Computing Planar Voronoi Diagrams in Double Precision: A Further Example of Degree-driven Algorithm Design

David L. Millman
University of North Carolina
Chapel Hill, North Carolina, USA
dave@cs.unc.edu

Jack Snoeyink
University of North Carolina
Chapel Hill, North Carolina, USA
snoeyink@cs.unc.edu

ABSTRACT

Geometric algorithms use numerical computations to perform geometric tests, so correct algorithms may produce erroneous results if insufficient arithmetic precision is available. Liotta, Preparata, and Tamassia, in 1999, suggested that algorithm design, which traditionally considers running time and memory space, could also consider precision as a resource. They demonstrated that the Voronoi diagram of n sites on a $U \times U$ grid could be rounded to answer nearest neighbor queries on the same grid using only double precision. They still had to compute the Voronoi diagram before rounding, which requires the quadruple-precision InCircle test. We develop a "degree-2 Voronoi diagram" that can be computed using only double precision by a randomized incremental construction in $O(n \log n \log U)$ expected time and $O(n)$ expected space. Our diagram also answers nearest neighbor queries, even though it doesn't even use sufficient precision to determine a Delaunay triangulation.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations*

General Terms

Algorithms, Theory

Keywords

Robust computation, Low-degree primitives, Reduced precision, Voronoi diagrams, Post-office problems

1. INTRODUCTION

Geometric algorithms usually evaluate geometric primitives (e.g., predicates like point/line comparison, left/right turn tests, or constructions like computing the line through two points or the intersection of two lines) by doing numerical computation on coordinate values. Building geometric algorithms on a tower of abstractions—including Euclidean geometry, Cartesian coordinates, and exact arithmetic on RealRAM—has greatly reduced the errors in published papers on geometric algorithms in the last 20 years. Researchers have developed many sophisticated ways to ensure that each layer of abstraction is implemented correctly and efficiently, including the exact geometric computing paradigm [5, 23], arithmetic filters [4, 11], and others [20, 21, 24]. Unfortunately, implementers often rely on the floating point arithmetic supplied by current programming environments. The numerical errors this introduces can undermine the entire tower.

Liotta, Preparata, and Tamassia [16] suggested another approach, which is to consider arithmetic precision to be a resource that should be optimized by the design and analysis of algorithms, much like the traditional resources of running time and memory space. Liotta *et al.* [16] also suggested a measure: the *arithmetic degree* of the predicates used. Most geometric predicates evaluate the sign of a polynomial whose variables are input values. Suppose that input variables can be scaled to b bit integers. Then a degree k term can be evaluated in bk bits, and a polynomial can sum a such terms in $bk + \log_2 a$ bits. Thus, the degree k can be considered the leading term that determines the precision required; we ignore the carry bits, just as we ignore constants in the asymptotic measures used for time and space. (In fact, since most evaluations need only the sign of the polynomial, the $\log_2 a$ can be avoided by Kahan summation [14].) This measure has been used to develop algorithms for intersecting line segments [1, 2, 17].

Liotta *et al.* [16] developed an *implicit Voronoi diagram* to answer Post Office queries for n sites on a $U \times U$ grid – finding the closest site to a query in $O(\log n)$ time and $O(n)$ space, and using predicates whose polynomials have maximum degree 2. (We call degree 2 predicates "double precision.") Unfortunately, their algorithm must first compute the entire Voronoi diagram before rounding it to their structure; Voronoi computation requires the InCircle test, which is an irreducible polynomial of degree 4. (They also sort Voronoi vertices by y coordinate, making their algorithm degree 5.) In subsection 2.1 we look at the degrees of several predicates and algorithms; the degree of an algorithm is the highest degree of its predicates.

We develop a new implicit Voronoi diagram structure that we are able to compute by randomized incremental construc-

SCG'10, June 13–16, 2010, Snowbird, Utah, USA.

tion in $O(n \log n \log U)$ expected time and $O(n)$ expected space, again using only double precision.

To appreciate the challenge in doing so, note that most algorithms compute Voronoi diagrams by computing the Delaunay triangulation, which uses the degree 4 InCircle test to verify that three points define an empty circle [12]. Our algorithms will compute the closest sites only at grid points, which is not enough information to obtain the dual Delaunay triangulation.

Fortune's sweep algorithm [10], computes the Voronoi directly, but uses a degree 6 predicate that orders two circumcircles by their extreme points. The notion of abstract Voronoi diagrams [15] reduces Voronoi computation to ordering bisectors around Voronoi vertices, which again uses InCircle. Sugihara and Iri's incremental construction [22], which tries to avoid geometric tests in favor of topological inference, still relies on InCircle. In their algorithm, the basic step is to carve out the Voronoi cell of a newly inserted site by walking through the current diagram. This doesn't seem to work without a notion of adjacency of cells, which seems difficult to achieve with predicates of degree two.

