Additively Weighted Voronoi Diagram on the Oriented Projective Plane*

Guilherme A. Pinto† and Pedro J. de Rezende‡

Institute of Computing, University of Campinas
C.P. 6176, 13083-970, Campinas, SP, Brazil
{guialbu,rezende}@ic.unicamp.br

Abstract

We present a new simple on-line incremental algorithm for the construction of the additively weighted Voronoi diagram, whose primitive operations are completely based on orientation tests. By working on the oriented projective plane $\mathbb{T}^2$, which embeds the Euclidean plane, we are able to avoid idiosyncrasies of the latter, such as the non connectivity of the closest site diagram and the difficulty of dealing with disconnected faces on the furthest site diagram. On $\mathbb{T}^2$, the closest and furthest site diagrams are antipodal, that is, they are, in fact, one and the same diagram, which turns out to always be connected, to have every face simply connected and, as a consequence, to have exactly $3n-6$ edges and $2n-4$ vertices, where n is the number of sites.

1 Introduction

We consider the additively weighted Voronoi diagram constructed on the oriented projective plane $\mathbb{T}^2$ [15], which is comprised of two copies of the Euclidean plane $\mathbb{E}^2$ plus one line at infinity. With the definition of Euclidean concepts such as distance and perpendicularity, $\mathbb{T}^2$ becomes the two-sided Euclidean plane. In this geometry, the closest and furthest site diagrams are antipodal, that is, when the sites are contained in one side of the plane, the part of the closest site diagram contained in the other side of the plane happens to be the furthest site diagram (as we know it in $\mathbb{E}^2$) and vice-versa. Also, the diagram is always connected and every face is simply connected, whereas in $\mathbb{E}^2$ the closest site diagram may have $O(n)$ connected components [14, 3] and the furthest site diagram may have faces with $O(n)$ connected components [11]. As a consequence, the diagram has exactly $3n-6$ edges and $2n-4$ vertices, where n is the number of sites, regardless of the number of sites in the convex hull. In Section 2 we introduce the basic concepts and show how the diagrams appear on $\mathbb{T}^2$.

In Section 3 we present a simple on-line incremental algorithm, whose primitive operations are completely based on orientation tests. Previous algorithms for the closest site diagram can be found, for instance, in [7, 14, 3, 16, 1, 13] and for the furthest site diagram in [13, 11]. Our algorithm is, as any incremental algorithm for planar Voronoi diagrams, not worst case optimal, but it is of practical interest for being on-line and very simple to implement. Moreover, no especial treatment is needed for infinite edges, and for the insertion of sites in the convex hull.

2 The Oriented Projective Plane

In $\mathbb{T}^2$, a point with homogeneous coordinates $[x, y, w]$ is *not* identical to $[-x, -y, -w]$. They are called antipodal points. The antipode of point p is denoted by $\neg p$. The set of points with $w > 0$ is called the *front side* of the plane, and those with $w < 0$ are the *back side*. The set of points with $w = 0$ (except for the invalid triplet $[0, 0, 0]$), which are referred to as improper points, is the line Ω at infinity. In the figures we will use two geometric models for $\mathbb{T}^2$: the flat model, with the usual mapping $[x, y, w] \mapsto (x/w, y/w)$ to Cartesian coordinates, and the spherical model with the mapping $[x, y, w] \mapsto (x, y, w)/\sqrt{x^2 + y^2 + w^2}$. We refer the reader to [15] for an introduction to this geometry.

In order to define the diagrams, we need to consistently operate and compare distance values. This becomes possible when we avoid division in the distance formula and associate each value x/w to a point $[x, w]$ of the oriented projective line $\mathbb{T}^1$. We have $\mathrm{d}_{\mathbb{T}^2} : \mathbb{T}^2 \times \mathbb{T}^2 \to \mathbb{T}^1_+$, defined as $\mathrm{d}_{\mathbb{T}^2}(a, b) = [\sqrt{(a_x b_w - b_x a_w)^2 + (a_y b_w - b_y a_w)^2}, a_w b_w]$. Note that the radical is always positive, so that the range of the function corresponds to one half of each side of

*Partially supported by FINEP/MCT (PRONEX II – SAI – "Sistemas Avançados de Informação", grant 76.97.1022.00).

†Supported by CNPq (grant 139084/96-7) and FAPESP (grant 96/09738-5).

‡Partially supported by CNPq (grant 300157/90-8).

$\mathbb{T}^1$. We call this set $\mathbb{T}^1_+$[8]. To compare two points $a,b \in \mathbb{T}^1$ we apply the rule: $a <_{\mathbb{T}^1} b$ iff $a_w > 0$ and $b_w < 0$; or a_w and b_w have the same sign and $a_x b_w < b_x a_w$. Note that $\mathrm{d}_{\mathbb{T}^2}(a,b) = \mathrm{d}_{\mathbb{T}^2}(b,a) = [x,w]$ and $\mathrm{d}_{\mathbb{T}^2}(\neg b,a) = \mathrm{d}_{\mathbb{T}^2}(b,\neg a) = [x,-w]$. Distances to improper points can also be compared, as described in [9]. Figure 1 shows an example with four segments in the flat (a) and spherical (b) models. Note that the spherical model allows for a better visualization of the segments, and their relative lengths. This figure also introduces our drawing conventions: points are open dots and lines are dashed when they are on the back side, on the flat model; or when they are not visible, on the spherical model.

