Utah State University

DigitalCommons@USU

All Graduate Theses and Dissertations

Graduate Studies

12-2020

Exact Generalized Voronoi Diagram Computation using a Sweepline Algorithm

Daniel Marsden
Utah State University

Follow this and additional works at: https://digitalcommons.usu.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Marsden, Daniel, "Exact Generalized Voronoi Diagram Computation using a Sweepline Algorithm" (2020). *All Graduate Theses and Dissertations*. 7947.
https://digitalcommons.usu.edu/etd/7947

This Thesis is brought to you for free and open access by the Graduate Studies at DigitalCommons@USU. It has been accepted for inclusion in All Graduate Theses and Dissertations by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

EXACT GENERALIZED VORONOI DIAGRAM COMPUTATION USING A SWEEPLINE ALGORITHM

by

Daniel Marsden

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

John Edwards, Ph.D. Major Professor	Xiaojun Qi, Ph.D. Committee Member
Minghui Jiang, Ph.D. Committee Member	D. Richard Cutler, Ph.D. Interim Vice Provost of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2020

Copyright © Daniel Marsden 2020

All Rights Reserved

ABSTRACT

Exact Generalized Voronoi Diagram Computation using a Sweepline Algorithm

by

Daniel Marsden, Master of Science

Utah State University, 2020

Major Professor: John Edwards, Ph.D.
Department: Computer Science

We introduce a geometric algorithm that computes the exact 2D Generalized Voronoi Diagram restricted to point, line segment, and non-overlapping simple polygon sites using a sweepline method. Our algorithm is an extension of Fortune's algorithm which computes the GVD on points and lines. With our approach, the Voronoi diagram can be computed in $O(n \log n)$ run-time with $O(n)$ space requirements. We describe the algorithm, show run-time complexity, and discuss algorithm robustness through testing.

(50 pages)

PUBLIC ABSTRACT

Exact Generalized Voronoi Diagram Computation using a Sweepline Algorithm

Daniel Marsden

Voronoi Diagrams can provide useful spatial information. Little work has been done on computing exact Voronoi Diagrams when the sites are more complex than a point. We introduce a technique that measures the exact Generalized Voronoi Diagram from points, line segments and, connected lines including lines that connect to form simple polygons. Our technique is an extension of Fortune's method. Our approach treats connected lines (or polygons) as a single site.

CONTENTS

LIST OF FIGURES

ACRONYMS

GVD	Generalized Voronoi Diagram
GVC	Generalized Voronoi Cells
RPG	Random Polygon Generator

CHAPTER 1

INTRODUCTION

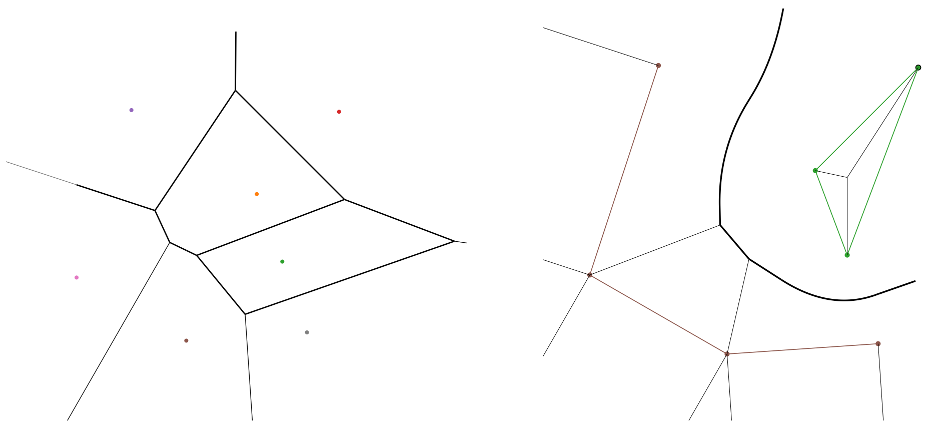


Fig. 1.1: The Voronoi diagram vs the GVD. Figure a shows the Voronoi diagram of point sites, while figure b shows the the Voronoi diagram of polylines and higher-order sites.

1.1 Overview and Motivation

The Voronoi diagram, along with other concepts such as triangulation and convex hull, are fundamental in computational geometry. The 2D Voronoi diagram of points is a plane partitioned into Voronoi regions about sites. For any site s, points in the region about s are closer to s than any other site [4]. Edges of Voronoi regions are equidistant from two or more sites. The Voronoi diagram is good at solving spatial problems. It is especially good at providing solutions to path planning problems [5]. A Voronoi diagram for which sites are objects other than points is often referred to as the Generalized Voronoi Diagram (GVD) [6]. Like the Voronoi diagram of point sites, the GVD is a plane partitioned into individual generalized Voronoi cells (GVCs) where points in the GVC are closer to the contained site than any other site. Applications of the GVD include path planning, computer graphics, pattern recognition, terrain modeling, and reconstruction [6].

Many methods have been introduced to compute the regular Voronoi diagram of points ever since the concept came about many years ago. Computing the GVD is more difficult due to the added complexity of handling higher-order sites and the impact it has on constructing the correct GVD. For this reason, it is common to compute an approximation to the GVD and thus avoid some complexities. In chapter 2 we discuss published methods that compute the regular Voronoi diagram including Fortune's method [1]. We also discuss approximation and exact algorithms for computing the GVD.

Our work presents an algorithm to compute the exact GVD of points, line segments, and polylines including polylines that close into a polygon. Our work follows Fortune's algorithm [1] with additions that handle line segments that are connected on a non-intersecting polyline. An intuitive approach to extending Fortune's algorithm to polylines would be to simply compute the diagram on all points and line segments and then prune out GVD edges that bisect line segments that are connected by a vertex. However, the pruning approach is challenging because Fortune's algorithm requires that y coordinates of all points and vertices are unique (assuming a horizontal sweep line, as we do). Our approach avoids these singularities by handling a polyline (or polygon) as a single site. In chapter 3 we discuss terms and definitions specific to our approach and describe algorithm details in chapter 4. To our knowledge, our work presents the first sweep line algorithm that computes the GVD on a variety of datasets including sets with closely spaced objects.

1.2 Main Contribution

The main contribution of our work is the algorithm definition to compute the exact GVD. Our algorithm breaks down each unique case for handling polyline input. We prove that our approach maintains an optimal runtime complexity of $O(n \log n)$ and space $O(n)$. To the best of our knowledge, our approach introduces the first sweep line method to compute the GVD with optimal worst-case complexity.