In earlier work, we were able to implement an $O(n \log(n+U))$-time randomized incremental construction to compute a reduced precision Voronoi diagram [19] using a degree 3 predicate for ordering bisectors along a line. In this paper, we define a trapezoidation of approximate Voronoi cells that is suitable for randomized incremental construction and use a history DAG (directed acyclic graph) rather than a walk to find which trapezoids need to be updated when a new site is inserted.

The nearest-neighbor transform, known as the discrete or digital Voronoi and related to the distance transform, computes nearest sites for for every point in the grid, spending at least $\Omega(U^2)$ time. Breu *et al.* [3] showed that $O(U^2)$ was achievable by computing a Voronoi diagram on the grid by divide and conquer. Chan [7] and Maurer [18] independently developed simpler algorithms that generalized to higher dimensions, and Cao *et al.* [6] gave a fast implementation on GPU processors. These algorithms were described as degree 5, although Chan observed that his was actually degree 3, and could be improved to degree 2 with a little more work; we previously defined the proxy trapezoidation [8] to more compactly store the output of this degree 2 algorithm. This paper extends the definition and shows how to construct the structure directly in linear expected space.

2. DEFINITIONS AND NOTATION

In this section, we first establish some notation for familiar concepts, such as Voronoi diagrams and trapezoid graphs, then define some of the key concepts for our structure.

2.1 The Voronoi Diagram and Grids

Given a set of n sites $S = \{s_1, \ldots s_n\}$ in the plane, the *Voronoi diagram* of S partitions the plane into maximally connected regions with the same set of nearest neighbor sites. Regions with one closest site are called *Voronoi cells*, regions with two closest sites are called *Voronoi edges*, and regions with three or more closest sites are called *Voronoi vertices*. Building a point location structure on top of the Voronoi diagram gives the classical solution to the *Post office problem*; after preprocessing the sites, you can determine the closest site to a query point, q, in $O(\log n)$ time.

We restrict our sites and query points to a $U \times U$ integer grid, denoted $\mathbb{U}$. The *discrete Voronoi diagram* labels only the grid points with their closest sites. Note that a line bisecting two grid points might not hit any points of the grid $\mathbb{U}$. In building our structures, it will be convenient to use the half-integer grid $\mathbb{U}_2 = \frac{1}{2}[2U+1]^2$ and the square $\mathcal{U} = [1/2, U+1/2]^2 \subset \mathbb{R}^2$. That way, at least the midpoint between two sites $(s_i + s_j)/2$ is on the half-integer grid. We sometimes say that a point is *single precision* to mean that it is on the grid $\mathbb{U}_2$.

Consider the degree of one basic construction and four predicates for Voronoi diagrams. We use a model of computation that includes constant-time arithmetic operations, and consider computations both with and without floor or integer division functions. For these basic predicates, floor makes no difference.

`VorVertex`(a, b, c)**:** A Voronoi vertex constructed from grid point sites $a, b, c \in \mathbb{U}$ has coordinates that can be expressed as rational numbers of degree 3 over degree 2.

`Point Comparison` $a \prec b$**:** Compare points by lexicographic order, so that $a \prec b$ if and only if $a_x < b_x$ or ($a_x = b_x$ and $a_y < b_y$). Comparing grid points $a, b \in \mathbb{U}_2$ is degree 1, comparing a grid point with a Voronoi vertex is degree 3, and comparing two Voronoi vertices is degree 5.

q.`Nearer`(s_i, s_j) **or** b_{ij}.`side`(q)**:** A *bisector* $b_{ij} = \{p \in \mathbb{R}^2 : \|p - s_i\| = \|p - s_j\|\}$ is the line equidistant to two sites. Determining if a point q is on a bisector, or determining the closer of the two sites, is degree two—simply compare squared distances.

`InCircle`(a, b, c, q)**:** The basic predicate for Voronoi computation [9], InCircle determines whether q is inside the circumcircle of $a, b, c \in \mathbb{U}$. This could be done by comparing the squared distances from VorVertex(a, b, c) to a and to q, which gives a degree 6 polynomial when you clear fractions, but that polynomial can factor to give the usual InCircle determinant, which is degree 4.

`Orient`(a, b, c)**:** The orientation test, which reports if the path a, b, c makes a right turn, left turn, or goes straight, is a determinant on the homogeneous coordinates of the inputs. It is a degree 2 test on grid points, which is the only way we will use it. Degree goes up by 2 for each Voronoi vertex: it takes degree 8 to determine orientation for three arbitrary Voronoi vertices, and degree-6 to determine the orientation of a grid point relative to an edge of a triangulation of a Voronoi diagram. (Thus, one should avoid point location structures that first triangulate.)