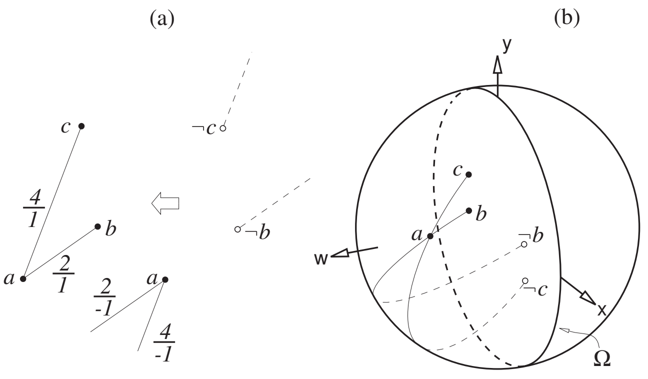

Figure 1: Comparing distances in $\mathbb{T}^2$

We also need a function to add a point in $\mathbb{T}^1$ and a real value, $+ : \mathbb{T}^1_+ \times \mathbb{R} \to \mathbb{T}^1$, defined as $a + v = [a_x + v a_w, a_w]$. Qualitatively, we get a bigger point (distance value) if $v > 0$, and a smaller one if $v < 0$. We hasten to say that we will never compute distances in the algorithm. It will be completely based in the usual determinant predicate which decides the orientation of three points $a, b, c \in \mathbb{T}^2$:

Function 1 `cCW(a, b, c)` *returns* `true` *if* `c` *lies to the left of the oriented line defined by* `a` *and* `b` *and* `false` *otherwise:*

```
return ( ( ax(bycw-cybw) + ay(bwcx-cwbx) +
           aw(bxcy-cxby)) > 0 );
```

2.1 Additively Weighted Voronoi Diagram

Let $\mathcal{S}$ be a finite set of points in $\mathbb{T}^2$, called sites. To each site $p \in \mathcal{S}$ we associate a weight $w(p)$, which can be assumed, without loss of generality, to be positive. For each $p,q \in \mathcal{S}$, let $H_{pq} = \{a \in \mathbb{T}^2 | \mathrm{d}_{\mathbb{T}^2}(a,p) - w(p) \leq_{\mathbb{T}^1} \mathrm{d}_{\mathbb{T}^2}(a,q) - w(q)\}$. The Voronoi face associated to a site p is $R_p = \bigcap_{q \in \mathcal{S}\setminus\{p\}} H_{pq}$. We follow [14] and define the Voronoi diagram $\mathcal{W}(\mathcal{S})$ to be the set of points which belong to more than one Voronoi face. A Voronoi edge is the intersection of two faces and a Voronoi vertex is the intersection of more than two faces. Some previous papers define this diagram using $dist(a,p) + w(p)$ instead. The definitions are equivalent, in the sense that we can multiply the weights of the sites by -1 to obtain the diagram generated by the other definition. This is interesting to note because the part of the diagram contained in the back side is, actually, equivalent to the furthest site diagram generated by the other definition, as $\mathrm{d}_{\mathbb{T}^2}(a,p) - w(p) \leq_{\mathbb{T}^1} \mathrm{d}_{\mathbb{T}^2}(a,q) - w(q)$ iff $\mathrm{d}_{\mathbb{T}^2}(\neg a,p) + w(p) \geq_{\mathbb{T}^1} \mathrm{d}_{\mathbb{T}^2}(\neg a,q) + w(q)$:

$$\begin{array}{rcll}
[x_1, w_1] + (-v_1) & \leq_{\mathbb{T}^1} & [x_2, w_2] + (-v_2) & \Longrightarrow \\
[x_1 - v_1 w_1, w_1] & \leq_{\mathbb{T}^1} & [x_2 - v_2 w_2, w_2] & \Longrightarrow \\
(x_1 - v_1 w_1)(w_2) & \leq & (x_2 - v_2 w_2)(w_1) & \Longrightarrow \\
(x_1 - v_1 w_1)(-w_2) & \geq & (x_2 - v_2 w_2)(-w_1) & \Longrightarrow \\
[x_1 - v_1 w_1, -w_1] & \geq_{\mathbb{T}^1} & [x_2 - v_2 w_2, -w_2] & \Longrightarrow \\
[x_1, -w_1] + (v_1) & \geq_{\mathbb{T}^1} & [x_2, -w_2] + (v_2). &
\end{array}$$

We can regard each site as a *closed disk* with radius equal to its weight. The intersection of two Voronoi faces is known to be one branch of a hyperbola opened towards the smallest disk, and the faces are star-shaped with respect to the sites [14], and these facts still hold in $\mathbb{T}^2$[9]. This diagram is more complex than the unweighted one in the sense that it may have finite faces with only two edges, and two faces may share more than one edge [14]. Figure 2(a) shows the diagram for $\mathcal{S} = \{p,q\}$, while Fig. 2(b) shows the diagram after three other sites were added to $\mathcal{S}$. Note in this example that the diagram is disconnected in $\mathbb{E}^2$. In $\mathbb{T}^2$, the diagram is always connected, and furthermore, since n is the number of faces of the induced planar subdivision and every vertex has valence 3, it has exactly $2n - 4$ vertices and $3n - 6$ edges [10].