In verifying our algorithm, we include the development of the `webGVD` software. `webGVD` is used to validate the capability of the algorithm in practice. With `webGVD` we demonstrate applications of the GVD including shortest path queries and shortest path queries with path diameter requirements. In chapter 5 we illustrate our results and performance on datasets. We also analyze some applications of GVD including shortest path planning with path diameter requirements.

CHAPTER 2
PRIOR WORK

2.1 Fortune's Work

In the paper "A Sweepline Algorithm for Voronoi Diagrams", Steven Fortune [1] introduces a sweep line algorithm for generating the Voronoi diagram of points, line segments, and weighted point sites on a 2D plane. Central to his approach is the geometric transform through the mapping defined in section 2.1 of his paper which allows the Voronoi diagram to be computed. In our work, we follow the usage of this mapping and refer to it as the beach line. His algorithm is a classic algorithm for computing the Voronoi diagram of points but its extension to line segments is largely ignored. The support of line segments introduces some complexity as extra cases need to be handled. We discuss Fortune's method in more detail in chapter 4.

2.2 GVD Approximation Algorithms

Because of the complexity of the GVD, several approximation algorithms have been proposed. Hoff *et al.* [7] and Garrido *et al.* [8] proposed grid-based approximation approaches that are highly efficient and suitable for graphics hardware but that have no approximation guarantees. Our approach avoids grid-based computation since nearest neighbor objects are tracked and stored as they are visited.

K. Sugihara [9] found a way to compute the approximation to the GVD by using the solution from the traditional Voronoi diagram. Other algorithms obtain a solution by computing distance fields between sites. Edwards *et al.* [3] and Coll *et al.* [10], [11] purposed methods using domain decomposition to construct distance fields for measuring an approximation to the GVD in both 2D and 3D. In both methods, quadtrees and octrees are used as data structures for distance field measurement. Edwards *et al.* [3] uses an adaptive algorithm with scalable space requirements when sites are closely spaced. The main benefits of these methods include their easy implementation, efficiency, robustness, and generality. Our approach eliminates the need to decompose the scene since it is not dependent on creating distance fields in mapping the GVD.

Since computing the GVD is processor-intensive, many methods choose to take advantage of the GPU hardware to accelerate the computation. Fischer *et al.* [12] introduced a tangent-plane algorithm for computing the kth-order Voronoi diagram of a set of point sites in image space. Their algorithm uses graphics hardware to compute the distance transform based on the input. Similarly, Hsieh *et al.* [13] purposed a GPU based method to compute an approximation of 3D Voronoi diagram using an incremental algorithm. For practical usage of these methods, they typically require extra hardware and software for GPU processing along with a rigorous compilation process. This makes it difficult for long term support, access, and portability.

Teichmann *et al.* [14] purposed an algorithm to compute the approximation to the GVD of convex polygonal data in 3D. Their work describes a robust adaptive marching algorithm which sub-divides space into tetrahedral and smaller type cells to build the set of Voronoi regions. Likewise, Vleugels *et al.* [15] introduced a general method which divides space into components called axial cells. In both cases, cell intersection is used compute the GVD approximation. These methods often have a $O(n^2)$ worst case runtime complexity when computing the approximation in three dimensions.

2.3 GVD Exact Algorithms

Compared to approximation algorithms, less work has been done on computing the exact GVD when the sites are more complex than a point. Computation typically requires computing all possible site bisectors between neighboring sites. The complexity of GVD computation increases quickly with the number of supported site types. Points and line segments are most common with circle and semi-circle sites being less common.

Yap [16] shows a model for computing the Voronoi diagram from a set of simple curve segments. "Slabs" are placed between segments and later merged with other "slabs" to create the diagram in a divide and conquer fashion. The benefit of this type of approach is that work can be done concurrently. Our approach generates the GVD of polylines and avoids the need for any merging since the exact GVD is generated altogether linearly.

Mark de Berg *et al.* [4] make claim that the exact GVD of a set of n disjoint line segment sites can be computed in $O(n \log n)$ time using $O(n)$ storage using a technique called retraction. Given a set of disjoint line segments the retraction technique creates two discs to map the GVD. The algorithm moves each disc about the plane such that the two discs with radius r always touch a site but never cross the line segment sites. Our approach maintains the same runtime and space complexity while having the capability to compute the exact GVD of polylines.

Imai [17] introduces a topological algorithm to compute the Voronoi Diagram of polygons using this approach resulting in the `PLVOR` program. Here the challenge is finding all nearest neighbors for each newly inserted site. A brute-force search for the nearest neighbors would require $O(n^2)$ time. Our method tracks nearest neighbor sites naturally as each site is visited and stored through the linear progression of the sweep line.

Influenced by Imai [17], Held [18] introduces an enhanced algorithm that uses geometric hashing to reduce the run-time complexity of finding the nearest neighbor. With this approach, the number of grid hash cells must be predetermined. If the number of cells is too small, there will be a high number of collisions. Using too many cells might result in a sparse matrix with no benefit since the algorithm may have to iterate through more cells than objects. The `VRONI` program is the current implementation of the Held algorithm. In his approach, the number of hash cells chosen is largely dependent on the density of the dataset. The higher the density the more cells will be needed for the hashing method to be effective. Our approach functions independently of predetermined variables on both sparse and very dense datasets illustrated later in chapter 5.

Sethia *et al.* [19] published introduce the `PVD` software that computes the GVD on polygon pockets using geometric predicates for robustness. Their approach targets a specific class of polygonal input whereas our method handles polylines and polylines which are closed to a polygon.

Zavershynskyi *et al.* [20] introduce an algorithm for higher-order Voronoi diagrams which relate to the GVD as they compute the Voronoi diagram of points, line segments and polylines. Similarly to our approach, the authors follow Fortune's method [1] for computing the GVD of point and line segment sites. Their approach incurs a higher runtime cost of $O(nk^2 \log n)$ and space cost of $O(nk)$ where k is the level in the tree structure used to store input site information.

Our approach is an improvement from previous methods because it scales well to handle large datasets and computes the exact GVD in $O(n \log n)$ time using $O(n)$ space without the need for predefined runtime variables.