2.2 Trapezoidation and Point Location

Any planar subdivision with line segments can be decomposed into trapezoids by making vertical cuts at the endpoints of each segment that extend until reaching another segment. We define three different, but related, trapezoidations in the next three subsections. Each trapezoid τ will have *top* and *bottom* lines, and a *left* and *right* point, with $\tau.left \prec \tau.right$ and both points on or between the *top* and *bottom* lines. Each trapezoid may have up to four neighbors, reached by crossing the vertical lines above or below $\tau.left$ and $\tau.right$, although if the subdivision is into monotone regions, as in Figure 1, then each trapezoid has at most two neighbors. When $\tau.left$ and $\tau.right$ have the same x coordinate, we can still think of τ as a trapezoid since the lexicographic order on points used by $\prec$ is consistent with

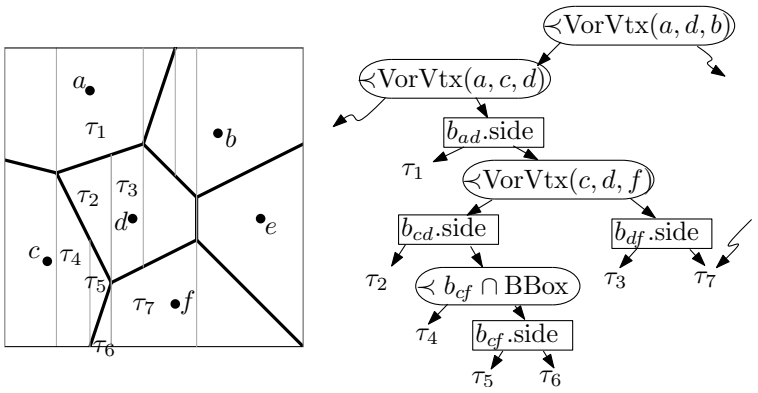

Figure 1: Trapezoidation of a small Voronoi diagram, and a piece of the trapezoid graph for the strip with labeled faces, $\tau_1, \ldots, \tau_7$.

skewing the points slightly, or rotating the vertical direction counter-clockwise. For example, the bisector of d and e in Figure 1 is the bottom edge for an infinitesimally thin trapezoid in the Voronoi cell of d and the top edge for an infinitesimally thin trapezoid in the Voronoi cell of e.

Trapezoid graphs [9, ch. 6] support point location queries, which identify the trapezoid containing a query point q. The *trapezoid graph* is a directed acyclic graph (DAG) with three types of nodes:

x-node: Stores a point v; compares a query $q \prec v$. If both q and v are grid points on $\mathbb{U}_2$, this test is degree 1; if v is a Voronoi vertex, this test is degree 3.

y-node: Given a query point q and a line ℓ, determine if q is above, below, or on ℓ. If ℓ is a bisector of two sites, or if ℓ is defined by two grid points on $\mathbb{U}_2$, then this test is degree 2. On the other hand, if ℓ is defined by two arbitrary Voronoi vertices, this test is degree 6.

leaf node: Leaf nodes represent trapezoids, for which the containing region is known.

Figure 1 shows a portion of the trapezoid graph for seven of the trapezoids in a small Voronoi diagram.

Liotta, Preparata, and Tamassia [16] achieved a degree-two post-office query by building a trapezoid graph for the Voronoi diagram. The straightforward way, depicted in Figure 1, would be to use Voronoi vertices (and intersections of bisectors with the bounding box) at x-nodes and bisectors at y nodes. Liotta *et al.* improve this by a simple observation: since the queries are grid points, rounding non-integer x nodes to half-integers reduces the degree of that test from 3 to 1 without changing any test results. They still used a degree 5 algorithm to compute the Voronoi diagram. Our previous work [19] showed how to build their structure directly with degree three in $O(n \log(n + U))$ expected time.

2.3 Voronoi Polygons on the Grid and Proxy Trapezoidation

Define the *Voronoi polygon* $C_S(s_i)$ to be the convex hull of the grid points of $\mathbb{U}_2$ in the closure of the Voronoi cell of site $s_i \in S$. We omit the subscript S when the set of sites is understood. Voronoi polygon $C(s_i)$ is contained in the Voronoi cell of s_i, and the containment may be proper. As illustrated in Figure 2, the set of all Voronoi polygons may leave *gaps* in $\mathcal{U}$, even though they cover all the grid points.

