An Efficient Implementation of Fortune's Plane-Sweep Algorithm for Voronoi Diagrams

Kenny Wong Hausi A. Müller*

Department of Computer Science
University of Victoria, Victoria, BC, V8W 3P6, Canada

Abstract

The Voronoi diagram is a fundamental geometric object for encoding proximity information. An efficient C implementation of Fortune's plane-sweep algorithm for constructing the Voronoi diagram and the Delaunay triangulation of a set of points in the plane is presented. Data structure representations and geometric primitives are described. The implementation can produce the Voronoi diagram of 10000 sites in about four seconds on a SPARCstation IPC. Functions to compute nearest neighbors, the closest pair, and the convex hull are included in the distribution package.

1 Introduction

The Voronoi diagram is an important geometric object which has many practical uses in diverse fields. Figure 1 depicts the planar Voronoi diagram of a set of point sites. The region around each site contains all those points for which that site is the nearest among all sites. The edges, called *Voronoi edges*, contain points which are equidistant between two neighboring sites. The edges intersect at points called *Voronoi vertices*, which are equidistant to three or more neighboring sites. The diagram encodes proximity information and is useful in studying a physical system whose properties depend on the spatial layout of its entities. Moreover, it can be used in solving various geometric problems, including finding the nearest neighbor for each site, computing the closest pair of sites, and producing the convex hull [8]. Closely related is a dual diagram called the Delaunay triangulation, which has applications in surveying, contouring, and fast geometric travelling salesperson problems [9].

*This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and the British Columbia Advanced Systems Institute.

There are many algorithms for constructing the Voronoi diagram [2]. However, Fortune's algorithm [4] has the combined advantages of being optimal like divide-and-conquer algorithms (i.e., $O(N \log N)$ for N sites) and being relatively simple like incremental algorithms by avoiding a difficult merge step. It conceptually sweeps a straight line or *sweepline* across the set of sites, constructing the Voronoi diagram behind this moving front. This technique is generically called *plane-sweep* and appears in other geometric algorithms [5, 6, 11].

There are two fundamental plane-sweep data structures: the *event queue* and the *sweep table*. General forms of these data structures are known as priority queues and ordered dictionaries. The event queue contains a sequence of lexicographically ordered points or events which define where the sweep must momentarily stop. This queue is updated as the sweep proceeds. The sweep table maintains a cross-section of how the sweepline intersects the regions and edges of the Voronoi diagram. This table is updated at each event. Efficient representations for these data structures are described.

Fortune's algorithm involves a geometric transform and computes a transformed diagram. The primal diagram and the Delaunay triangulation can be computed simultaneously [4, 12]. Efficient geometric primitives for implementing the required numerical calculations in floating-point arithmetic are presented.

Our implementation, which we call Tess, is written completely in C and has been tested on Sun 3, Sun 4 (SPARC), RS/6000, NeXT, and SGI 4D architectures with the supplied C compilers. (The header files are also compatible with C++.) This implementation can produce the Voronoi diagram of 10000 sites in about four seconds on a SPARCstation IPC using double-precision floating-point arithmetic. The source code is too long to be listed here; refer to the distribution files. See the `README` file for installation instructions and usage information.

We developed Tess before discovering Fortune's own implementation in the NETLIB archives. Naturally, both implementations share some common design ideas. For example, they both presort the set of sites and separate sites from vertices in the event queue. Nevertheless, there are some major differences. First, Fortune's event queue and sweep table are represented by linked lists with an intricate bucketing scheme to allow relatively efficient searches. The performance of this scheme is sensitive to the bucket size. Tess uses an implicit heap and a threaded binary tree with randomization. The efficiency of these data structures is comparable to Fortune's scheme and arguably simpler. Second, Tess provides a compact, doubly-connected edge list representation [8] for holding the computed Voronoi diagram. Unlike Fortune's approach, this representation captures all the incidence relationships among the Voronoi edges and vertices. Finally, Fortune's implementation was not designed to be a library routine; it is a stand-alone program. As such, it does not clean up and free memory after processing. Tess is a library routine. We supply drivers that demonstrate its use.

Section 2 of this paper presents some generally accepted definitions and some needed new terminology. Section 3 briefly outlines Fortune's algorithm. The implementation details are divided between the next two sections. Sections 4 and 5

describe the data structures and the geometric primitives. Section 6 presents some performance characteristics.

2 Preliminaries

Let $\Re$ denote the set of real numbers and $\Re^2$ the real plane. For a point $p \in \Re^2$, let p_x and p_y be its x- and y- Cartesian coordinates. For points p and $q \in \Re^2$, $p < q$, if $p_y < q_y$ or $p_y = q_y$ and $p_x < q_x$. A site is a point in $\Re^2$. Throughout this paper, we deal with a fixed set S of N sites. For a site $p \in S$ and an arbitrary point $z \in \Re^2$, $d_p(z)$ is the Euclidean distance between z and p. Moreover, $d(z)$ is the Euclidean distance from z to the nearest site. The perpendicular *bisector* B_{pq} (or equivalently, B_{qp}) of the line segment joining two distinct sites p and $q \in S$ is $\{z \in \Re^2 \mid d_p(z) = d_q(z)\}$, a line which divides the plane into two half-planes, H_{pq} (containing p) and H_{qp} (containing q). All points in H_{pq} are closer to p than q, and vice versa for H_{qp}. Formally, for sites p and $q \in S$, H_{pq} is $\{z \in \Re^2 \mid d_p(z) < d_q(z)\}$, an open convex set.