CHAPTER 3
TERMS and DEFINITIONS

3.0.1 Common Notations and Terms

Notation	Description
B(a,b)	Bisector between a and b
I(a, b)	Intersection between a and b
d(a, b)	Euclidean distance from point a and b that is $\|\|p - v\|\|$
l_{p1p2}	The line from point $p1$ and $p2$
$quad(a, b, c)$	The quadratic equation: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

- Point Site. A point in 2D composed of an x and y value $\{x, y\}$
- Point Site Queue Order. Here we follow the ordering outlined by [1] in section 2 and additionally we define the order for comparison between point sites and line segments. Each point p in 2D is sorted in lexicographical order where $p < q$, if any of the following cases are true:
  - $p_y < q_y$
  - $p_y = q_y$ and $p_x < q_x$

  When compared to a line segment site l, $p < l$ is true if any of the following cases are true:
  - $p_y < l_{p1_y}$
  - $p_y = l_{p1_y}$ and $p_x < l_{p1_x}$
- Line Segment Site. A line segment in 2D composed of two connected points l_{p1} and l_{p2}

- Line Segment Queue Order. Each line segment is ordered such that $l1 < l2$ if any of the following cases are true:
  - $l1_{p1_y} < l2_{p1_y}$
  - $l1_{p1_y} = l2_{p1_y}$ and $l1_{p1_x} < l2_{p1_x}$
  - $l1_{p1} = l2_{p1}$ and the cross product of the vectors $v1, v2$ is less than 0 where $v1 = l1_{p2} - l1_{p1}$ and $v2 = l2_{p2} - l2_{p1}$

  In the final case where $l1_{p1} = l2_{p1}$ by using the cross product we can know that $l1_{p2}$ is either left or right of $l2_{p2}$.

- Close Event. A close event C composed of a point C_p which is the point which marks a vertex on the GVD. This vertex is equidistant neighboring sites. A y-value C_y defined as $C_y = C_{p_y} - r$ where r is the radius to all nearest sites.

- Close Event Queue Order. Each close event $C1$ when compared to another event such as a line segment site l, point site p or another close event $C2$ is ordered such that $C1 < l$, $C1 < p$ and $C1 < C2$ in the following cases:
  - $C1_y < l_{p1_y}$
  - $C1_y < p_y$
  - $C1_y < C2_y$

- Beach Line Tree. A binary search tree representing the placement of all nodes and edges (See Fig 3.1).

- Usegment. A Usegment represents the parabola beach line segment about a point site. It is the bisector between the sweep line and a point site (See Fig 3.2).

- Vsegment. A Vsegment represents the V beach line segment about a segment site. It is created from the bisector between the sweep line and line segment site (See Fig 3.2).

- Segment Node. A node in the beach line tree representative of a Usegment or Vsegment (See Fig 3.1).

- Edge Node. Internal nodes in the beach line tree connecting separate segment nodes. Each edge node e is representative of the connection between two adjacent segment nodes (See Fig 3.1).
- Edge. An edge represents the intersection between beach line segments. Each edge is created with a start point and directional data. These components of the edge allow for the algorithm to predict close events as we will discuss later in section 3.3.1.

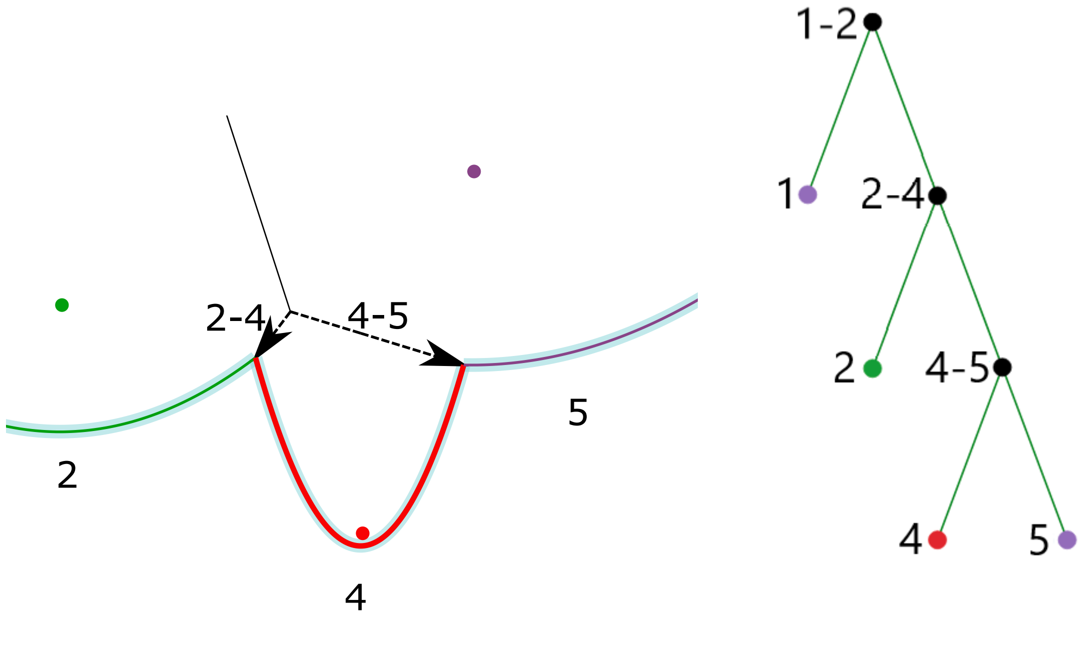


Fig. 3.1: Beach line tree components. In figure a we highlight the Usegment corresponding to node 4 of the tree. Figure b shows the beach line tree T. In figure a we show edge nodes 2-4 and 4-5 which form the parent edge nodes to leaf nodes with adjacent beach line segments. We mark each edge with its directional component. This is the direction of travel along the GVD edge.

3.1 Input Queue

We define the input priority queue Q as the structure containing all input. The sort order is determined by the definition of the object sort order as points are sorted differently than line segments and close events. For the most part, we follow Fortune's approach [1] with the exception that polylines are sorted in a contiguous fashion next to their connected counterparts at specific y values. We note that each line segment site l consists of two endpoints points (l_{p1}, l_{p2}) in 2D where $p1_y \neq p2_y$ and $p1_y > p2_y$. Every point site p is also in R^2. Point sites are sorted by the y component. Line segment sites are sorted by $p1_y$ and in case of a tie are sorted left to right. Close events are ordered based on the y component of the event. Polylines are labeled to differentiate medial axis edges from the GVD. Like in Fortune's method we require sites to be in general position in that y-coordinate of point sites and end points of line segments must be unique.

3.2 Beach Line

The beach line is the boundary of where the structure of the GVD is known and where it is not known – the GVD is known in the region above the beach line. We call the constituent parts of the beach line *Usegments* and *Vsegments*. A Usegment is a parabolic beach line segment that bisects the sweep line and a point, and a Vsegment bisects the sweep line and the line portion of a line segment (See Fig 3.2). We also use the term *Bsegment* to refer to a generic beach line segment, whether it is a Usegment or a Vsegment. Intersections of beach line segments trace out edges of the GVD.