The algorithm to be presented is based on two primitive procedures: the calculation of the two *circumcenters* of three given sites; and the counterclockwise circular order predicate of three points b, c and d with respect to a fourth point a. The latter procedure can be implemented with three calls to the function `cCW`, as follows:

Function 2 `countCWO(b, c, d, a)` *returns* `true` *if the counterclockwise circular order of the points* `b`,`c` *and* `d` *around* `a` *is* `bcd` *and* `false` *otherwise:*

```
bool abc = cCW(a, b, c);
bool abd = cCW(a, b, d);
bool acd = cCW(a, c, d);
return ((abc && (!abd || acd)) || (!abd && acd));
```

3 An Incremental Algorithm

We assume that there is no four cocircular disks in $\mathcal{S}$, and that no disk is entirely contained in another disk

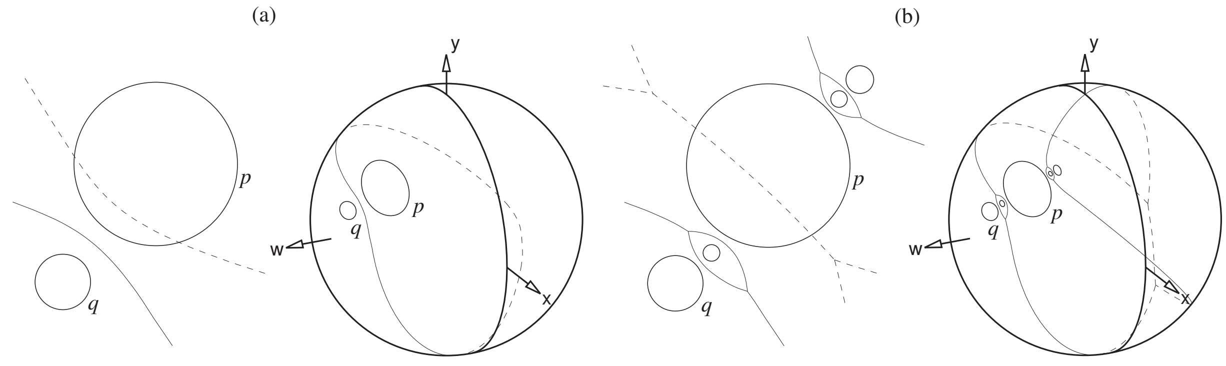

Figure 2: Additively weighted Voronoi diagram in $\mathbb{T}^2$

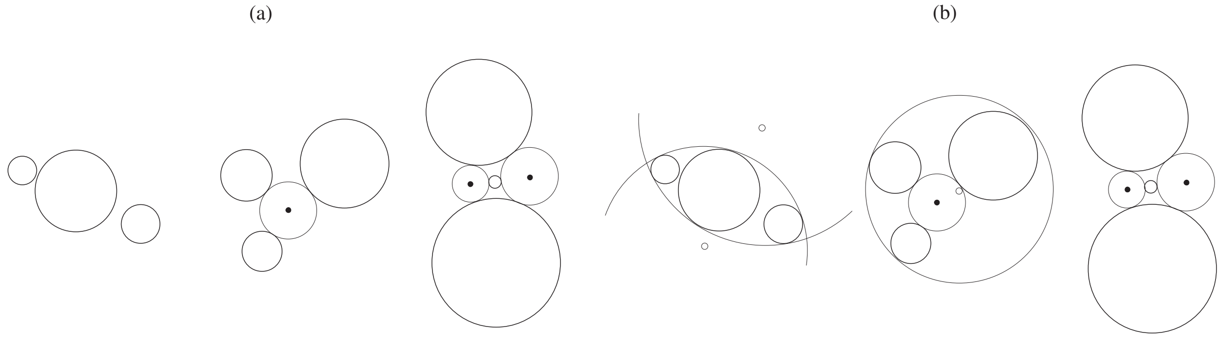

Figure 3: The circumcenters of three sites in $\mathbb{E}^2$ and $\mathbb{T}^2$

(see [10] for a discussion). A *circumcenter* is a point equidistant to three disks. In $\mathbb{E}^2$, three disks may have $0, 1$ or 2 circumcenters (see [13] and Fig. 3(a)). In $\mathbb{T}^2$, there are exactly 2 circumcenters, as Fig. 3(b) shows, for the three possible configurations.

Function 3 `circumCenters(p, q, r, a, b)` *calculates the two circumcenters of the sites* `p,q,r` *storing them in points* `a` *and* `b`.

Computing this function amounts to solving a system of three quadratic equations as suggested in [6]. This calculation can also be viewed as the intersection of three right cones in $\mathbb{E}^3$ as noted in [13]. The details of the implementation of this function can be found in [8].