For a site $p \in S$, the *Voronoi region* R_p is the set of all points in $\Re^2$ strictly closer to p than any of the other $N - 1$ sites in S. Succinctly, $R_p = \bigcap_{q \in S, q \neq p} H_{pq}$. Thus, the region R_p is an open, convex, polygonal set which contains p and may be bounded or unbounded. When unbounded, R_p has a polygonal border defined by a finite set of line segments and exactly two rays. Two sites are called *neighboring* if the borders of their Voronoi regions share a point. The *Voronoi diagram* V is $\Re^2 - (\bigcup_{p \in S} R_p) = \{z \in \Re^2 \mid \exists p, q \in S, p \neq q, d_p(z) = d_q(z) = d(z)\}$. Every point in V is equidistant from two or more distinct neighboring sites. Intuitively, V consists of the polygonal borders of the Voronoi region for each site. Thus, V is the union of line segments and rays (or lines if all sites are collinear). These line segments and rays in V are subsets of bisectors. A *Voronoi vertex* is a point in V that is equidistant from three or more distinct neighboring sites and thereby incident to three or more line segments or rays in the Voronoi diagram. If p, q, and r are any three distinct non-collinear sites, then the intersection of B_{pq}, B_{qr}, and B_{rp} is a unique point corresponding to a *candidate vertex*. This intersection is also the center of a *candidate circle* defined by the sites p, q, and r on its circumference. A candidate circle that has no other sites located strictly in its interior is called a *Voronoi circle* and has a Voronoi vertex at its center. Note that there may be more than three sites on its circumference. Informally, a *Voronoi edge* is a maximal, connected subset of V which is either: a line segment between two Voronoi vertices, or a ray emanating from a Voronoi vertex, or a bisector of two sites. Because every Voronoi edge is a subset of a bisector, it has two defining sites—the two nearest sites to which the interior points of the edge are equidistant. The Voronoi edge defined by these two sites, say p and q, is denoted by e_{pq} (or e_{qp}). If e_{pq} is a ray, then p and q are hull points of the convex hull of S and R_p and R_q are unbounded regions. Because of a duality between the Voronoi diagram and the Delaunay planar triangulation, there are $O(N)$ edges and vertices in a Voronoi diagram for N sites [8].

There is a complication in directly applying the plane sweep technique for constructing the Voronoi diagram. The Voronoi region for a site is intersected by the sweepline (and hence, the cross-section in the sweep table should be updated to

include this region) *before* the site is encountered. We cannot know which region needs to be recorded in the sweep table without seeing the site first. Fortune proposed a geometric transformation or mapping to overcome this problem [4]. The solution computes a transformed Voronoi diagram in which the earliest point of a region encountered (intersected) by the sweepline is at the site itself. Intuitively, the rest of the region is deflected away (actually upward in the plane) with the site as its earliest encountered (lowest) point. Thus, a region, albeit transformed, is processed when the site occurs and not before. The primal Voronoi diagram can be computed simultaneously. The transformation can be implemented efficiently.

The transformation proposed by Fortune involves the continuous mapping $*$ from $\Re^2$ to $\Re^2$ defined by: $*(z) = (z_x, z_y + d(z)), \forall z \in \Re^2$. Within the algorithm, this mapping is applied to the bisector of two neighboring sites. Hereafter, consider only bisectors of such sites. Under $*$, an edge of V becomes part of a branch of a hyperbola unless it is orginally vertical. Several properties of the mapping are discussed in Fortune's paper. The sites stay fixed under $*$. If e_{pq} is an edge of V, then $*(e_{pq})$ is a section of a hyperbola or a vertical line. The unique lowest point of a region $*(R_p)$ is p. The mapping $*$ is one-to-one on V. The mapping preserves incidence relationships of Voronoi edges and vertices; when two edges meet, their transformed edges also meet. Furthermore, hyperbolic branches corresponding to two different bisectors intersect at most once because the two bisectors themselves can only intersect at most once. We casually refer to Voronoi regions, edges, and vertices in the transformed diagram $*(V)$ although, strictly speaking, we should refer to the *images* of regions, edges, and vertices in V under $*$.

Consider the effect of $*$ on a bisector B_{pq} of two neighboring sites p and q. There are two cases: B_{pq} is non-vertical or B_{pq} is vertical. Without loss of generality, assume $p \geq q$. First, if $p_y \neq q_y$ and B_{pq} is non-vertical, then $*(B_{pq})$ is the upwardly opening branch of a hyperbola with p at its minimum. The branch $*(B_{pq})$ splits into left and right hyperbolic rays called *minus* and *plus boundaries*: C^-_{pq} monotonically decreasing to the left of p and C^+_{pq} monotonically increasing to the right of p. This scheme also splits B_{pq} into left and right rays. The lowest point of a boundary is called its *minpoint*. Here, p (the greater of the two sites) is the minpoint of C^-_{pq} and C^+_{pq}. Second, if $p_y = q_y$ and B_{pq} is vertical, then $*(B_{pq})$ is a vertical, upwardly directed ray rooted at the midpoint of a line segment between p and q, $((p_x+q_x)/2, p_y)$. Define $C^0_{pq} = *(B_{pq})$ and call it a *vertical* boundary. Here, the minpoint is the midpoint. Let C_{pq} denote one of C^-_{pq}, C^+_{pq}, or C^0_{pq} when the choice of *direction* ($-$, $+$, or 0) is irrelevant to the discussion. These boundaries are equivalently expressed as C^-_{qp}, C^+_{qp}, or C^0_{qp}.

A *boundary ray* is an unbounded, connected subset of a boundary. The lowest point of a boundary ray is called its *base* (this point may or may not correspond to the minpoint). A *boundary segment* is a bounded, connected subset of a boundary. The lowest point of a boundary segment is also called its *base*. The highest point is called its *summit*. We also refer to these subsets using the same notation as for boundaries (i.e., C^-_{pq}, C^+_{pq}, or C^0_{pq}). Note that a horizontal line intersects a boundary ray or segment at most once. The sweep table in Fortune's algorithm contains a record of the transformed regions and boundary rays intersected by the horizontal

sweepline.