Define the *cell proxy* of s_i, denoted $P_S(s_i)$, to be the line segment from the minimum to the maximum point of $C_S(s_i)$, as illustrated in the middle of Figure 2. Because the n Voronoi polygons can actually have a total of $\Theta(n \log U)$ sides, we will often prefer to work with the n proxy segments instead. Define the *proxy trapezoidation* to be the vertical visibility map of all cell proxies, as shown in the same figure. Note that the proxy trapezoidation for a set of n sites S has at most $3n+1$ trapezoids, and is canonical—it is completely determined by the sites.

In [8], we observed that if we can compute a proxy trapezoidation, then we can use it to answer post office queries using degree 2, because only the sites contributing the top and bottom proxies are candidates for the closest sites to the grid points inside a proxy trapezoid. (Note that the only grid point actually on a vertical side is the endpoint of a proxy segment that defines the side, and we already know the neighbor of a proxy – the lexicographic ordering used by $\prec$ implicitly perturbs grid points directly above a defining endpoint into the trapezoid to the right, and directly below a defining endpoint into the trapezoid on the left.)

LEMMA 1. *If $q \in \mathbb{U}$ is a query point in a trapezoid τ with $\tau.top = P(s_i)$ and $\tau.bottom = P(s_j)$ then q is in $C(s_i)$ or $C(s_j)$.*

PROOF. Suppose that some grid point $q \in \tau$ belongs in a different Voronoi polygon $C(s_k)$. Since the Voronoi polygons are convex and non-overlapping, the cell proxy $P(s_k)$ must pass through trapezoid τ, which is a contradiction. □

COROLLARY 2. *Given cell proxies $P_S(s_i)$ for all $s_i \in S$, one can build a point location structure that answers post office queries in $O(\log n)$ expected time and linear expected space.*

PROOF. We simply use randomized incremental construction of the trapezoid graph of the proxy trapezoidation, as in [9, ch. 6]. A query determines the trapezoid τ that contains the query point q using degree 1 tests at x-nodes and degree 2 orientation tests at y-nodes. We can then use the degree 2 `q.Nearer()` test to decide which of the two candidate sites is closer. □

2.4 Voronoi Trapezoidation and Conflicts

A few additional concepts will help us describe a randomized incremental construction of the proxy trapezoidation.

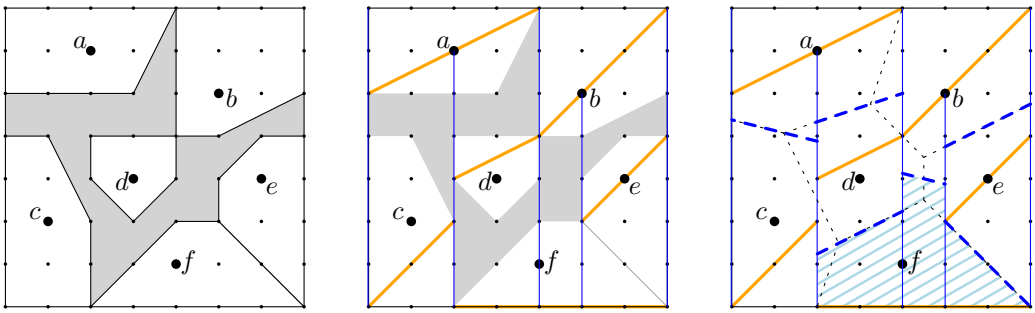

Figure 2: *Left:* **Voronoi polygons of sites on a grid may leave gaps.** *Middle:* **Proxy trapezoidation of proxy segments, which connect the maximal and minimal points in a Voronoi polygon. The lexicographic order for points implies that grid points that appear on the vertical segments are actually in trapezoids to the left or right, unless they are proxy segment endpoints. Thus, f is in the trapezoid that contains d, a is in the trapezoid above the proxy for d, and there is an infinitesimally thin trapezoid between the endpoints of the proxies for c and d.** *Right:* **Splitting trapezoids of the proxy trapezoidation by bisectors gives the Voronoi trapezoidation.**

First, if we split each trapezoid of the proxy trapezoidation with the bisector of sites donating the proxies at top and bottom, then we obtain the Voronoi trapezoidation depicted at the right of Figure 2. This trapezoidation is also canonical, since it is derived from the proxy trapezoidation.

It is important to note that we don't actually compute y-coordinates of bisectors; the *top* and *bottom* pointers just point to line equations for proxies, bounding box sides, or bisectors, each of which can be represented as a degree 2 polynomial. All *left* and *right* points are proxy endpoints or bounding box corners, and all are from $\mathbb{U}_2$. Thus, we observe:

LEMMA 3. *One can test if a grid point q is inside a given trapezoid τ of a Voronoi trapezoidation in constant time using degree 2.*

In the *Voronoi trapezoidation* every trapezoid τ intersects some proxy $P(s_i)$ and either a bisector b_{ij} or a bounding box side; we set $\texttt{site}(\tau) = s_i$ and $\texttt{bisector}(\tau) = b_{ij}$ (or the bounding box side in the latter case).