3.1 Contributing edges

We call *clearance disk* a closed disk tangent to two or three disks of $\mathcal{S}$. Two closed disks are tangent if their intersection is a single point. The diagram can, then, be regarded as the set of the centers of all clearance disks of $\mathcal{S}$ [16]. Given the diagram $\mathcal{W}(\mathcal{S})$ and a new site k (from now on, k always denotes the new site being added to $\mathcal{S}$), the idea of the algorithm is to find the vertices of R_k and, then, update the diagram. Clearly, every vertex of R_k must lie on some edge of $\mathcal{W}(\mathcal{S})$, as it is the center of a clearance disk tangent to k and to two sites of $\mathcal{S}$. We call *contributing edges* the edges of $\mathcal{W}(\mathcal{S})$ which contain at least one vertex of R_k. One edge e may contribute, at most, two vertices, as any vertex lying on e must be a circumcenter of k and the two sites of $\mathcal{S}$ which generate e, and there are exactly two circumcenters.

Let e_{left} and e_{right} be the sites on the left and right faces of the oriented edge e. The preceding discussion implies that a circumcenter of k, e_{left} and e_{right} is a vertex of R_k iff it lies on e. Let e_{orig} and e_{dest} be the vertices at the origin and destination of e. We know that the supporting hyperbola of e is star-shaped with respect to both e_{left} and e_{right}. This implies that a circumcenter a of k, e_{left} and e_{right} (a always lies on the hyperbola) is on e iff the counterclockwise circular order of e_{orig}, a and e_{dest}, with respect to e_{left}, is $e_{orig} a e_{dest}$ (see Fig 5). The algorithm uses the following function:

Function 4 `edgeNotContributing(e)` *returns* `true` *if edge* `e` *does not contribute to* R_k, `false` *otherwise:*

```
circumCenters(e_left, e_right, k, a, b);
return ( !( countCWO(e_orig, a, e_dest, e_left) ||
            countCWO(e_orig, b, e_dest, e_left) ) );
```

At this point, we could design a brute force algorithm which would find all vertices of R_k using the function `edgeNotContributing` to test every edge of $\mathcal{W}(\mathcal{S})$, followed by a circular sorting of the vertices around k (R_k

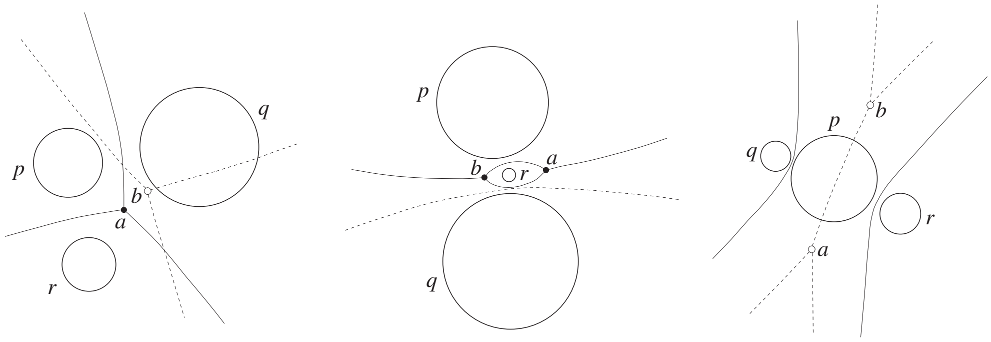

Figure 4: Base cases for $\mathcal{W}(\mathcal{S})$

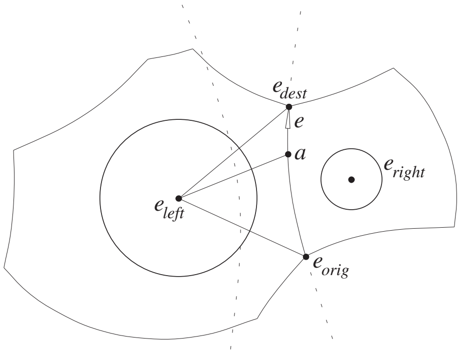

Figure 5: Testing an edge with the orientation predicate

is star-shaped) and, finally, updating the diagram. Better yet, as we will see in Section 3.3, if we define the more specific concept of *oriented contributing edge*, we can find the vertices of R_k consecutively, in the *correct order*, with a very simple traverse procedure.

It is worth noting that in $\mathbb{E}^2$ this characterization through contributing edges is useless, as R_k may have no vertices at all (its boundary may not be connected to $\mathcal{W}(\mathcal{S})$).

3.2 The base case

The base case, three sites, has the three possible configurations shown in Fig. 4. The diagram has always the same topology (as opposed to the Euclidean diagram): the two circumcenters are connected by three edges separating the three sites in three faces. The following functions construct the base case:

Function 5 `createEdge(a, b, p, q)` *creates an oriented edge with origin* `a`*, destination* `b`*, left and right faces* $R_{\mathtt{p}}$ *and* $R_{\mathtt{q}}$*, and returns a pointer to this edge.*

Function 6 `baseCase(p, q, r)` *constructs* $\mathcal{W}(\mathcal{S})$ *for* $\mathcal{S} = \{p, q, r\}$ *and returns a pointer to one of the edges of the diagram:*

```
point a, b;
circumCenters(p, q, r, a, b);
if ( counterCWO(p, q, r, a) ) swap(a, b);
createEdge(a, b, p, q);
createEdge(a, b, q, r);
return createEdge(a, b, r, p);
```

