mkaravel@tem.uoc.gr` and Institute of Applied and Computational Mathematics, Foundation for Research and Technology - Hellas.

[1]An adaptor is a class or a function that transforms one interface into a different one.

way of going from triangulated Delaunay graphs to planar subdivisions represented through a DCEL data structure. Although the look and feel is that of a DCEL data structure, internally we keep the graph data structure representing triangular graphs.

The adaptation might seem straightforward at a first glance, and this is true if our data do not contain degenerate configurations. The situation becomes complicated whenever we want to treat the artifacts introduced in our representation due to these degenerate configurations. Suppose for example that we have a set of sites that contains subsets of sites in degenerate positions. The dual of the computed triangulated Delaunay graph is a Voronoi diagram that has all its vertices of degree 3, and for that purpose we are going to call it a *degree-3 Voronoi diagram* in order to distinguish it from the true Voronoi diagram of the input sites. A degree-3 Voronoi diagram can have degenerate features, namely Voronoi edges of zero length, and/or Voronoi faces of zero area, and do not correspond to the true geometry of the Voronoi diagram.

The manner that we treat such issues is by defining an *adaptation policy*. The adaptation policy is responsible for determining which features in the degree-3 Voronoi diagram are to be rejected and which not. The policy to be used can vary depending on the application or the intended usage of the resulting Voronoi diagram. What we care about is that firstly the policy itself is consistent and, secondly, that the adaptation is also done in a consistent manner. The latter is the responsibility of the adaptor we provide, whereas the former is the responsibility of the implementor of a policy. We currently provide two types of adaptation policies, which are discussed in Section 5.

Delaunay graphs can be mutable or non-mutable. By mutable we mean that sites can be inserted or removed at any time, in an entirely on-line fashion. By non-mutable we mean that once the Delaunay graph has been created, no changes, with respect to the set of sites defining it, are allowed. If the Delaunay graph is a non-mutable one, then the Voronoi diagram adaptor is a non-mutable adaptor as well. If the Delaunay graph is mutable then the question of whether the Voronoi diagram adaptor is also mutable is slightly more complex to answer. In Section 6 we discuss the issue in detail.

3 Software design

The class `Voronoi_diagram_2<DG,AT,AP>` implements our generic adaptor. It is parametrized by three template parameters which are required to be models of corresponding concepts (see Fig. 1). The first template parameter must be a model of the `DelaunayGraph_2` concept, which corresponds to the interface required from a class representing a Delaunay graph. Currently, all classes of CGAL that represent Delaunay graphs are models of this concept, namely, Delaunay triangulations, regular triangulations, Apollonius graphs and segment Delaunay graphs [1]. The second template parameter must be a model of the `AdaptationTraits_2` concept, which is responsible for accessing the geometric information needed from the specific Delaunay graph in order to perform the adaptation. We discuss this concept in detail in Section 4. Finally, the third template parameter must be model of the `AdaptationPolicy_2` concept, which refers to the policy used to perform the adaptation. This concept is discussed in detail in Section 5.

The `Voronoi_diagram_2<DG,AT,AP>` class has been intentionally designed to provide an interface similar to CGAL's arrangements: Voronoi diagrams are special cases of arrangements after all. The interfaces of the two classes, however, could not be identical. The reason is that arrangements in CGAL do not yet support more than one unbounded faces, or equivalently, cannot handle unbounded curves. On the contrary, a Voronoi diagram defined over at least two generating sites, has at least two unbounded faces.

On a more technical level, the `Voronoi_diagram_2<DG,AT,AP>` class imitates the representation of the Voronoi diagram (seen as a planar subdivision) by a DCEL (Doubly Connected Edge List) data structure. We have vertices (the Voronoi vertices), halfedges (oriented versions of the Voronoi edges) and faces (the Voronoi cells). We can perform all standard operations of the DCEL data structure: go from a halfedge to its next and previous in the face; go from one face to an adjacent one through a halfedge and its twin (opposite) halfedge; walk around the boundary of a face; enumerate/traverse the halfedges incident to a vertex from a halfedge, access the adjacent face; from a face, access an adjacent halfedge; from a halfedge, access its source and target vertices; from a vertex, access an incident halfedge.

In addition to the above possibilities for traversal, we can also traverse the following features through iterators: the vertices of the Voronoi diagram; the edges or halfedges of the Voronoi diagram; the faces of the Voronoi diagram; the bounded/unbounded faces of the Voronoi diagram; the bounded/unbounded halfedges of the Voronoi diagram; the sites defining the Voronoi diagram.

Finally, depending on the adaptation traits passed to the Voronoi diagram adaptor, we can perform point location queries, namely given a point p we can determine the feature of the Voronoi diagram (vertex, edge, face) on which p lies.