The trapezoids with $\texttt{site}(\tau) = s_i$ cover the Voronoi polygon $C(s_i)$ and therefore contain all grid points in the Voronoi cell of s_i. We can't really say how the covering trapezoids relate to the true Voronoi cell of s_i; clearly they may miss portions of the cell on the left and right ends, because they all use the proxy $P(s_i)$ as top or bottom. As the striped trapezoids in Figure 2 show, they don't necessarily form a convex region and may contain more or less than the true Voronoi cell – all we guarantee is that they contain the grid points of $C(s_i)$. This means that it would be difficult to bound the work necessary to insert a new site s by walking through the Voronoi trapezoidation of existing sites.

Consider a subset of sites $R \subseteq S$. We say a site $s_k \notin R$ is *in conflict* with trapezoid τ of the Voronoi trapezoidation of R if and only if there exists a $g \in \tau \cap \mathbb{U}_2$ such that $\|g - s_k\| < \|g - \texttt{site}(\tau)\|$, i.e. if some grid point in τ is closer to s_k than to $\texttt{site}(\tau)$. We extend the conflict to the proxy trapezoid whose split produced τ, as well. We call g a *witness* to the conflict.

In the next section we will give a detailed description of how to maintain the proxy/Voronoi trapezoidations as sites are inserted. As in many randomized incremental constructions, we will find it useful to keep the history DAG of the proxy trapezoidation: as a new site is added, some old trapezoids will be deleted and their area filled with new trapezoids – the parents of a new trapezoid will be the minimal set of old trapezoids that cover all the grid points that it contains. This implies that we can trace conflicts through history:

LEMMA 4. *If site $s \in S$ is in conflict with a trapezoid τ of the history DAG, then it is in conflict with at least one parent of τ.*

We will actually be able to use a trapezoid graph to store the history of updates, so parents and children will not be explicit in the data structure, but they will serve a role in the analysis. For sites S, let D2-Voronoi(S) denote the proxy trapezoidation with its history represented in a trapezoid graph, along with the Voronoi trapezoidation implied by introducing bisectors in each trapezoid.

3. CONSTRUCTION

In this section we give the details of our degree 2 randomized incremental construction (RIC). We assume that the sites have been shuffled randomly; let $S_i = \{s_1, s_2, \ldots, s_i\}$. We further assume that we have built the proxy trapezoidation and its history that constitute D2-Voronoi(S_{i-1}) and want to obtain D2-Voronoi(S_i) by inserting s_i.

After observing how incremental construction affects the Voronoi polygons and proxy segments, as defined in Section 2.3, we introduce a few more predicates and constructions for grid points in trapezoids. Finally, we describe the updates to the trapezoidation, and analyze the expected time and space.

LEMMA 5. *In an incremental construction, a Voronoi polygon on the grid can only shrink:*

$$\forall s \in S_{i-1}, \quad C_{S_i}(s) \subseteq C_{S_{i-1}}(s),$$

and a proxy segment changes if and only if the new site s_i is in conflict with at least one of its endpoints:

$$\forall s \in S_{i-1}, \qquad P_{S_i}(s) \neq P_{S_{i-1}}(s) \iff$$
$$s_i \text{ conflicts with an endpoint of } P_{S_{i-1}}(s).$$

3.1 Predicates and Constructions

We will continue to use a model of computation in which arithmetic operations are constant-time. Computations do not use floor or integer division unless otherwise stated.

LEMMA 6. *(i) Determining if bisector b_{ij} intersects the vertical segment $\overline{rt}$ with $r, t \in \mathbb{U}_2$ takes degree 2 and constant time. (ii) Determining the shortest subsegment $\overline{ab} \subset \overline{rt}$ with $a, b \in \mathbb{U}_2$ that intersects b_{ij} takes degree 2 and $\log \|r - t\|$ time, or constant time if floor is allowed.*

PROOF. For (i), compare the result of b_{ij}.`side`() on the two endpoints of $\overline{rt}$. (ii) can use floor, or can binary search with b_{ij}.`side`(). □

LEMMA 7. *There is a degree 2 procedure* `HullVertices`(), *taking $O(\log U)$ time, that computes the convex hull of the grid points, if any, that lie in a region bounded by a constant number of bisectors or lines defined by two grid points.*