3.2.1 Beach Line Structure

We follow Fortune's [1] suggestion in Algorithm 1 for the usage a binary tree structure. The purpose of the tree is to store each visited site along. Leaf nodes have a one-to-one correspondence with a Bsegment. Internal nodes correspond to intersections of Bsegments (see figure 3.1). Ordering of nodes is based on their Usegment or Vsegment intersection boundaries in 2D. An in-order traversal of the tree visits Bsegments ordered from left to right. The beach line tree forms a hierarchical structure of all objects in a scene. The advantages of using a binary search tree include the ability to track neighboring sites.

3.2.2 Beach Line Segments

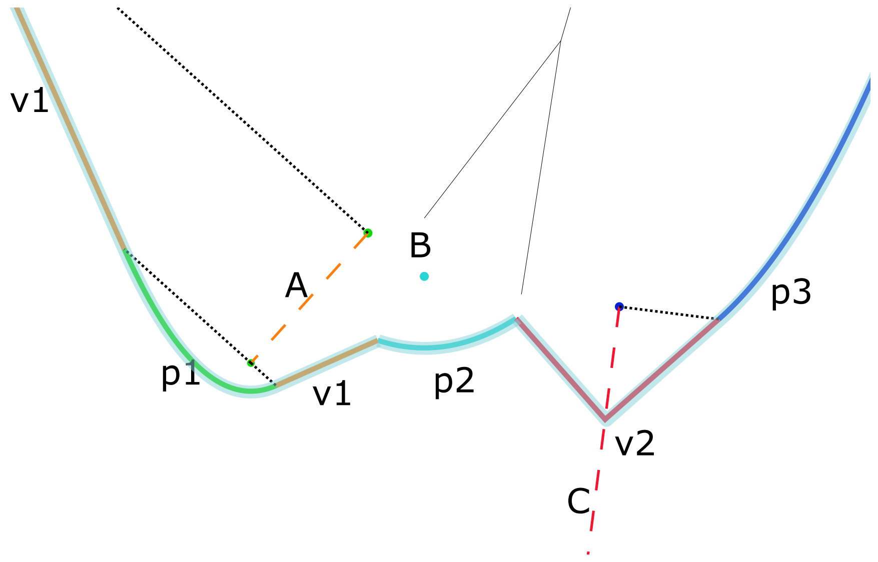


Fig. 3.2: Beach line composition. The beach line, highlighted in light blue, is composed of both lines and parabolic elements. We show individual Usegments ($p1$, $p2$, and $p3$) and Vsegments ($v1$ and $v2$) representing the locus of bisecting points between the sweep line and each site (A, B, and C). Usegments are formed about all point sites including start and endpoints of each segment site. Vsegments are formed from bisecting the line portion of the line segment with the sweep line.

Segment bisectors act as tracking mechanisms for nearest neighbor sites to allow for the computation of node insertion points. Figure 3.2 shows the rendering of both Usegment and Vsegment bisectors. As a bi-product they provide a visualization aid for the development of the GVD. The computation of each bisector is as follows:

- Usegment Bisector. Provided the sweepline location sl_y and the site location p we can compute the site bisector. The resulting bisector is a geometric parabola that follows the vertex form equation, $y = (x - h)^2/4p + k$ where:

$$h = p_x$$

$$k = (sl_y + p_y)/2$$

$$p = (p_y - sl_y)/2$$

- Vsegment Bisector. Provided the sweepline sl and the site location l we can compute the Vsegment. The bisector is composed of two directional vectors V_{v1} and V_{v2} a single point V_p. V_p is the intercept point between the line segment and the sweep line. To compute each directional vector we compute θ as the smallest positive angle between l and sl. We define the angle $\beta = \theta/2$. Using the sin and cos function we can compute each directional vector as:

$$v1 = \{x, y\} = \{cos(\beta), sin(\beta)\}$$

$$v2 = \{x, y\} = \{cos(\beta + \pi/2), sin(\beta + \pi/2)\}$$

3.2.3 Intersections

For the beach line to function properly we must be able to determine the intersection between all types of Bsegments. Intersection points mark the boundary or edge between Bsegments. The order of Bsegment pairs is maintained through a parent edge node in the beach line tree (See Fig 3.1). Intersection points between pairs is done in order with the following definition for each case:

- Usegment to Usegment: From section 3.2.2, we use $h1, k1$, and $p1$ for variables of the first parabola Usegment and $h2, k2$, and $p2$ for variables of the second parabola. We compute the x-value of each intersection using the quadratic formula where:

$$a = 0.25 * (1/p1 - 1/p2)$$

$$b = 0.5 * (h2/p2 - h1/p1)$$

$$c = 0.25 * (h1 * h1/p1 - h2 * h2/p2) + k1 - k2$$

$$xvals = quad(a, b, c)$$

From $xvals$ we determine all intersection points. If a single solution is found we use that point. For multiple solutions we determine which Bsegment is lower at $center_x$

value between each intersection point. If the previous Bsegment is lower at $f(center_x)$ the right most intersection is used otherwise the left most intersection is used.

- Usegment to Vsegment: From the Vsegment, we identify the directional vector v from the Vsegment incident to the Usegment based on the order of the Bsegments. We use the parametric from of the vector $q(t) = q + tv$ where the point q and directional vector v is used from the Vsegment. From the Usegment we use variables h, k, and p as defined in section 3.2.2. *tvals* are computed from using the quadratic equation:

$$a = v_x * v_x/(4 * p)$$

$$b = 2 * v_x * (q_x - h)/(4 * p) - v_y$$

$$c = (q_x * q_x - 2 * q_x * h + h * h)/(4 * p) + k - q_y$$

$$tvals = quad(a, b, c)$$

  Using the *tvals* we can determine the points of intersection along the Usegment. Just like in the case for Usegment to Usegment we determine the solution using $center_x$.