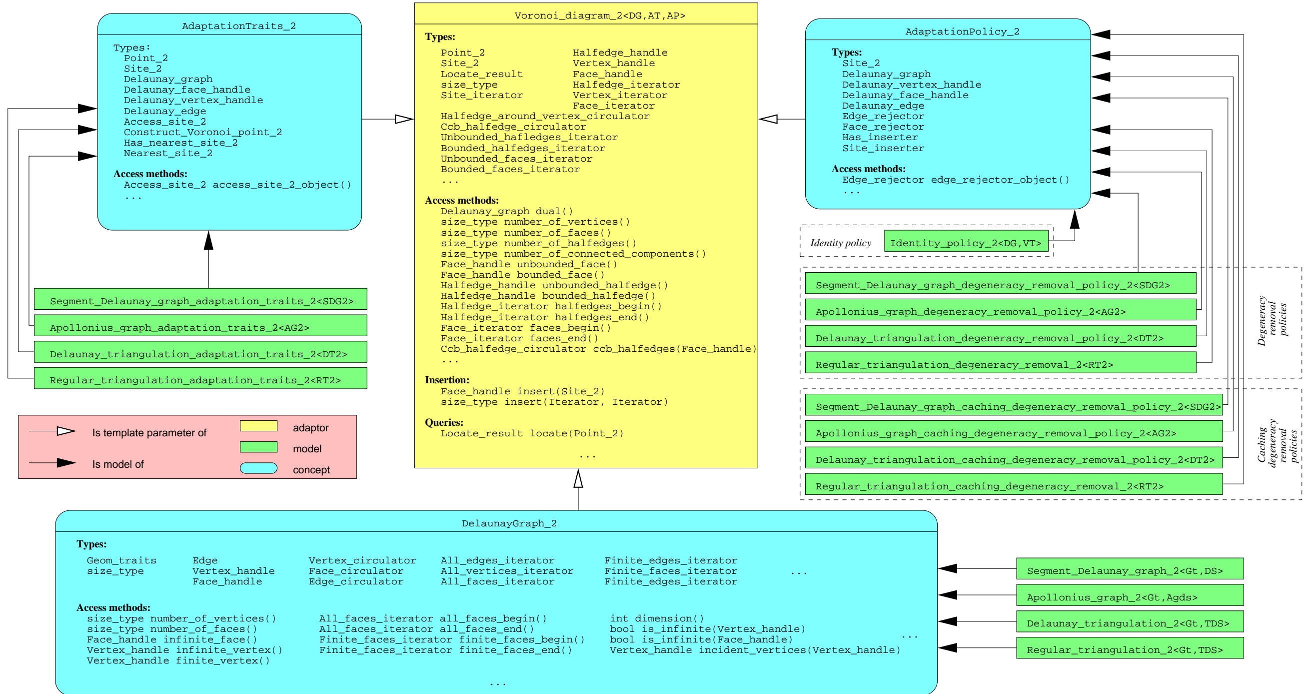

Figure 1: The design of the Voronoi diagram adaptor, and the relations between the various concepts, their models and the adaptor.

4 The adaptation traits

The `AdaptationTraits_2` concept defines the types and functors required by our adaptor in order to access geometric information from the Delaunay graph. In particular, it defines the type of the generating sites, and provides functors for accessing these sites in the Delaunay graph as well as constructing Voronoi vertices given their dual triangular faces in the Delaunay graph.

Finally, it defines a tag that indicates whether nearest site queries are to be supported by the Voronoi diagram adaptor. If such queries are to be supported, a corresponding functor is also required. Given a query point, the nearest site functor should return information related to how many and which sites of the Voronoi diagram are at equal and minimal distance from the query point. This way of abstracting the point location mechanism allows for multiple different point location strategies, which are passed to the Voronoi diagram adaptor through different models of the `AdaptationTraits_2` concept. The point location queries of the `Voronoi_diagram_2<DG,AT,AP>` class uses internally this nearest site query functor.

Along with our adaptor we provide four adaptation traits classes, all of which support nearest site queries. These four classes serve as adaptation traits for CGAL's Apollonius graphs, Delaunay and regular triangulations and segment Delaunay graphs, respectively.

5 The adaptation policy

When we perform the adaptation of a triangulated Delaunay graph to a Voronoi diagram, a question that arises is whether we want to eliminate certain features of the Delaunay graph when we construct its Voronoi diagram representation. We resolve such issues, in a generic way, via the introduction of an adaptation policy. The adaptation policy is responsible for determining which features in the degree-3 Voronoi diagram are to be rejected and which not. The policy to be used can vary depending on the application or the intended usage of the resulting Voronoi diagram.

The concept `AdaptationPolicy_2` defines the requirements on the predicate functors that determine whether a feature of the triangulated Delaunay graph should be rejected or not. More specifically it defines an `Edge_rejector` and a `Face_rejector` functor that answer the question: "should this edge (face) of the Voronoi diagram be rejected?".