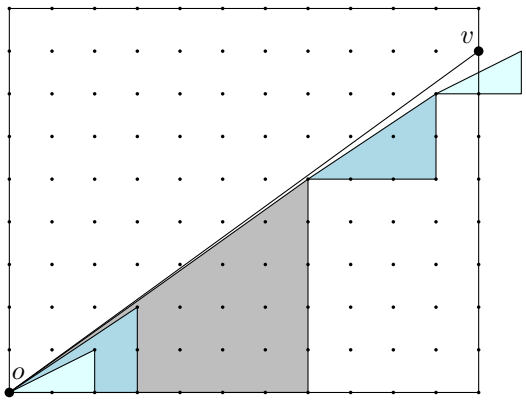

Figure 3: Each darker triangle's hypotenuse at left is an increasingly accurate rational representation of the slope of segment $\overline{ov}$; the `GCD walk` reuses these at right to compute the convex hull of the grid points below $\overline{ov}$.

PROOF. Euclid's GCD algorithm, when interpreted geometrically, gives the sequence of best rational approximations to an arbitrary slope, ordered by increasing denominator. Kahan and Snoeyink [13][Lemma 4.6] turned this into a procedure `HullVertices`(o, v) that takes the origin, an arbitrary point v, and computes the convex hull of the grid points in the bounding box of $\overline{ov}$ that are on or below $\overline{ov}$. Their procedure runs in $\Theta(\log(\|o-v\|)$ time and is degree 2; the highest degree predicate is the orientation test on grid points. The lower bound applies because the convex hull of grid points in the intersection of $O(1)$ halfplanes may have this many vertices.

With a small modification, their procedure can start from a unit segment between grid points, from Lemma 6(ii), and follow a bisector within a region. We call this operation the `GCD walk`. In each step we evaluate the degree 2 predicates, bisector `side`, and `orientation` for grid points. The walk takes time proportional to the log of the maximum side of the bounding box of the region in which it runs. This remains bounded by $O(\log U)$. □

This has two useful corollaries.

COROLLARY 8. *Predicate s_i.`inConflict`(τ) determines if s_i is in conflict with trapezoid τ of the Voronoi trapezoidation in $O(\log U)$ time and degree two.*

PROOF. Suppose that $s_u =$ `site`(τ), as in Figure 4. We can use the `GCD walk` to determine the convex hull of grid points that are in τ between $\tau.left$ and $\tau.right$ and bounded by `bisector`(τ) and b_{ui}, taking $O(\log U)$ time and degree 2. □

For the next corollary, and as shown with a dash-dotted line in Figure 4, the proxy of any convex hull of grid points is the line segment from the minimum to the maximum under lexicographic order ($\prec$).

COROLLARY 9. *Given a site s_i in conflict with a trapezoid τ of the Voronoi trapezoidation,* `findProxies`(s_i, τ) *constructs, in $O(\log U)$ time and degree 2, the proxies for two convex hulls of the grid points in trapezoid τ of the Voronoi trapezoidation—those that are closer to* `site`(τ), *and those that are closer to s_i.*

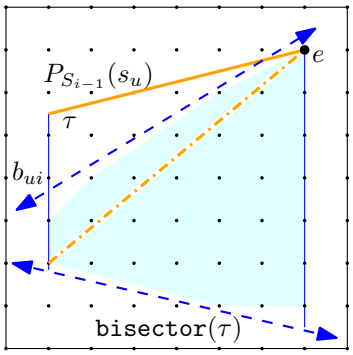

Figure 4: Suppose that new site s_i is in conflict with a Voronoi trapezoid τ, with $s_u =$ `site`(τ). As drawn, $\tau.top$ is proxy $P_{S_{i-1}}(s_u)$ and $\tau.bottom =$ `bisector`(τ). The new bisector b_{ui} cuts the shaded area from τ, which is the convex hull of the grid points that witness the conflict by being closer to s_i than s_u. The dash-dotted line depicts the proxy of s_i in τ.

3.2 Incremental construction

Now, suppose that we have constructed D2-Voronoi(S_{i-1}). We update the proxy trapezoidation and its history as we insert the new site s_i using three step. The first step, *Find Conflict*, identifies the set of trapezoids of D2-Voronoi(S_{i-1}) that are in conflict with s_i. The second step, *Proxy Update*, finds the new proxy $P_{S_i}(s_i)$ and updates all other proxies that changed on the insertion of s_i. The third step, *Trapezoidation Update* splits and merges proxy trapezoids to reflect these proxy changes, and updates the trapezoid graph to capture the history, producing D2-Voronoi(S_i).

Find Conflict:.