- Vsegment to Vsegment: Before the intersection between Vsegments is computed, we determine the intersection point between segment sites $s1$ and $s2$ where $p = I(s1, s2)$. We compute the intersection as if $s1$ and $s2$ are unbounded lines. If $p \in s1$ or $s2$ we set s. Here we refer to the associated Vsegment of s as s_v. From s know that there can be at most two intersection points along only one directional vector of s_v. We know this because the site s obscures the other directional vector. If $p \notin s1$ or $s2$ then we know that there is at most one intersection between all directional vectors. We know this since the two outer vectors will be obscured. With this information we compute intersection points between non-obscured directional vectors by following the solution for any two parametric equations with where:

$$L = p1 + t * v1 = M = p2 + u * v2$$

To solve this we isolate the unknowns and compute t and u where:

$$t = p2 + u * v2 - p1/v1$$

$$u = p1 + t * v1 - p2/v2$$

Each equation is really two equations for both the x and y component where:

$$(p2_x + u * v2_x - p1_x)/v1_x = t = p2_y + u * v2_y - p1_y)/v1_y$$

$$(p1_x + t * v1_x - p2_x)/v2_x = u = p1_y + t * v1_y - p2_y)/v2_y$$

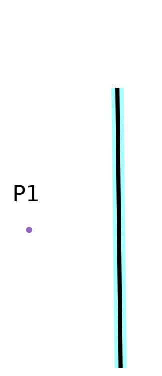
When t and u are isolated we get the result:

$$t = (v2_x * (p1_y - p2_y) + v2_y * (p2_x - p1_x))/(v1_x * v2_y - v1_y * v2_x)$$

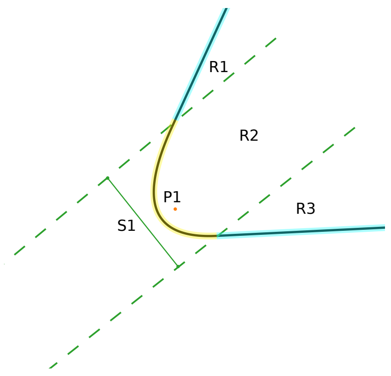
$$u = (v1_x * (p2_y - p1_y) + v1_y * (p1_x - p2_x))/(v2_x * v1_y - v2_y * v1_x)$$

If multiple intersection points are the result from the vector intersections, we follow the same procedure as the two previous cases to find a single solution.

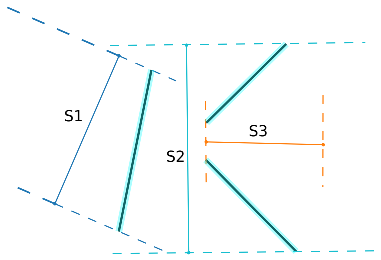
3.3 Site Bisectors



(a)



(b)



(c)

Fig. 3.3: Site bisectors. We illustrate bisectors between site types. Line bisectors are highlighted in turquoise and parabola bisectors are highlighted in yellow. The bisector between point sites is line between $p1$ and $p2$ perpendicular to the vector connecting the two sites (See Fig 3.3a). A Point-segment bisector is a parabola about $p1$ in figure 3.3b. We note that is bisector is bounded to region $R2$ determined by the size of the segment site. Regions $R1$ and $R2$ are formed from the line bisectors between $P1$ and the endpoints of $S1$. Figure 3.3c shows two cases of segment to segment bisector creation. In the first case, a single bisector between $S1$ and $S2$ is formed. It is important to note that their intersection point p is $p \notin S1$ or $S2$. In the second case, the intersection between segments $S2$ and $S3$ is $\in S2$. This results in two bisectors around $S3$ which are within the bounds of $S2$.

We define the computation for site bisectors. The computation of site bisectors is similar to that of Bsegments except bisectors are determined from site to site in general position instead of the sweep line with a fixed orientation.

- Point Point Bisector. The bisector between $p1$ and $p2$ is computed by first computing the mid point as: $m = (p1 + p2)/2$. The vector from $p1$ to $p2$ is then computed as $v = p2 - p1$. Lastly we compute each endpoint for the bisecting line l as:

$$l_{p1} = \{x, y\} = \{v_y + m_x, -v_x + m_y\}$$

$$l_{p2} = -l_{p1}$$

- Segment Point Bisector. For any point $q \notin s$ and not co-linear to s. We compute θ as the angle of rotation of s. Much like the computation of the bisector for a point site arc node, the resulting bisector is a parabola, however, in this case we must account for θ of s. We refer to the angled parabola as a general parabola. The computation of a general parabola can be done by first the computing the parabola about q given s vertex form equation, $y = (x-h)^2/4p + k$ where:

$$h = q_x$$

$$v1 = normalize(s_{p2} - s_{p1})$$

$$v2 = p - s_{p1}$$

$$p = k = cross(v1, v2)_z/2$$

The second component of computing the general parabola requires us to apply a rotational transform of θ to the result of the previous equation. In special cases if q is co-linear to s, then the bisector is the equal to the bisector between q and the closest endpoint point of s either s_{p1} or s_{p2}.

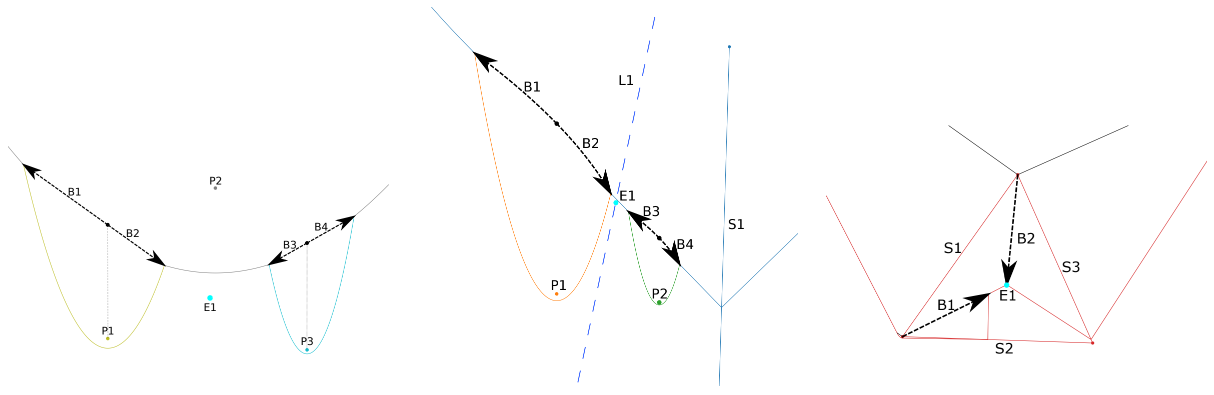
- Segment Segment Bisector. The bisector between $s1$ and $s2$ results in either one or two bisectors $b1$ and $b2$. Each bisector is in parametric form given a point and a directional vector. Angle θ is computed as the smallest angle between sites. For any pair of disjoint segments we compute their intersection point as $p = I(s1, s2)$. If $p \notin s1$ and $p \notin s2$, then the result is a single bisector $b1$ where:

$$b1_p = p$$

$$b1_v = \{\cos\theta, \sin\theta\}$$

If p exist on either $s1$ or $s2$ a second bisector is created in the same fashion except angle θ is the large angle between sites. For connected line segments we know only a single bisector exists within the bounds of each segment. We also know that it is bisector created from the smallest angle which must be less than π. With this information we compute a single bisector just like in the two previous cases with θ as the smallest angle between the sites.