The `if` test is due to a subtle problem: as the faces have only two edges, we cannot circular sort the vertices with respect to the sites. Our test to find the vertices of R_k, already sketched in the previous section, which will be used in the insertion of the other sites, relies on the orientation of the edges. As we choose one of the circumcenters for the origin and create edges having left and right faces respectively R_p and R_q, R_q and R_r, R_r and R_p, there is a *specific* circumcenter for the origin. Consider one of the edges e. If we choose the wrong circumcenter, the counterclockwise circular order of e_{orig}, a generic point x lying on the edge and e_{dest} will not be $e_{orig} x e_{dest}$. It is interesting to note that, as the edges separate the sites and the faces are star-shaped with respect to them, the circular order of the sites with respect to one circumcenter is opposite to the other. The correct origin for the function `createEdge`, which is ensured by the `if` test, must be the circumcenter that produces the counterclockwise circular order prq.

3.3 The traverse procedure

Figure 6(a) introduces the edge functions used in the traverse procedure, which is defined for oriented edges. The names come from the well-known *quad-edge* data structure [5]. For a given oriented edge e: $onext(e)$ is the next edge in counterclockwise direction with the same origin; $rprev(e)$ is the previous edge in counterclockwise direction around the same face; and $sym(e)$ is the opposite oriented edge.

The traverse procedure to be presented needs a first contributing edge. Figure 6(b) shows an example of the insertion of site k in a diagram. The contributing edges are shown in bold lines. Note that the edges con-

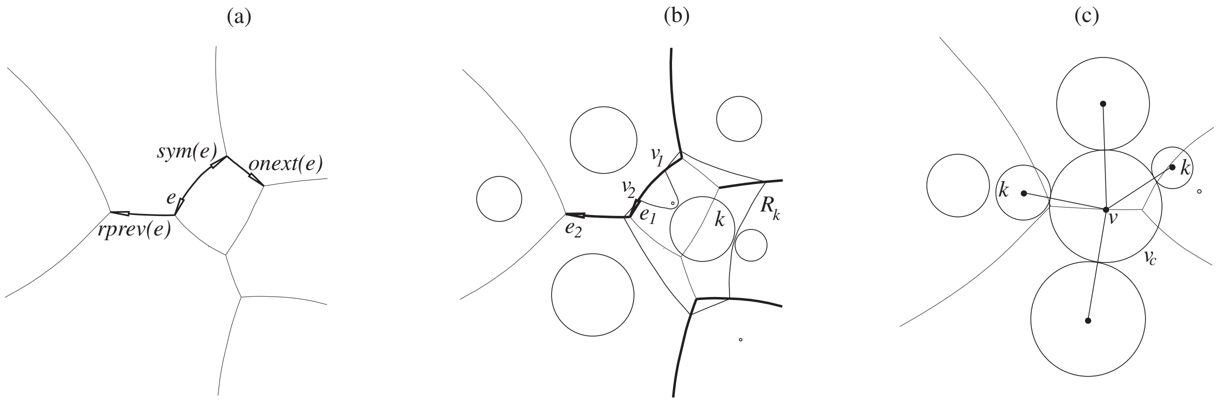

Figure 6: Oriented contributing edges for the traverse procedure

tributing one vertex have one of their endpoints inside R_k and the other outside. The edges contributing two vertices have either both endpoints inside R_k (edge e_1 in the figure), or both outside R_k (in this case, R_k has only two vertices and the procedure works fine without any modifications, see Section 3.3.2). The first contributing edge must be one oriented in such a way that its origin is inside R_k. If this is not true, the procedure will fail to find all vertices. We say, then, that edge e_2 is a contributing edge, but $sym(e_2)$ is not. If an edge contributes one vertex, it is easy to see that the clearance disk with center in the endpoint contained in R_k must intersect k and the one with center in the endpoint outside R_k must not. We need not test the intersection of k with these clearance disks to decide which edge is oriented from inside R_k to outside. Suppose edge e contributes one vertex v as shown in Fig. 6(c). The boundary of the clearance disk v_c with center in v and tangent to k, e_{left} and e_{right} is divided into two arcs by the tangency points in e_{left} and e_{right}. Site k is tangent to v_c in one of these arcs (the figure shows two possible locations for k). The clearance disks between v_c and the clearance disk with center on the endpoint not contained in R_k do not intersect k and those between v_c and the one contained in R_k do intersect. We can, then, use the circular order predicate to decide which is the correct oriented contributing edge. It is the one such that `countCWO(` e_{left}`,` k`,` e_{right}`,` v `) == true`.

For the edges contributing two vertices we *also* use the circular order predicate to assign one vertex to each oriented edge. An oriented edge contributes the first encountered vertex when going from its origin to its destination. As shown in Fig. 6(b), for e_1, `countCWO(` $e_{1\,orig}$`,` v_1`,` v_2`,` $e_{1\,left}$ `) == true`, and, for $sym(e_1)$, `countCWO(` $sym(e_1)_{orig}$`,` v_1`,` v_2`,` $sym(e_1)_{left}$ `) == false`. Then, e_1 contributes v_1 and $sym(e_1)$ contributes v_2.