We collect the trapezoids of the D2-Voronoi(S_{i-1}) that are in conflict with s_i by traversing the history DAG. Starting at the root, we visit only those trapezoids that are in conflict with s_i, or whose parent was in conflict with s_i. At each trapezoid, we test for conflict by Corollary 8, spending $O(\log U)$ time and degree two. By Lemma 4, if there is a conflict, it persists all the way to the root, so we will find it.

Proxy Update:.

Adding s_i to S_{i-1} creates a new proxy $P_{S_i}(s_i)$ for s_i, and usually forces some old proxies to be updated, for example, proxy $P_{S_{i-1}}(s_u)$ in Figure 4. Thanks to Lemma 5, we can identify which proxies must be updated by testing their endpoints: In Figure 4, trapezoid τ is defined in part by endpoint e of proxy $P_{S_{i-1}}(s_u)$. Testing $e.\texttt{Nearer}(s_i, s_u)$ identifies, in constant time and degree 2, whether this proxy must be updated. Moreover, we can find all proxies to update by testing only those that define trapezoids in conflict with s_i, since an endpoint that witnesses the need to update the proxy is also a gridpoint that witnesses a conflict with s_i.

For each Voronoi trapezoid τ in conflict with s_i, we can apply Corollary 9 to find the min and max grid points in the regions that we obtain if we split τ by the bisector of $\texttt{site}(\tau)$ and s_i. The min and max grid points identified from regions on the s_i side of the bisector become the endpoints of the new proxy $P_{S_i}(s_i)$. Similarly, for any proxy that is updated, we take the min and/or max points identified from the regions closer to the corresponding site. Thus, we spend $O(\log U)$ time and degree 2 on each trapezoid in conflict with s_i.

Trapezoidation Update:.

Having identified the updates to proxies, we must now update the trapezoidation and the trapezoid graph. We do this in four substeps.

First, we add all new proxy endpoints, splitting the trapezoids that contain them. In the trapezoid graph, the corresponding leaves (trapezoids to split) become x-nodes that point to two new leaf nodes (the results of the splits).

Second, we add the segment for the new proxy $P_{S_i}(s_i)$ by walking through the trapezoidation, splitting and merging trapezoids as needed. This operation is the same as in any trapezoidation of line segments, and the corresponding updates to the trapezoid graph can be found in textbooks [9, ch. 6]. Briefly, if k trapezoids are crossed by the proxy, then the corresponding k leaves are replaced by copies of a y-node for the segment, which point to $k+1$ new leaves for the resulting trapezoids. (Note that some of the trapezoids crossed may not have been found in the history since they may not have witnesses to conflicts. They did, however, have witnessed conflicts on both ends. This is important, because we could not afford to go walking through trapezoids looking for possible grid points to include in the Voronoi polygon $C_{S_i}(s_i)$.)

Third, we shorten the old proxies that need to be updated. Notice that trapezoids that simply replace $P_{S_{i-1}}(s)$ with a shorter $P_{S_i}(s)$ as the *top* or *bottom* change geometrically, but do not need to change their representations. Thus, this operation is like the inverse of adding a proxy in step two: we erase $P_{S_{i-1}}(s)$ from those trapezoids that do not intersect $P_{S_i}(s)$, merging and splitting as needed.

Fourth, we merge any adjacent trapezoids whose left and right boundaries are defined by grid points that used to be proxy endpoints but are no longer. In the trapezoid graph this just redirects pointers to corresponding leaves.

LEMMA 10. *The leaves of the trapezoid graph corresponding to new trapezoids in the proxy trapezoidation of S_i are $O(1)$ deeper than the leaves for their parent trapezoids in the proxy trapezoidation of S_{i-1}.*

PROOF. Each update in substeps 1–3 adds a single level to the trapezoid graph, and each trapezoid of the proxy trapezoidation can be involved in at most 4 updates (to *top*, *bottom*, *left*, and *right*). □

3.3 Analysis

We can use Mulmuley's general framework of stoppers and triggers (or definers and killers, as described in [9, ch. 9.3]) to analyze the expected time and space for this D2-Voronoi.

LEMMA 11. *The expected size of the D2-Voronoi on n sites is linear in n.*

PROOF. The fact that the proxy trapezoidation is canonical and has linear size implies that the expected amount of work to update while inserting one segment is constant. Thus, the expected size over n insertions is linear. □

LEMMA 12. *The expected time to build the D2-Voronoi on n sites is $O(n \log n \log U)$.*

PROOF. The general analysis framework says that the total number of conflicts that have to be chased through history is $O(n \log n)$. Each conflict is handled by predicate `inConflict` and construction `findProxies`, which give a multiplicative factor of $\log U$. Each update to the trapezoid graph adds at most a constant number of nodes to the search path, so the total time to do the updates is $O(n \log n \log U)$. □

4. CONCLUSION AND OPEN PROBLEMS

Ten years ago Liotta *et al.* [16] described a structure for solving post office queries in double precision. We are happy to report that their structure can also be built efficiently in double precision. To our knowledge, this is the first construction of a planar Voronoi diagram with double precision in sub-quadratic time. We plan to implement the current algorithm to compare with exact arithmetic implementations for building Voronoi diagrams.