3 Algorithm Outline

This section outlines the main parts of Fortune's algorithm in fairly abstract terms. A detailed presentation is provided in Fortune's paper [4]. Implementation details are deferred until Sections 4 and 5. Usually, the sites, regions, candidate vertices, and edges mentioned in this section are in the transformed space (i.e., in the diagram $*(V)$). It should be clear from context which diagram we are considering. The input is given as a set S of distinct sites; the output is a set of transformed Voronoi edges and vertices in $*(V)$. Let us call Fortune's Voronoi algorithm $\mathcal{F}$. This section ends by describing the construction of the primal diagram V.

$\mathcal{F}$ creates semi-infinite boundary rays and marks them with transformed Voronoi vertices to form boundary segments. (Some rays may remain unmarked.) These boundary segments and rays map to connected subsets of Voronoi edges in the primal space. The two main data structures are an event queue and a sweep table. The event queue is a priority queue, denoted by Q, which contains the transformed sites and candidate vertices to be processed in lexicographic order. The lexicographically smallest *event* is retrieved (and the conceptual sweepline advances to that point). The sweep momentarily stops at a site or candidate vertex to update the sweep table, an ordered list, denoted by T, of semi-infinite boundary rays and regions intersected by the sweepline. These *objects* are totally ordered by the x-coordinates of their intersection with the sweepline.

$\mathcal{F}$ ensures that a vertex event retrieved from Q is a Voronoi vertex by deleting candidate vertices from Q whose circle properly contains a site. This non-empty circle testing is implicit in the algorithm. $\mathcal{F}$ detects a candidate vertex in $*(V)$ by any intersection of a boundary ray (say, C_{qr}) with an adjacent boundary ray (say, C_{rs}) in T. This intersection, if it exists, is always on or above the sweepline, by the monotonicity of boundary rays. Three distinct sites q, r, and s are involved; they define a candidate circle with a vertex v at the center and $*(v)$ at the top. The pseudocode for $\mathcal{F}$ is outlined in Figure 2. We modify the initialization steps to properly handle the case of non-unique bottommost sites.

Fortune verified the algorithm's correctness assuming exact arithmetic and its $O(N \log N)$ performance given suitable data structures [4]. Under exact arithmetic, $\mathcal{F}$ works correctly even in the presence of degeneracies in the placement of the sites. A degeneracy is four or more cocircular sites, three or more collinear sites, or a site lying on a bisector. Degeneracies cause zero-length edges to be produced. A degeneracy is detectable if candidate vertices coincide. These coinciding vertices may be processed in any order.

The construction of the primal diagram V is now described in detail. The process involves recovering the inverse image of a boundary ray or segment. The primal diagram V is the union of all the inverse images for every boundary ray or segment ever created by $\mathcal{F}$. These inverse images may be bounded or unbounded.

Furthermore, this process constructs V simultaneously with $*(V)$, as claimed by Fortune.

The inverse image of a boundary ray C_{pq} is a connected subset of the bisector B_{pq}. Assume that $p > q$ and denote the base of C_{pq} by λ. Let μ be the minimal y-coordinate of the sites. Deriving the inverse of λ, denoted λ^{-1}, involves three mutually exclusive cases. First, if λ is a transformed vertex, then λ^{-1} is the corresponding primal vertex. Second, if C_{pq} is a non-vertical boundary ray and λ is its minpoint, then λ^{-1} is the point on B_{pq} directly under p. Third, if C_{pq} is an initial vertical boundary ray, then λ is the point $((p_x + q_x)/2, \mu)$ and λ^{-1} is $((p_x + q_x)/2, -\infty)$. We need not test for these cases explicitly; λ^{-1} is known when the boundary ray is created. That is, the first case occurs when handling a candidate vertex, the second when handling a site, and the third when handling non-unique bottommost sites. If C_{pq} is marked, then it has a summit σ which is always a transformed vertex. The inverse of σ, denoted by σ^{-1}, is the corresponding primal vertex. The desired subset of B_{pq} is that section between λ^{-1} and σ^{-1} (which might be a ray because of the third case). This subset or inverse image can be output as soon as C_{pq} is marked. If C_{pq} is not marked and $p_y \neq q_y$, the desired subset of B_{pq} (either the left or right ray from λ^{-1}) depends on the direction (minus or plus) of C_{pq}. If C_{pq} is not marked and $p_y = q_y = \mu$, the desired subset is all of B_{pq}. If C_{pq} is not marked and $p_y = q_y \neq \mu$, the subset is the upward ray along B_{pq} from λ^{-1}. Thus, V can be constructed simultaneously with $*(V)$, without an extra phase.

The calculation for λ^{-1}, if it is the point on B_{pq} directly under p, is unnecessary. To form the Voronoi edge, e_{pq}, we end up generating *two* subsets of B_{pq} with λ^{-1} as their common point. The solution to this minor inefficiency produces the edge using one subset and avoids computing the coordinates of λ^{-1}. Since this problem arises only when handling a site, two boundaries C^-_{pq} and C^+_{pq} are involved. Thus, we mutually link the two boundaries so that changes in one are noted in the other, and vice-versa. The Voronoi vertex associated with the summit of the first boundary to be marked is recorded as an attribute of the other boundary. The construction of the desired bisector subset depends, now, on what happens to the second boundary. Let t be the above primal vertex and let C_{pq} be the second boundary. There are two cases. First, if C_{pq} gets marked by a transformed vertex with an associated primal vertex v, then the resulting subset of B_{pq} is that segment between t and v. Second, if C_{pq} remains unmarked, then the subset is the ray along B_{pq} from t which is directed toward the left or right depending on the direction of C_{pq} (minus or plus). Furthermore, if neither of the two boundaries are marked at all, then the resulting subset is all of B_{pq}. Thus, exactly one bisector subset is produced for each Voronoi edge in V. (These steps are necessary for the diagram output representation we use.)