The vertex contributed by an edge e is the point where the boundary of R_k crosses e leaving the face of e_{right} and entering the face of e_{left}, in counterclockwise direction with respect to k. We need to find the next vertex, which is the point where the boundary leaves the face of e_{left}. The procedure tests the edges of $R_{e_{left}}$ in clockwise direction from e:

Function 7 `traverseProcedure`

```
firstEdge = e = findSomeContributingEdge();
do { processContributingEdge( e ); e = onext( e );
     while ( e != firstEdge &&
             edgeNotContributing( e ) )
           e = rprev( e );
} while ( e != firstEdge );
```

We have implemented the function `findSomeContributorEdge` by testing every edge sequentially until `edgeNotContributing` returns `false`. This sequential test does not affect the worst-case time complexity of the algorithm, $O(n^2)$ (since R_k may have $O(n)$ edges). In practice, however, it may be the major factor in the running time, and some heuristic should be used instead. One option is the simple walking method suggested in [4], which can be used, because the face of the site of $\mathcal{S}$ closest to k must have a contributing edge. More elaborated heuristics exist (see [2] for instance).

Figure 7(a) shows, in bold lines, the edges traversed by the procedure for the example of Fig. 6(b). The function `processContributingEdge` is called for every oriented contributing edge and updates the diagram. The update is very simple (see Fig. 7(b) and (c)): the origins of the oriented contributing edges are changed to the corresponding vertices; consecutive vertices are joined by a new edge; and, in the end, some edges of $\mathcal{W}(\mathcal{S})$ may need to be deleted.

3.3.1 Correctness of the traverse procedure

The correctness of the procedure is based on the star-shape property of the faces of $\mathcal{W}(\mathcal{S})$. The function `findSomeContributingEdge` returns one contributing

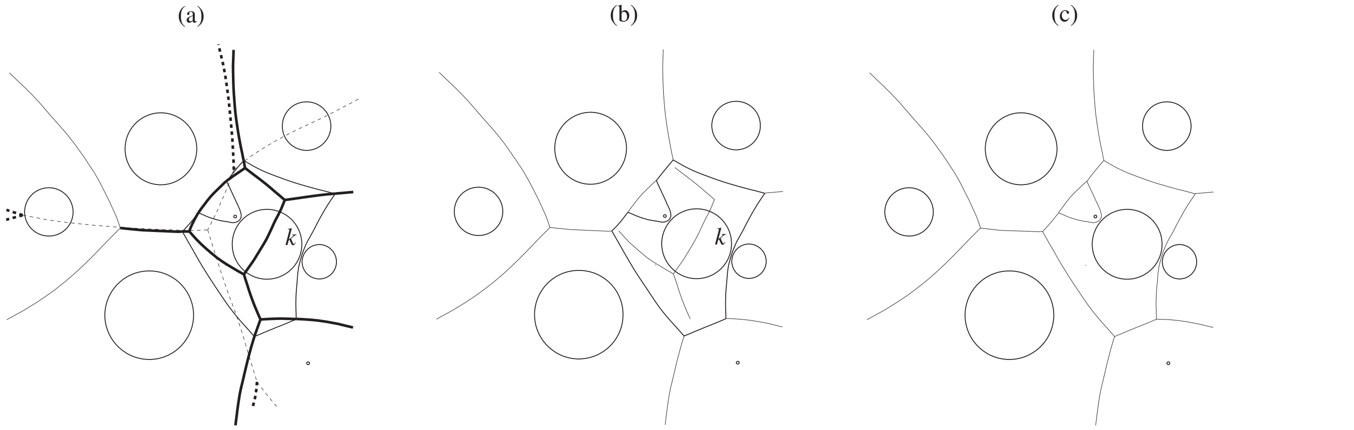

Figure 7: The traverse procedure updating the diagram

edge e (there must be at least one such edge, since R_k *has* vertices in $\mathbb{T}^2$). The vertex contributed by the edge e is the point where the boundary of R_k crosses e leaving the face of e_{right} and entering the face of e_{left}, in counterclockwise direction with respect to k. The `traverseProcedure` will test all edges of $R_{e_{left}}$ in clockwise direction. Denote these edges by $e_1, e_2, \ldots, e_i$ as in the Fig. 8(a). The boundary of R_k must leave $R_{e_{left}}$, thus, the procedure will eventually find another vertex. Note that the boundary of R_k may enter and leave one face several times, so that it is not obvious that the procedure always finds the correct vertex where the boundary leaves the face.

The argument that shows the correctness of the procedure, however, is indeed simple. If the boundary leaves $R_{e_{left}}$ through e_1 then the procedure correctly finds the vertex as e_1 is the first tested edge. But suppose e_1 contributes two vertices v_1 and v_2 (see Fig. 8(a)). We saw that v_1 is assigned to e_1, which is correct because the boundary cannot leave the face through v_2 as it would have to enter it some time again through v_1 and would not be able to leave it.

Generally, if the boundary leaves $R_{e_{left}}$ through e_j, then none of the edges $e_1, e_2, ..., e_{j-1}$, which are tested before e_j, can be a contributing edge and the procedure will reach e_j. This is due to the fact that every part of the boundary crossing $R_{e_{left}}$ is a part of the bisector of k and e_{left}. If any of $e_1, e_2, ..., e_{j-1}$ could be a contributing edge, then $R_{e_{left}}$ would become disconnected, which is impossible (see Fig. 8 (b)). This also implies that every traversed edge, except the contributing edges, is entirely contained in R_k.