3.3.1 Bisector Intersection



(a) (b) (c)

Fig. 3.4: Directional Bisectors. We show each bisector mapped with a directional component. As edges are formed between Bsegments the direction of the edge is determined by site, sweep line and order information. In Fortune's paper [1] these are called half edges. Each half edge follows the bisector between sites. In figure 3.4a we show individual half edges between point sites. Using the intersection from $B2$ and $B3$ we compute the close event at $E1$. Figure 3.4b shows the half edges formed between a point site and a line segment site. Again, $E1$ is computed except the half edges $B2$ and $B3$ are parabolic bisectors. In computing $E1$ the line $L1$ is purposed for the re-use of ray to parabola intersect methods. $L1$ is defined as the bisector between points $P1$ and $P2$. We provide an example of half edges between segment sites in figure 3.4c. From two half edges we can determine the point $E1$ equidistant between $S1$, $S2$ and $S3$.

Bisector intersection is important for determining equidistant points between a group 3 or more sites. The intersection of bisectors follows the same equations for that of Bsegments as discussed in section 3.2.3 except for some differences for general parabolas and lines. In these cases, angle θ is used to determine the angle of rotation of the parabola. The bisecting line is transformed using a rotational matrix. We then compute the *tvals* using the quadratic equation to determine the intersection points along the general parabola.

CHAPTER 4
ALGORITHM

4.1 Fortune's Algorithm

We give a brief description of Fortune's algorithm [1] here. While most descriptions treat only point sites, we include Fortune's handling of line segment sites as well.

Let G be a collection of points, line segments, connected line segments, and simple polygons. GVD(G) computes the GVD using the elements of G as sites. We maintain two lines, a sweep line, and a beach line. The sweep line, by our convention, extends horizontally and travels down (along the negative y axis). We execute changes to the current GVD state when the sweep line intersects "events" (defined below). The locus of points equidistant from the sweep line and a point site is a parabola and similarly, the locus of points equidistant from the sweep line and a line segment site is a line segment (for the site itself) and two parabolas (for the endpoints of the site). The beach line is a collection of line segments and parabolas (see Figure 3.2) representing the locus of points equidistant from the sweep line and sites above the sweep line.

As the sweep line travels down it encounters "events", which are points at which the hierarchical structure of the beach line changes. For example, a point site is a *point event* that adds a Usegment to the beach line. Consider a line segment site with endpoints a and b. By convention, $a_y > b_y$. a is a specialized site event called a *start segment event* that adds two Usegments split by a Vsegment (see Figure 3.2). The Vsegment bisects the line segment site and the sweep line, one bisection on the left, and one on the right. The two Usegments are subsets of the parabola bisecting endpoint a and the sweep line and lie to the left and right of the bisecting Vsegment. b is an *end segment event* that inserts a Usegment splitting the Vsegment opened by event a. In addition to site events, close events represent points at which beach line segments get "squeezed" out by neighboring segments. Determining where these close events belong is a matter of finding the point equidistant from the points or lines corresponding to the beach line segment being closed and its neighbors on either side.

Assume sites are in general position in that all points sites and line segment site endpoints have unique y coordinates. Events, both site and close events, are reverse sorted by y. Line segment sites add two events, one for each endpoint (a start segment event and an end segment event). A priority queue Q is typically used to maintain the events, giving $O(\log n)$ performance for insertions and removals, where n is the total number of events. See Algorithm 1. Our extension to Fortune's algorithm is in the *AddBSegment* subroutine (line 7).

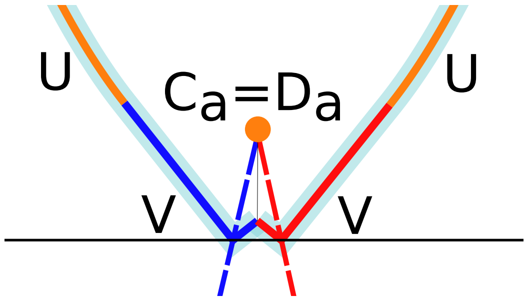
4.1.1 Polylines

We extend Fortune's algorithm to handle polylines, including polylines that are closed into a polygon. Our contribution is updating the beach line structure appropriately when encountering vertex point sites (sites that are endpoints of two line segments). In the following discussion, we use $b((U|V)^*)$, where * is the Kleene star, to describe a series of beach line segments defined by Usegments (U) and Vsegments (V). For example, $b(UVVU)$ corresponds to the four segments shown in Fig. 4.1a.

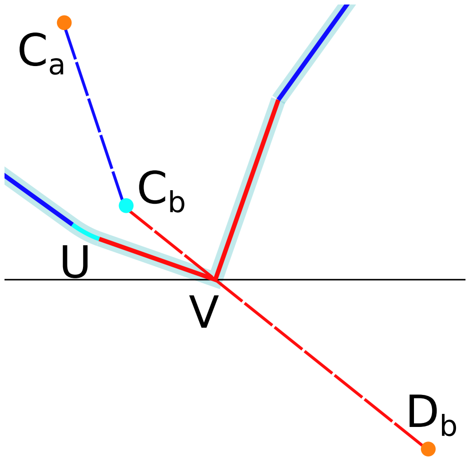
Inserting into the beach line

When encountering site events, our contribution is appropriately handling a vertex event e that is the vertex of two adjacent line segments C and D. Where the original algorithm handles the $b(UVU)$ beach line segments corresponding to a single line segment (see Fig. 3.2), our extension handles the $b(UVVU)$, $b(VU)$, and $b(UV)$ beach line segments corresponding to vertex sites (see Fig. 4.1).