The Voronoi edges, once produced, are not used again by the algorithm. This is a practical advantage if main memory is limited; the partially constructed diagram can be saved to disk during the sweep. However, in general, the unbounded edges of V must wait until the end of the algorithm when the remaining unmarked boundary rays in the sweep table are enumerated.

4 Data Structures

This section is the first of two implementation sections. Our implementation is referred to as Tess. The event queue and sweep table data structures are described. The data structures are realized as abstract data types that are independent of the algorithm; they have no notion of boundary rays or candidate vertices, for example. This abstract approach has the added benefit of allowing performance improvements to the algorithm by simply changing the underlying data structure representations to more efficient alternatives. We also describe the output data structure that Tess uses to represent the constructed Voronoi diagram.

4.1 Event Queue

The event queue is realized with an abstract data type, called EQ, which is similar to a priority queue (see also the `eq.h` file in the distribution). Let Q denote an instance of EQ. Here are the pertinent access functions. In this context, a handle is a unique reference to an event in the event queue. (Tess does not use EQ_Size, but it is provided for completeness.)

$Q \leftarrow$ EQ_New()	Q gets a new, empty event queue
$Q \leftarrow$ EQ_Renew(q)	Q gets an empty event queue by clearing q
EQ_Free(Q)	dispose of Q
$i \leftarrow$ EQ_Size(Q)	return the number of events in Q
$b \leftarrow$ EQ_IsEmpty(Q)	return whether Q is empty
$e \leftarrow$ EQ_EventAt(Q, h)	return the event in Q at handle h
$e \leftarrow$ EQ_Min(Q)	return the minimum event in Q
$h \leftarrow$ EQ_Insert(Q, p, e)	insert an event e at location p into Q and return its handle
$e \leftarrow$ EQ_DeleteMin(Q)	delete and return the minimum event from Q
$e \leftarrow$ EQ_Delete(Q, h)	delete and return the event from Q at handle h
EQ_Replace(Q, h, p, e)	replace the event in Q at handle h with an event e at location p

To implement the event queue, Tess uses an implicit heap, an array representation of a complete binary tree where a heap-order condition is maintained. To simulate efficient adjustment of the heap, an array of pointers to heap records is used. (An index field in each record is needed for properly updating this pointer array.) An event is represented by a generic pointer in each heap record. Thus, the heap representation makes no assumption on the data types of the events it must manage. A handle is actually a pointer which allows immediate access to a heap record for efficient deletions and replacements of events. Clients should not make use of this fact, however, and access the fields of a heap record directly. Moreover, there is no fixed limit on the number of events allowed; the pointer array is extended if it gets full. We also considered and evaluated sorted arrays and splay trees as alternative representations, but the implicit heap provided the best combination of performance and simplicity.

Since the sites are known in advance and are fixed under $*$, it is not necessary to mix sites and candidate vertices as events in Q. Thus, Tess separates them into two different data structures and uses a merging scheme to extract the next event to be

processed. The sites are kept in a sorted array; the candidate vertices are kept in the described EQ data structure. From experiments on uniformly distributed sites, the average maximum number of candidate vertices at any moment for 10000 sites is about 150 (and seems to be $O(\sqrt{N})$). Separating the two kinds of events gives about a 15 percent speed increase. To produce the sorted array, Tess calls an efficient quicksort routine [10]. An index into the sorted array is maintained for the next site to be processed. A site and its corresponding region is uniquely represented by its index in this array. Also, lexicographic comparisons of site locations are made more efficient by comparing only the integer indices of the sites (versus the real-valued coordinates).

4.2 Sweep Table

The sweep table is realized with an abstract data type, called ST, an ordered dictionary without the membership operation (see also the `st.h` file). Let T denote an instance of ST. Here are the access functions. In this context, a handle is a unique reference to an object in the sweep table.

$T \leftarrow$ ST_New()	T gets a new, empty sweep table
$T \leftarrow$ ST_Renew(t)	T gets an empty sweep table by clearing t
ST_Free(T)	dispose of T
$x \leftarrow$ ST_ObjectAt(T, h)	return the object in T at handle h
$x \leftarrow$ ST_Before(T, h)	return the object preceding the object in T at handle h
$x \leftarrow$ ST_After(T, h)	return the object following the object in T at handle h
$h \leftarrow$ ST_PlaceHolder(T, s)	return the placeholder handle for sentinel object s in T
$h \leftarrow$ ST_Insert(T, h_1, h_2, x)	insert an object x into T between the objects at handles h_1 and h_2, and return its handle
ST_Delete(T, h)	delete the object from T at handle h
ST_Replace(T, h, x)	replace the object in T at handle h with an object x
ST_InOrder(T, f, z)	call function f with z as client data on each object in T, in ascending order
$x \leftarrow$ ST_Search(T, g, d)	search for and return an object in T using a binary decision function g, and set d to the final decision

The ST_Search function is somewhat unusual in that it does not search for an object with a given key; instead, it performs something like a binary search by following the decisions made by a function g. This user-defined function accepts an object and assumes that all the objects are totally ordered (which they are). It returns whether to jump further back (-1) or further forward (+1) in this sequence of objects when continuing the search. The search (or elimination process) continues until one object is left, and that object is returned along with the final decision (-1 or +1).

To implement the sweep table, Tess uses a binary search tree. The insertion and deletion routines use randomness to reduce skewing. The tree is threaded and has one object associated with each node. Each node has parent and children pointers as well as pointers to the next and previous nodes in an inorder traversal of the tree. The handle for an object is just a pointer to its corresponding node. These pointers allow fast accesses to adjacent nodes as well as worst case constant time insertions, deletions, and replacements, giving the tree some of the local

manipulation properties of a doubly-linked list. For an insertion (between two given adjacent objects), the new node becomes either the right child of the first node or the left child of the second node. This choice depends on whether the link is possible and a simulated coin flip. Insertions and deletions are fast because the search, which would normally take place in a binary tree to insert or delete a node, is decoupled and placed in the separate search function. That is, we exploit the locality of accesses into the sweep table to avoid unnecessary searching. Furthermore, unused nodes are entered on a free list to reduce storage management overhead.