When we defined oriented contributing edges, we said that their origins must be contained in R_k. We invite the reader the follow the `traverseProcedure`, for the example of Fig. 6(b), choosing $sym(e_2)$ as the first oriented contributing edge (which has its origin outside R_k). The vertices v_1 and v_2, in that figure, are just not found. In this case, the arguments above can not be applied because the traversed edges are not inside R_k. We could, alternatively, traverse the faces in counterclockwise direction to find the vertices in clockwise direction. The requirement would be, then, that the contributing edges always have their origins outside R_k.

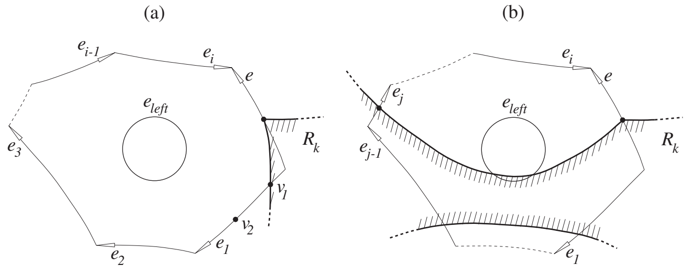

Figure 8: Correctness of the traverse procedure

3.3.2 A special case

When R_k has only two edges (and two vertices), there is only one edge e of $\mathcal{W}(\mathcal{S})$ contributing the two vertices. This is a special case, in the sense that the two oriented contributing edges have their origins *outside* R_k. However, the traverse procedure correctly finds the two vertices. Figure 9(a) shows the edges traversed by `traverseProcedure` in an example.

The update differs from the general case: choose one oriented contributing edge e and change its destination to the corresponding vertex; create a new edge from the origin to the corresponding vertex of the other oriented contributing edge; and join the vertices by two edges separating k from e_{left} and e_{right}. Figure 9(b) shows the resulting diagram. This case is similar to our base case, but here there is no need for an `if` test, as the orientation of the contributing edges are already consistent.

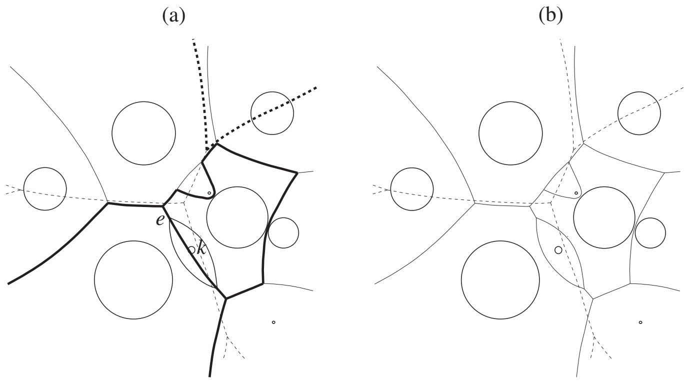

Figure 9: Special case when updating the diagram

4 Conclusion

The subtlety of this algorithm is the fact that the primitives encapsulate all concepts of $\mathbb{T}^2$. The function `circumCenters` returns two points of $\mathbb{T}^2$ which are used by the function `edgeNotContributing` in simple orientation tests (which are defined for all points in $\mathbb{T}^2$). The traverse procedure and the update procedure need not distinguish finite and infinite edges, or the front and the back parts of the diagram. Once the closest site diagram in $\mathbb{T}^2$ is constructed, the closest site and the furthest site diagrams in $\mathbb{E}^2$ can be retrieved by simply ignoring, respectively, the back and the front side of the plane.

We have successfully implemented this algorithm, and the details can be found in [8]. Figure 10 is an image generated by the implementation showing the front and the back parts of a diagram which, in $\mathbb{E}^2$, would be disconnected, and whose corresponding furthest site diagram would have a disconnected face.

One interesting generalization of this construction in $\mathbb{T}^2$ is the higher order Voronoi diagram. In [12], the order k diagram is shown to be antipodal to the order $(n-k)$ diagram, and an algorithm, which also profits from $\mathbb{T}^2$, is presented for constructing the diagrams of all orders.

Lastly, we note that any topological data structure able to represent partitions of the sphere can be used to represent the diagram in $\mathbb{T}^2$. Fortunately, the geometric representation of the edges, which are arcs of hyperbolas in $\mathbb{T}^2$, also have a simple and uniform representation. This subject has independent interest and appears in [9].

References

[1] F. Aurenhammer, Power diagrams: properties, algorithms and applications, SIAM J. Comput. 16 (1) (1987) 78–96.

[2] J-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud and M. Yvinec, Applications of Random Sampling to On-line Algorithms in Computational Geometry. Discrete & Comp. Geometry 8 (1992) 51–71.

[3] S. Fortune, A sweepline algorithm for Voronoi diagrams, Algorithmica 2 (1987) 153–174.

[4] P. J. Green and R. Sibson, Computing Dirichlet tessellations in the plane, Computer Journal 21 (1977) 168–173.