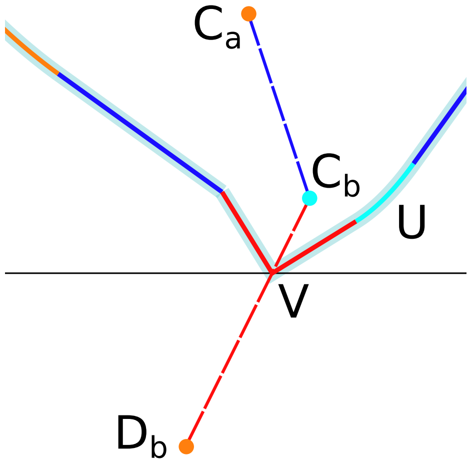
Algorithm 2 describes the algorithm for updating the beach line structure when encountering a point or vertex event. If the event is a vertex site event (line 2) we test for three cases. The first is the $C_a = D_a$ case (Fig. 4.1a) where we insert four segments into the beach line: a Usegment for the vertex itself, split into two pieces by two Vsegments, one for each line segment. The second case is when the line segments make a left-hand turn from C



(a) $B(UVVU)$



(b) $B(UV)$



(c) $B(VU)$

Fig. 4.1: Three configurations of vertex events with beach line highlighted in light blue. a $C_a = D_a$ case, where four beach line segments are inserted. b $C_b = D_a$ left-hand turn case, where two beach line segments are inserted. c $C_b = D_a$ right-hand turn case, where two beach line segments are inserted. The case where $C_b = D_b$ is not shown as it is handled the same as a non-vertex point site.

to D (lines 9-10 and Fig. 4.1b). In this case, the Usegment for the vertex appears only on the left with the Vsegment for the line segment on the right. The third case is symmetric to the second (lines 11-12 and Fig. 4.1c). There is a fourth case, where $e = C_b = D_b$ (lines 14-15), but it turns out that this is identical to the non-vertex case (lines 17-18) with one caveat: instead of splitting an existing beach line segment (lines 23-24), the new Usegment will be inserted between two existing Vsegments, which are the two Vsegments bisecting the two line segments. It is tempting to handle inserting the Usegment between two existing Bsegments as a special case when $e = C_b = D_b$. However, we note that any new Bsegment to be born exactly at the intersection of two existing Bsegments, so this is a case not unique to our extension.

In the original sweep line algorithm, all Bsegments in the interior of the beach line had a corresponding close event that generated a Voronoi edge. In our case, however, when a Bsegment is surrounded by other Bsegments from the same polyline, handling of the close event (line 10 of Algorithm 1) does not create an edge. Handling this case requires a little more bookkeeping – each site must carry a label.

Algorithm 1 GVD

```
1:  Initialize beach line $L$
2:  Populate priority queue $Q$ with point sites and endpoints of line segment sites
3:  **while** $Q$ is not empty **do**
4:      $e \leftarrow$ pop from $Q$
5:      $dirty \leftarrow$ empty list
6:      **if** $e$ is a site **then**
7:          $dirty \leftarrow$ AddBSegment($e$)
8:      **end if**
9:      **if** $e$ is a close event **then**
10:         $dirty \leftarrow$ RemoveBSegment($e_{bsegment}$)
11:     **end if**
12:     **for** $Bsegment$ in $dirty$ **do**
13:         $c \leftarrow$ CreateCloseEvent($Bsegment$)
14:         add $c$ to $Q$
15:     **end for**
16: **end while**
```

Algorithm 2 `AddBSegment`

```
Accept as an argument site event $e$
if $e$ is a vertex then
    Let $C$ and $D$ be adjoining line segments
    if $e = C_a = D_a$ then
        $t \leftarrow b(UVVU)$
    else if $e = C_b = D_a$ then
        $u \leftarrow C_a - C_b$
        $v \leftarrow D_b - D_a$
        if $(u \times v) \cdot (0, 0, 1) > 0$ then
            $t \leftarrow b(UV)$
        else
            $t \leftarrow b(VU)$
        end if
    else if $e = C_b = D_b$ then
        $t \leftarrow b(U)$
    end if
else
    $t \leftarrow b(U)$
end if
if $T$ not defined then
    $T \leftarrow t$
else if $\exists c \in T$ s.t. $c_{xmin} < e_x < c_{xmax}$ then
    Find $c \in T$ s.t. $c_{xmin} < e_x < c_{xmax}$
    Split $c$ with $t$
    ret $\leftarrow c_{\text{left}}, c_{\text{right}}$
else
    Find $c, d \in T$ s.t. $c_{xmax} = e_x = d_{xmin}$
    Insert $t$ between $c$ and $d$
    ret $\leftarrow c, d$
end if
return ret
```

4.1.2 Complexity

Fortune's algorithm is $O(m \log m)$ time and $O(m)$ in space [1], where m is the number of sites. As our sites are arbitrarily complex, our complexity is derived in terms of the number of point sites summed with the number of vertices in polyline sites, which we call n. Using this definition, Fortune's algorithm would remain $O(n \log n)$ and $O(n)$ in space and time, respectively.

Lemma 1. *Algorithm* `AddBSegment` *(Algorithm 2) is* $O(\log n)$ *in time.*

Proof. At most 4 bsegments are added to tree T. The 4 bsegments are leaf nodes of tree t, so the tree has $O(1)$ nodes and can be constructed in linear time. One insertion (the root of tree t) into tree T is made, requiring a $O(\log n)$ search and $O(1)$ insertion. □

Theorem 1. *Algorithm GVD (Algorithm 1) is* $O(n \log n)$ *in time.*

Proof. The main while loop in Algorithm 1 executes $O(n)$ times. By Lemma 1 `AddBSegment` is $O(\log n)$. All other portions of the algorithm are as in the original algorithm, resulting in $O(n \log n)$ run-time complexity. □

Theorem 2. *Algorithm GVD (Algorithm 1) is* $O(n)$ *in space.*

Proof. Instead of the one Bsegment added by Fortune's algorithm in `AddBSegment`, we add at most 4. No other changes to Fortune's algorithm affect space. □

CHAPTER 5

RESULTS

5.1 Implementation

The enhanced GVD algorithm is implemented in JavaScript and, as such, is run in a browser. As performance isn't our primary goal at this stage, we chose this implementation strategy for explanatory and educational purposes. JavaScript was chosen to make development, testing, replication, and adoption simple because the code needn't be compiled. The software implementation is referred to as `webGVD`.

Results for the `webGVD` were gathered using various datasets presented in published papers.

- Cities Dataset. With the publication of the article "Benchmarks for Grid-Based Pathfinding" by N. Sturtevant [2] the cities dataset was also released. We use this dataset for stress testing our implementation. Shown in figure 5.2 we compute GVD along with its counter-part the medial axis of many line segments. The reason for so many line segments is due to the fact that this data was in a pixel grid type format instead of point to point format. All outer edges and pixels are traced and converted to individual segments.