This representation works well in practice and is markedly better than a binary tree without randomness. We considered and evaluated array and randomized search tree (treap) [1] representations, but the simple binary tree gave the best performance. The treap provided more efficient searches, but the overhead of performing tree rotations when inserting and deleting an object more than offset this advantage on random point sets. We suspect this to be true for balanced tree schemes also. However, Tess may be compiled to use a treap representation instead.

Only boundary ray objects are stored in the sweep table. Two integer fields within the record for a boundary ray determine the regions which come immediately before and after the ray.

4.3 Voronoi Diagram and Delaunay Triangulation

The Voronoi diagram and Delaunay Triangulation is realized with a data structure called TS. TS is the main interface to Tess (see also the `ts.h` file). Let V denote an instance of TS. Here are some selected functions on TS.

TS_Init	initialize Tess
TS_Stop	terminate Tess
$V \leftarrow$ TS_New(A, N)	V gets a Voronoi diagram and Delaunay triangulation of the N sites in array A
TS_Free(V)	dispose of V
TS_Save(V, f, o)	save V to a file f using options in o
TS_Load(V, f)	load V from a file f
TS_DrawVor(V, p, r, z)	draw V using the line segment drawing routine p with client data z in an area at least as large as the rectangle r
TS_DrawDel(V, p, z)	draw the dual (Delaunay triangulation) of V using the line segment drawing routine p with client data z
TS_Prep(V)	prepare incident edge information for V
$I \leftarrow$ TS_Hull(V)	I gets the indices of the sites on the convex hull
$i \leftarrow$ TS_Closest(V)	i gets the index of the shortest Delaunay edge in V
$I \leftarrow$ TS_Nearest(V, i)	I gets the indices of the nearest neighboring sites to the site at index i

The two drawing functions TS_DrawVor and TS_DrawDel take a procedure to be called for rendering line segments. This technique is used to separate out any graphical display concerns from Tess, making extensions to alternative output devices easy.

The TS representation is public and consists of three arrays of sites, vertices, and

edges. Three counts specify the size of each array. The site array is lexicographically sorted; the indices of sites in this array are used elsewhere in the representation. For example, a vertex in the vertex array is represented by a triple of site indices. (The vertex location is also included, though it is redundant.) This triple corresponds to a Delaunay triangle. The edge array is basically a doubly-connected edge list structure [8]. Each oriented Voronoi edge consists of the type of edge, source and destination vertex indices, left and right site indices, and counter-clockwise successor edge indices. The successor indices are links which capture incidence information about regions, edges, and vertices. A Voronoi edge is always oriented so that the left site index is larger. There are six types of Voronoi edges: a line segment, a vertically upward ray, a vertically downward ray, a ray extending to the left, a ray extending to the right, and an infinite line. (An infinite endpoint is indicated by a -1 vertex index.) Similar declarations also exist for Delaunay edges. The TS representation is set up so that the Voronoi diagram and Delaunay triangulation, which are duals of each other, share storage. There is also an array that maps a Voronoi vertex to an incident Voronoi edge and an array that maps a site to an incident Delaunay edge. These two arrays are only computed on demand (or as required) by calling TS_Prep.

Unlike incremental or divide-and-conquer algorithms, this representation (or more specifically, the doubly-connected edge list) does not participate in the Voronoi diagram computation. For Tess, it is merely an output structure; the fields of the edge list are set only once, to the final value, and never changed. The construction of this structure is documented in the source code.

5 Primitives

This section describes the efficient implementation of the main geometric primitives needed in $\mathcal{F}$. There are two main primitives. The first detects and computes candidate vertices. The second is used for locating the region where a new site resides. This primitive reduces to determining which side of a boundary ray (left or right) the new site lies.

5.1 Candidate Vertex

A candidate vertex in $*(V)$ is the intersection of two adjacent boundary rays in the sweep table. Two adjacent boundary rays do not necessarily intersect, however. Tess does not compute the intersection, if it exists, as the intersection of two hyperbolic arcs. Instead, as Fortune outlines, Tess computes the intersection as the transformed intersection of two bisectors. Consider two adjacent boundary rays C_{uv} and C_{vw}. The non-existence of this intersection is sometimes easy to detect.

Theorem 1 *If C_{uv} and C_{vw} are two consecutive minus and plus boundary rays in the sweep table, then they do not intersect or they belong to the same bisector.*

PROOF. We shall prove the contrapositive statement: if C_{uv} and C_{vw} are any two consecutive, intersecting boundary rays in the sweep table belonging to different bisectors, then the case that C_{uv} is a minus boundary ray and C_{vw} is a plus boundary ray never occurs.

The argument is by induction on the number of processed events (sites and Voronoi vertices). In the base case of zero events, the sweep table is empty and the statement is trivially true. Assume the statement to be true after processing k events. Consider what happens when processing the $k+1$st event.

When handling a site, two consecutive minus and plus boundary rays belonging to the same bisector are inserted between two other boundary rays. By inductive assumption, the statement is true for the two other boundary rays. Moreover, regardless of their directions, the new insertions cannot violate the statement.

When handling a vertex p, a new boundary ray replaces two existing ones. Suppose we cause a violation of the statement. There are two cases. First, we could have inserted a new plus boundary ray after an existing minus one. Second, we could have inserted a new minus boundary ray before an existing plus one.