[5] L. Guibas and J. Stolfi, Primitives for the manipulation of general subdivisions and the computations of Voronoi diagrams, ACM Transactions on Graphics 4 (2) (1985) 74–123.

[6] M. Held, On the computational geometry of pocket machining, Lectures Notes in Computer Science 500 (1991).

[7] D. T. Lee and R. L. Drysdale, Generalization of Voronoi diagrams in the plane, SIAM J. Comput. 10 (1) (1981) 73–86.

[8] G. A. Pinto, Generalizações do diagrama de Voronoi construídas através de cônicas no plano projetivo orientado e suas visualizações, Master's Thesis, Institute of Computing, UNICAMP (1998).

[9] G. A. Pinto and P. J. de Rezende, Representation of conics in the oriented projective plane, Proc. of the X SIBGRAPI, published by IEEE, Campos do Jordão, Brazil (1997) 71–78.

[10] G. A. Pinto and P. J. de Rezende, Additively weighted Voronoi diagram on the oriented projective plane, Technical Report IC-00-03, Institute of Computing, UNICAMP (2000).

[11] D. Rappaport, A convex hull algorithm for discs and applications, Computational Geometry: Theory & Applications 1 (1992) 171–187.

[12] P. J. de Rezende and R. B. Westrupp, An Optimal Algorithm to Construct All Voronoi Diagrams for k Nearest Neighbor Search in $\mathbb{T}^2$, Proc. of the XII SIBGRAPI, published by IEEE, Campinas, Brazil (1999) 7–15.

[13] H. Rosenberger, Order-k Voronoi diagrams of sites with additive weights in the plane, Algorithmica 6 (1991) 490–521.

[14] M. Sharir, Intersection and closest-pair problems for a set of planar discs, SIAM J. Comput. 14 (2) (1985) 448–468.

[15] J. Stolfi, Oriented projective geometry: a framework for geometric computations, 1st edition, Academic Press, Inc. (1991).

[16] C. K. Yap, An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments. Discrete & Comp. Geometry 2 (1987) 365–393.

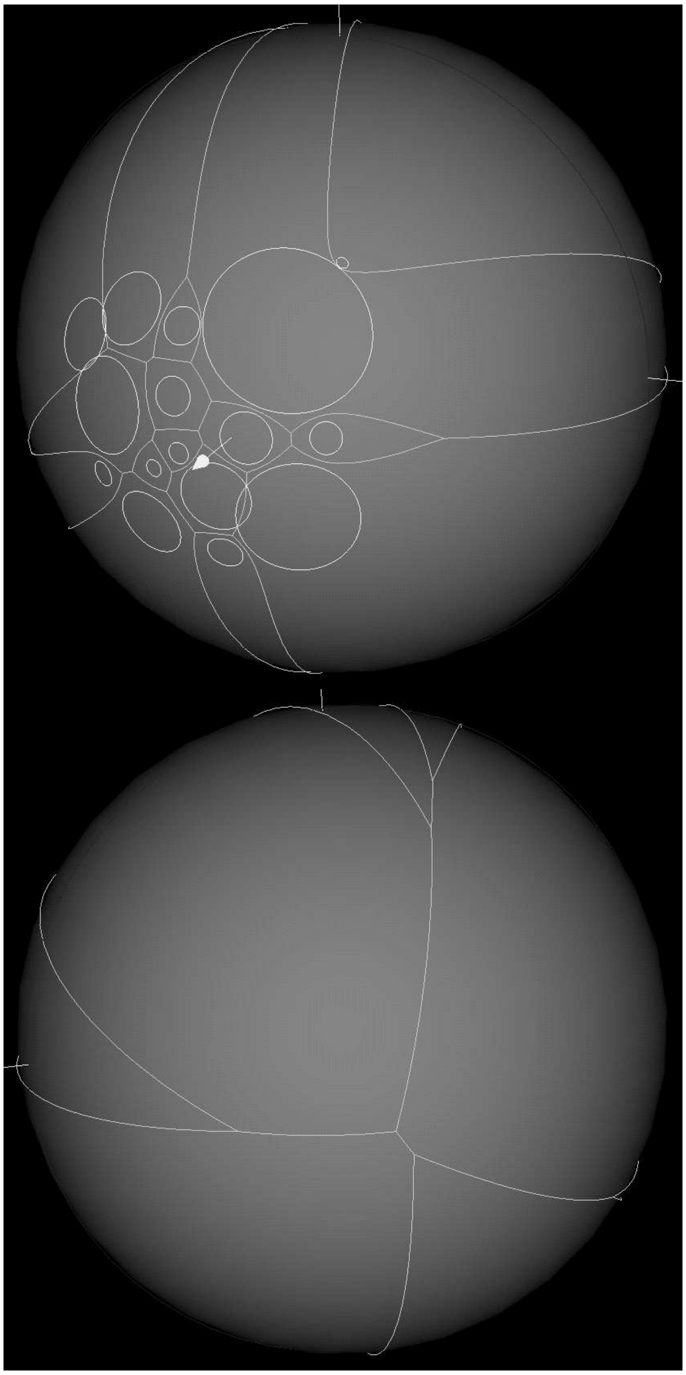

Figure 10: A diagram in the spherical model of $\mathbb{T}^2$