It is interesting to note that a randomized incremental construction using the degree 4 InCircle test can be implemented to run in $O(n \log n)$ time, that our earlier degree 3 algorithm is in $O(n \log n + n \log U)$, while this degree 2 algorithm is in $O(n \log n \log U)$. Is there inherent loss of efficiency with restricted predicates? Any lower bound above $\Omega(n \log n)$ would be interesting.

Actually, we believe that by maintaining dynamic convex hulls of grid points in trapezoids and using binary search to test if a witness to a conflict exists before applying our lemmas to extract proxies, we can improve the current algorithm's running time to $O(n \log n \log\log U + n \log U)$. We omit this from the abstract because we prefer the relative simplicity of the current algorithm, and because we would like to reduce the complexity to $O(n \log n + n \log U)$ in the degree 2 case as well.

What other problems are amenable to solution with limitations on the precision? In particular, since degree 2 suffices to compute squared Euclidean distance in any dimensions, what can be said about higher dimensional Voronoi diagrams using restricted predicates?

5. REFERENCES

[1] J.-D. Boissonnat and F. P. Preparata. Robust plane sweep for intersecting segments. *SIAM J. Comput.*, 29(5):1401–1421, 2000.

[2] J.-D. Boissonnat and J. Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. *Comput. Geom. Theory Appl.*, 16(1):35–52, 2000.

[3] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman. Linear time Euclidean distance transform algorithms. *IEEE PAMI*, 17:529–533, 1995.

[4] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Math.*, 1–2:25–47, 2001.

[5] C. Burnikel, J. Könemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *SCG '95: Proceedings of the Eleventh Annual Symposium on Computational Geometry*, pages 418–419, 1995.

[6] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan. Parallel banding algorithm to compute exact distance transform with the GPU. In *I3D '10: Proc. 2010 Symp. on Interactive 3D Graphics and Games*, New York, NY, USA, 2010. ACM.

[7] T. M. Chan. Faster core-set constructions and data-stream algorithms in fixed dimensions. *Comput. Geom. Theory Appl.*, 35(1):20–35, 2006.

[8] T. M. Chan, D. L. Millman, and J. Snoeyink. Discrete Voronoi diagrams and post office query structures without the incircle predicate. In *Proceedings of the Nineteenth Annual Fall Workshop on Computational Geometry*, 2009. `http://cs.unc.edu/~dave/mySite/media/papers/CMS09_FWCG.pdf`.

[9] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag New York, Inc., 3rd edition, 2008.

[10] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[11] S. Fortune and C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, July 1996.

[12] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Trans. Graph.*, 4(2):74–123, 1985.

[13] S. Kahan and J. Snoeyink. On the bit complexity of minimum link paths: Superquadratic algorithms for problems solvable in linear time. *Comput. Geom. Theory Appl.*, 12(1-2):33–44, 1999.

[14] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40, 1965.

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

[16] G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries: An illustration of degree-driven algorithm design. *SIAM J. Comput.*, 28(3):864–889, 1999.

[17] A. Mantler and J. Snoeyink. Intersecting red and blue line segments in optimal time and precision. In *Discrete and Computational Geometry*, number 2098 in LNCS, pages 244–251. Springer Verlag, 2001.

[18] C. R. Maurer, Jr., R. Qi, and V. Raghavan. A linear time algorithm for computing exact Euclidean distance transforms of binary images in arbitrary dimensions. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(2):265–270, 2003.

[19] D. L. Millman and J. Snoeyink. Computing the implicit Voronoi diagram in triple precision. In *WADS*, volume 5664 of *LNCS*, pages 495–506. Springer Verlag, 2009.

[20] D. Priest. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. Ph.D. thesis, Dept. of Mathematics, Univ. of California at Berkeley, 1992.

[21] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[22] K. Sugihara and M. Iri. Construction of the Voronoi diagram for 'one million' generators in single-precision arithmetic. *Proc. IEEE*, 80(9):1471–1484, Sept. 1992.

[23] C.-K. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1-2):3–23, 1997.

[24] C. K. Yap. Robust geometric computation. In *Handbook of Discrete and Computational Geometry*, pages 927–952. CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2004.