In the first case, the sweep table contains: $\ldots, C^-_{uv}, C_{vt}, C_{tw}, \ldots$, and becomes: $\ldots, C^-_{uv}, C^+_{vw}, \ldots$ by replacing C_{vt} and C_{tw} (which do not belong to the same bisector) with C^+_{vw}. If C^-_{uv} and C^+_{vw} intersect, they must intersect at some point below, one, or above the current sweepline (through p). If they intersect below, then their intersection, a candidate vertex, would have been processed at some earlier sweepline position. They would be deleted from the sweep table at that time—a contradiction to their presence now. If they intersect above, then C^-_{uv} intersects the sweepline to the right of where C^+_{vw} intersects—a contradiction since C^-_{uv} is left of C^+_{vw} in the total order. Otherwise, they must intersect the current sweepline at p. We have a degeneracy because C^-_{uv}, C^+_{vw}, C_{vt}, and C_{tw} all intersect at p; the sites u, v, w, and t are cocircular. Since C^-_{uv} and C_{vt} were consecutive and intersected, C_{vt} must have been a minus boundary ray (by inductive assumption and because it cannot be vertical). Since C^-_{vt} and C_{tw} were consecutive and intersected, C_{tw} must have been a minus boundary ray (by inductive assumption and because it cannot be vertical). Since C^-_{vt} and C^-_{tw} both intersect at p, p is never to the right of the higher of v and w. Thus, C^+_{vw} should not have been created and inserted—a contradiction.

The second case is similar. Thus, the statement is true after the $k+1$st event. ■

Corollary 1 *If C_{uv} and C_{vw} are two consecutive minus and vertical boundary rays in the sweep table, then they do not intersect.*

Corollary 2 *If C_{uv} and C_{vw} are two consecutive vertical and plus boundary rays in the sweep table, then they do not intersect.*

Lemma 1 *If C_{uv} and C_{vw} are two consecutive vertical boundary rays in the sweep table, then they do not intersect.*

Furthermore, the above conditions are used within Tess and can be quickly verified without resorting to floating-point arithmetic. Otherwise, additional computation is required.

The following conditions show that the *existence* of an intersection is also sometimes easy to detect.

Lemma 2 *Suppose the sweepline is* $\ldots, C_{rq}, C_{qs}, \ldots$ *and becomes* $\ldots, C_{rq}, C^-_{pq}, C^+_{pq}, C_{qs}, \ldots$ *while processing a site* p*. If* C_{rq} *intersects* C_{qs}*, then* C_{rq} *intersects* C^-_{pq} *and* C^+_{pq} *intersects* C_{qs}*.*

Lemma 3 *If the sweepline ends with* $C_{rq}, C^-_{pq}, C^+_{pq}$*, then* C_{rq} *intersects* C^-_{pq}*.*

Lemma 4 *If the sweepline begins with* $C^-_{pq}, C^+_{pq}, C_{qs}$*, then* C^+_{pq} *intersects* C_{qs}*.*

Boundaries C_{uv} and C_{vw} intersect only if the bisectors B_{uv} and B_{vw} intersect (the converse does not hold). If these bisectors intersect, then they intersect at the center, c, of a circle through sites u, v, and w. The center is given by the following equations.

$$\begin{aligned} c_x &= (b_1(v_y - w_y) - b_2(u_y - v_y))/2a \\ c_y &= (b_2(u_x - v_x) - b_1(v_x - w_x))/2a \end{aligned}$$

where

$$\begin{aligned} a &= (u_x - v_x)(v_y - w_y) - (u_y - v_y)(v_x - w_x) \\ b_1 &= (u_x - v_x)(u_x + v_x) + (u_y - v_y)(u_y + v_y) \\ b_2 &= (v_x - w_x)(v_x + w_x) + (v_y - w_y)(v_y + w_y) \end{aligned}$$

Tess caches the required subexpressions to save time should they be needed again in future calculations. Even if the bisectors intersect, we must perform a test for whether the boundary rays themselves intersect. This test is needed because the boundary rays are each only a part of a transformed bisector. The required test is given by the following theorem.

Theorem 2 *Two consecutive boundary rays* C_{uv} *and* C_{vw} *belonging to different bisectors intersect if and only if the sites* u*,* v*, and* w *form a left turn (i.e.,* $(v_x - u_x)(w_y - u_y) - (v_y - u_y)(w_x - u_x) > 0$*, or equivalently* $a > 0$*).*

PROOF. Omitted.

The $a > 0$ test is efficient and the value of a can be used to compute the bisector intersection if the boundary rays do intersect. Theorem 2 is critical to the proper construction of the doubly-connected edge list output representation.

Corollary 3 *Let* C_{tu}*,* C_{uv}*, and* C_{vw} *be three consecutive boundary rays. If* C_{tu} *and* C_{uv} *intersect and* C_{uv} *and* C_{vw} *intersect, then* C_{tu} *and* C_{uw} *(from* C_{uv} *and* C_{vw}*) intersect and* C_{tv} *(from* C_{tu} *and* C_{uv}*) and* C_{vw} *intersect.*

This corollary suggests that candidate vertices (intersections) which are deemed non-Voronoi may simply be replaced by other appropriate candidate vertices, instead of being deleted from the event queue.

Once we know that the boundary rays intersect, $*(c)$ is computed by adding to c_y the distance to any of the three sites u, v, or w. This requires computing a square root.

It may seem that this candidate vertex detection and intersection calculation for two particular boundary rays is repeated as insertions and deletions of other boundary

rays are made in the sweep table, but this cannot happen. Once two adjacent boundary rays become non-adjacent, they remain non-adjacent until the algorithm finishes or until one of the rays is deleted.

Where three consecutive boundary rays intersect in two candidate vertices, it is only necessary to keep the smaller vertex (the one located lower). The other vertex should be deleted or replaced from the event queue since it cannot be a Voronoi vertex. Tess uses this optimization to reduce the size of the event queue.