We have implemented two types of policies that provide two different ways for answering the question of which features of the Voronoi diagram to keep and which to discard. The first one is called the *identity policy* and corresponds to the `Identity_policy_2<DG,VT>` class. This policy is in some sense the simplest possible one, since it does not reject any feature of the Delaunay graph. The Voronoi diagram provided by the adaptor is the true dual (from the graph-theoretical point of view) of the triangulated Delaunay graph adapted.

The second type of policy we provide is called *degeneracy removal policy*. If the set of sites defining the triangulated Delaunay graph contains subsets of sites in degenerate configurations, the graph-theoretical dual of the triangulated Delaunay graph has edges and potentially faces that are geometrically degenerate. By that we mean that the dual of the triangulated Delaunay graph can have Voronoi edges of zero length or Voronoi faces/cells of zero area. Such features may not be desirable, in which case we would like to eliminate them. The degeneracy removal policies eliminate exactly these features. Along with our Voronoi diagram adaptor we provide four degeneracy removal policies, namely for Apollonius graphs, Delaunay triangulations, regular triangulations and segment Delaunay graphs.

A variation of the degeneracy removal policies are the *caching degeneracy removal policies*. In these policies we cache the results of the edge and face rejectors. Such policies really pay off when we have a lot of degenerate data in our input set of sites. This is due to the fact that detecting whether a Voronoi edge or a Voronoi face is degenerate implies computing the outcome of a predicate in a possibly degenerate or near degenerate configuration, which is typically very costly (compared to computing the same predicate in a generic configuration). We provide four caching degeneracy removal policies, one per degeneracy removal policy mentioned above.

6 Mutable vs. non-mutable policies

In addition to the edge and face rejectors the adaptation policy defines a boolean tag, the `Has_inserter` tag. Semantically, this tag determines if the adaptor is allowed to insert sites in an on-line fashion (on-line removals are not yet supported). In the former case, i.e., when on-line site insertions are allowed, an additional functor is required, the `Site_inserter` functor. This functor takes as arguments a reference to a Delaunay graph and a site, and inserts the site in the Delaunay graph. Upon successful insertion, a handle to the vertex representing the site in the Delaunay graph is returned. In our discussion of the adaptation policies, we did not indicate the value of the `Has_inserter` tag for the degeneracy removal and caching degeneracy removal policies. The issue is discussed in detail in the sequel.

In Section 2 we raised the question whether the adaptor is a mutable or non-mutable one, in the sense of whether we can add/remove sites in an on-line fashion. The answer to this question depends on: (1) whether the Delaunay graph adapted allows for on-line insertions/removals and (2) whether the associated adaptation policy maintains a state and whether this state is easily maintainable when we want to allow for on-line modifications.

As we mentioned above, the way we indicate if we allow on-line insertions of sites is via the `Has_inserter` tag. A *true* value indicates that our adaptation policy allows for on-line insertions, whereas a *false* value indicates the opposite. Note that these values *do not* indicate if the Delaunay graph supports on-line insertions, but rather whether the Voronoi diagram adaptor should be able to perform on-line insertions or not. This delicate point will be become clearer below.

If the Delaunay graph is non-mutable, the Voronoi diagram adaptor cannot perform on-line insertions of sites anyway. In this case not only degeneracy removal policies, but rather every single adaptation policy for adapting the Delaunay graph in question should have the `Has_inserter` tag set to *true*.

If the Delaunay graph is mutable we can choose between two types of adaptation policies, those that allow these on-line insertions and those that do not. At a first glance it may seem excessive to restrict existing functionality. There are situations, however, where such a choice is necessary. Consider a caching degeneracy removal policy. If we do not allow for on-line insertions then the cached quantities are always valid since the Voronoi diagram never changes. If we allow for on-line insertions the Voronoi diagram can change, which implies that the results of the edge and face degeneracy testers that we have been cached are no longer valid or relevant. In these cases, we need to update the cached values, and ideally we would like to do this in an efficient manner.

For our caching degeneracy removal policies, our choice was made on the grounds of whether we can update the cached results efficiently when insertions are performed. For CGAL's Apollonius graphs, Delaunay triangulation and regular triangulations it is possible to ask what are the edges and faces of the Delaunay graph that are to be destroyed when a query site is inserted. This is done via the `get_conflicts` method provided by these classes. Using the outcome of the `get_conflicts` method the site inserter can first update the cached results (i.e., indicate which are invalidated) and then perform the actual insertion. Such a method does not yet exist for segment Delaunay graphs. We have thus chosen to support on-line insertions for all non-caching degeneracy removal policies, i.e., the caching degeneracy removal policy for segment Delaunay graphs does not support on-line insertions, whereas the remaining three caching degeneracy removal policies support on-line insertions.

References

[1] The CGAL homepage. `http://www.cgal.org/`.

[2] R. Klein. *Concrete and Abstract Voronoi Diagrams*, volume 400 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1989.