5.2 Site-Boundary Test

When processing a new site p, Fortune's algorithm performs a search in the sweep table for the region within which p lies. The search is done by a technique reminiscent of binary search or bisection on the sweep table in T. The technique repeatedly rejects regions within which p cannot reside. Tess actually searches for the boundary rays which immediately precede and follow the appropriate region (in a sweep table which contains only boundary rays).

The x-coordinates of the intersections of every boundary ray in T, in order, form a non-decreasing sequence $x_1, x_2, \ldots, x_k$. Determining the region where p resides is a question of where p_x lies in this sequence. Suppose $x_i \leq p_x \leq x_j$ where x_i and x_j are adjacent in the sequence (i.e., $j - i = 1$). The site p lies in the region between the two adjacent boundary rays corresponding to x_i and x_j. Note that the sequence generally changes as the sweepline advances. Moreover, a binary search avoids the need to compute this sequence completely (or equivalently, the need to consider every boundary ray). The required decision function for this search is described below.

For the rest of this subsection, let us consider a boundary ray C_{uv} in T which intersects the sweepline (currently through p) at a point z. If $p_x < z_x$, reject every boundary ray to the right of C_{uv} in T (i.e., go left in the sequence). Similarly, if $p_x > z_x$, reject every boundary ray to the left (i.e., go right in the sequence). If $p_x = z_x$, it does not matter which portion we reject as long as we are consistent during execution. Hereafter, let us always reject every ray to the left (i.e., go right).

The test primitive is efficiently implemented for vertical boundary rays: simply compute $z_x = (u_x + v_x)/2$ and make the obvious comparison. This z_x value can also be cached since it is independent of the sweepline position for vertical boundary rays. Thus, let us now only consider non-vertical boundary rays. Without loss of generality, assume that $u_y > v_y$. (This somewhat simplifies the implementation.) Tess uses the following detailed specification for the test; this is the most efficient of four alternatives considered in [12].

Cache:

$$
\begin{aligned}
s_1 &= u_x - v_x \\
s_2 &= u_y - v_y \\
s_3 &= s_1(u_x + v_x) + s_2(u_y + v_y) \\
s_4 &= s_1 + s_1 \\
s_5 &= s_2 + s_2 \\
s_6 &= s_5\, u_y
\end{aligned}
$$

Compute:

$$
\begin{aligned}
s_7 &= s_5 p_y \\
s_8 &= s_4 p_x - s_3 \\
s_9 &= s_7 + s_8 \\
s_{10} &= u_x - p_x
\end{aligned}
$$

For C_{uv}^-, if $s_9 < 0$ or $(p_y - u_y)(s_6 + s_8 + s_9) < s_5 s_{10}^2$, then go left, otherwise go right.
For C_{uv}^+, if $s_9 > 0$ or $(p_y - u_y)(s_6 + s_8 + s_9) > s_5 s_{10}^2$, then go left, otherwise go right.

5.3 History Test

The number of these site-boundary tests is effectively reduced by the following improvement which uses past decisions. We call it the *history test*. Define the following.

$$
\begin{aligned}
P(C_{uv}) &= \{q \mid q_y \le p_y \text{ and the location of } q \text{ relative to } C_{uv} \text{ is known}\} \\
\alpha(C_{uv}^+) &= \max\{q_x \mid q \in P(C_{uv}^+) \text{ and } q \text{ is left of or on } C_{uv}^+\} \\
\beta(C_{uv}^-) &= \min\{q_x \mid q \in P(C_{uv}^-) \text{ and } q \text{ is right of or on } C_{uv}^-\}
\end{aligned}
$$

If C_{uv} is a plus boundary and $p_x < \alpha(C_{uv}^+)$, then $p_x < z_x$ is true (otherwise, we cannot tell without further computation). This stems from the monotonicity property of boundary rays and that the sweepline never retreats. Similarly, if C_{uv} is a minus boundary ray and $p_x \ge \beta(C_{uv}^-)$, then $p_x \ge z_x$ is true. When a boundary ray C_{uv} is created, Tess initializes $P(C_{uv})$ to contain its base and sets α or β accordingly.

6 Performance

We have spent a fair amount of effort tuning the code and paying attention to "hotspots" that slow down the implementation. No routine dominates the running time. Storage management is improved by preallocating or reusing data structures and maintaining free lists.

Besides being efficient, we expect the implementation of a geometric algorithm to be correct in handling all cases (including degenerate ones), to be stable in spite of numerical errors that occur in floating-point arithmetic, and to be dependably accurate in the final result.

For efficiency and accuracy, Tess uses double-precision floating-point arithmetic; the input data is given in single-precision. Cached values are computed and stored in double-precision. Round-off errors in the cached values for a boundary ray are not propagated into the cached values of another boundary ray. Moreover,

Table 1: Performance Comparison (timings in seconds)

N	Uniform		Normal		Exponential	
	Tess	Fortune	Tess	Fortune	Tess	Fortune
2000	0.73	0.95	0.72	0.97	0.75	0.97
4000	1.50	1.97	1.55	1.95	1.55	2.03
6000	2.27	2.97	2.30	3.10	2.32	3.12
8000	3.00	4.05	2.98	4.10	3.10	4.23
10000	3.80	5.07	3.80	5.17	3.82	5.37
12000	4.73	6.33	4.73	6.28	4.80	6.50
14000	5.43	7.17	5.53	7.32	5.50	7.70
16000	6.30	8.35	6.20	8.50	6.43	8.83

redundancy is avoided as far as we can determine (one example is the history test). Non-redundancy involves not recomputing a test if the decision can be deduced from decisions already made; if this can be ensured, an algorithm cannot introduce inconsistencies in the relationship of its geometric entities [7]. Striving to remove redundancy helped make Tess more efficient as a side benefit. (However, we do not know yet if our particular site-boundary test primitive is numerically stable, especially when the two sites defining the boundary are nearly cohorizontal.) Tess passes every case in our hand-verified test suite of 95 files, which includes many degenerate cases. Also, Tess may be compiled to make extensive assertion checks during the computation.

Table 1 indicates the performance of Tess compared to Fortune's implementation on uniform, normal, and exponential distributions. The computer used is a Sun SPARCstation IPC with 12 megabytes of main memory.

7 Conclusion

This paper described an efficient implementation of Fortune's algorithm for computing the planar Voronoi diagram. Details of the algorithm's data structures and geometric primitives were discussed. Future work involves proving the stability of the site-boundary test and proving the correct handling of degeneracies in floating-point arithmetic.

References

[1] Cecilia R. Aragon and Raimund G. Seidel. Randomized search trees. In FOCS89 [3], pages 540–545.

[2] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[3] *Symposium on Foundations of Computer Science Proceedings*, volume 30, 1989.

[4] Steven Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(2):153–174, 1987.

[5] K. Hinrichs, J. Nievergelt, and P. Schorn. Plane-sweep solves the closest pair problem elegantly. *Information Processing Letters*, 26:255–261, January 1988.

[6] K. Hinrichs, J. Nievergelt, and P. Schorn. A sweep algorithm for the all-nearest-neighbors problem. In H. Noltemeier, editor, *International Workshop on Computational Geometry Proceedings*, pages 43–54, Würzburg, Germany, March 1988. Springer-Verlag.

[7] Victor Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In FOCS89 [3], pages 500–505.

[8] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[9] Gerhard Reinelt. Fast heuristics for large geometric traveling salesman problems. *ORSA Journal on Computing*, 4(2):206–217, 1992.

[10] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, Massachusetts, 2 edition, 1988.

[11] Gary M. Shute, Linda L. Deneen, and Clark D. Thomborson. An $o(n \log n)$ plane-sweep algorithm for l_1 and l_∞ Delaunay triangulations. *Algorithmica*, 6:207–221, 1991.

[12] Kenny Wong. *Techniques for Optimizing Fortune's Plane-Seep Algorithm for Voronoi Diagrams*. PhD thesis, University of Victoria, Department of Computer Science, April 1991.

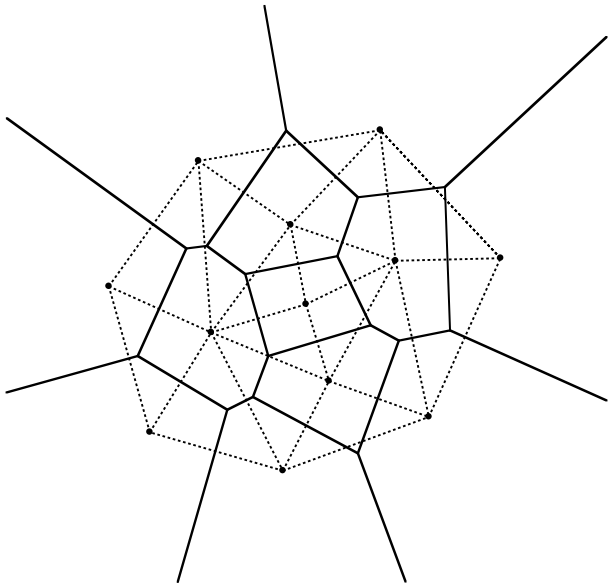

Figure 1: Voronoi Diagram and Delaunay Triangulation

let $p_1, p_2, \ldots, p_m$ be the sites with minimal y-coordinate, ordered by x-coordinate
$Q \leftarrow S - \{p_1, p_2, \ldots, p_m\}$
create initial vertical boundary rays $C^0_{p_1,p_2}, C^0_{p_2,p_3}, \ldots, C^0_{p_{m-1},p_m}$
$T \leftarrow *(R_{p_1}), C^0_{p_1,p_2}, *(R_{p_2}), C^0_{p_2,p_3}, \ldots, *(R_{p_{m-1}}), C^0_{p_{m-1},p_m}, *(R_{p_m})$
while not IsEmpty(Q) **do**
    $p \leftarrow$ DeleteMin(Q)
    case p of
    p is a site in $*(V)$:
        find the occurrence of a region $*(R_q)$ in T containing p,
            bracketed by C_{rq} on the left and C_{qs} on the right
        create new boundary rays C^-_{pq} and C^+_{pq} with bases p
        replace $*(R_q)$ with $*(R_q), C^-_{pq}, *(R_p), C^+_{pq}, *(R_q)$ in T
        delete from Q any intersection between C_{rq} and C_{qs}
        insert into Q any intersection between C_{rq} and C^-_{pq}
        insert into Q any intersection between C^+_{pq} and C_{qs}
    p is a Voronoi vertex in $*(V)$:
        let p be the intersection of C_{qr} on the left and C_{rs} on the right
        let C_{uq} be the left neighbor of C_{qr} and
            let C_{sv} be the right neighbor of C_{rs} in T
        create a new boundary ray C^0_{qs} if $q_y = s_y$,
            or create C^+_{qs} if p is right of the higher of q and s,
            otherwise create C^-_{qs}
        replace $C_{qr}, *(R_r), C_{rs}$ with newly created C_{qs} in T
        delete from Q any intersection between C_{uq} and C_{qr}
        delete from Q any intersection between C_{rs} and C_{sv}
        insert into Q any intersection between C_{uq} and C_{qs}
        insert into Q any intersection between C_{qs} and C_{sv}
        record p as the summit of C_{qr} and C_{rs} and the base of C_{qs}
        output the boundary segments C_{qr} and C_{rs}
    endcase
endwhile
output the remaining boundary rays in T

Figure 2: Fortune's Algorithm