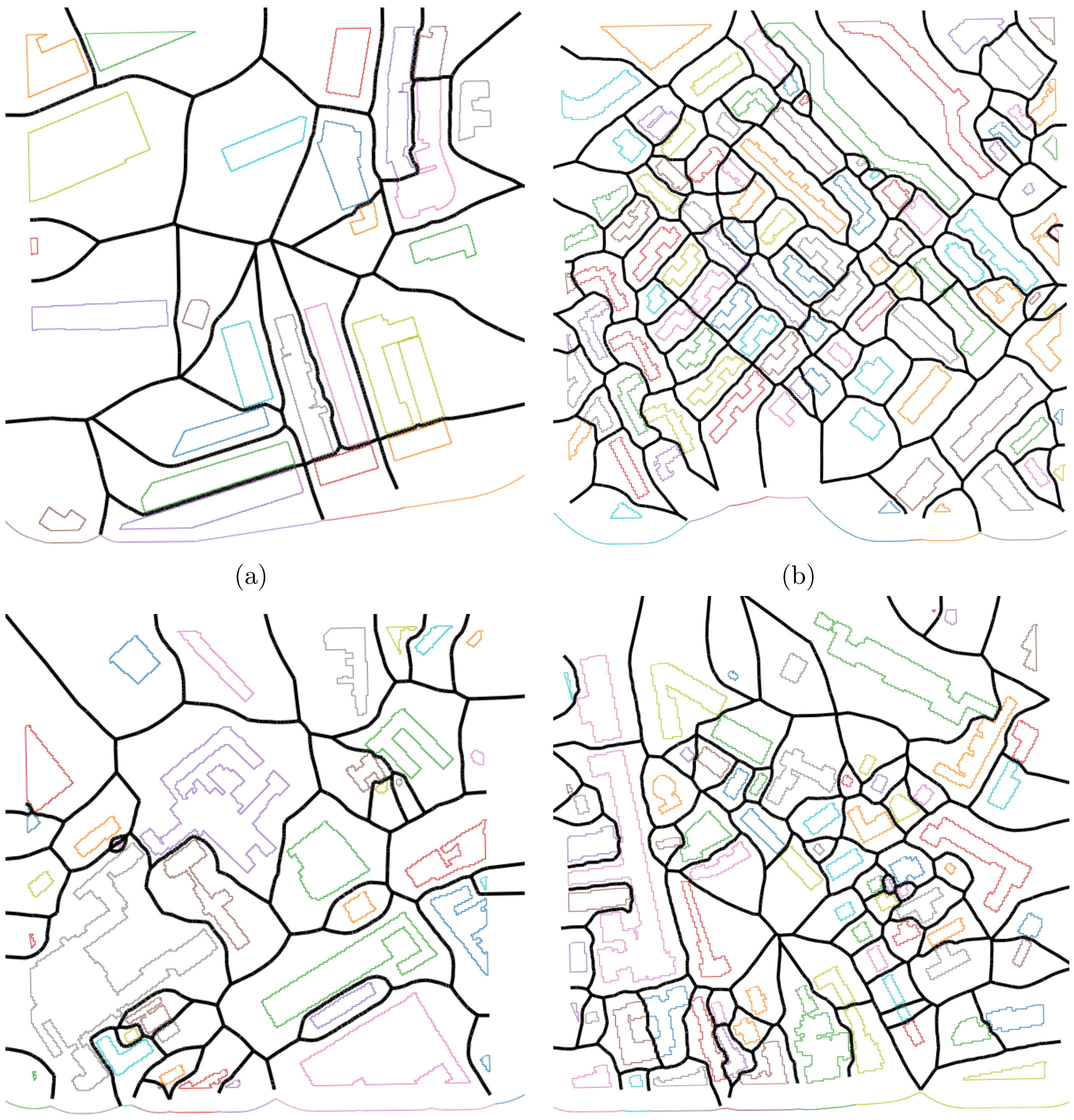


Fig. 5.1: City Data GVD. In his work, [2] provides a number of useful city data-sets that can be used as input into our algorithm. This shows the computed the GVD for some of the cites in the cities dataset including the Sydney a, Boston b, Berlin c and Moscow d. At the bottom of each figure the beachline or wave-front can be identified as plays a major role in generating the GVD. Using the computed results we can extract helpful information such as identifying the most narrow streets or alleyways in a city or find the shortest path from one side of the city to the other.

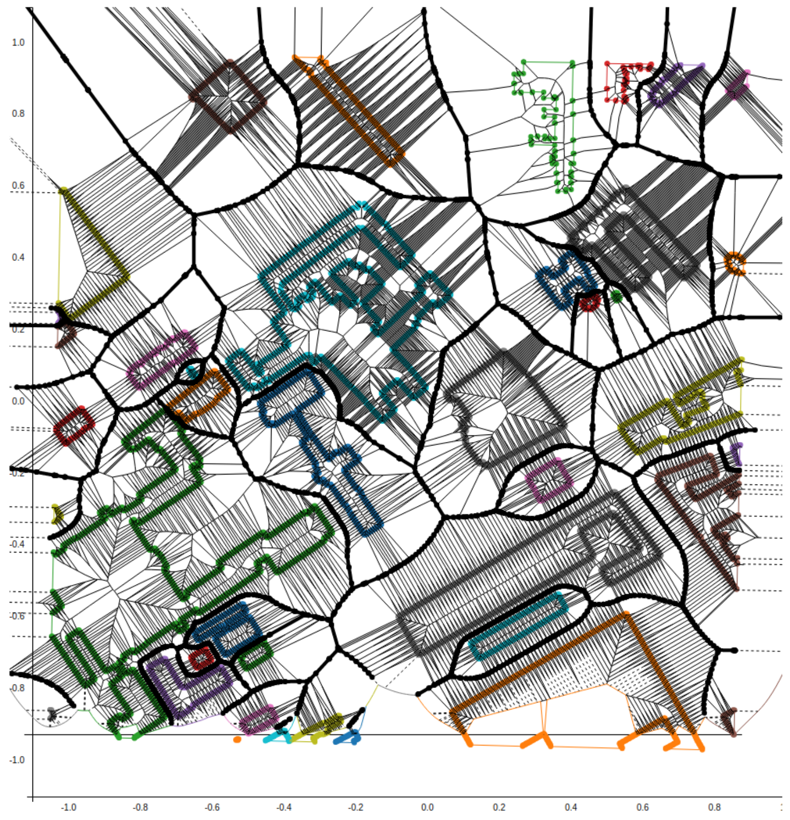
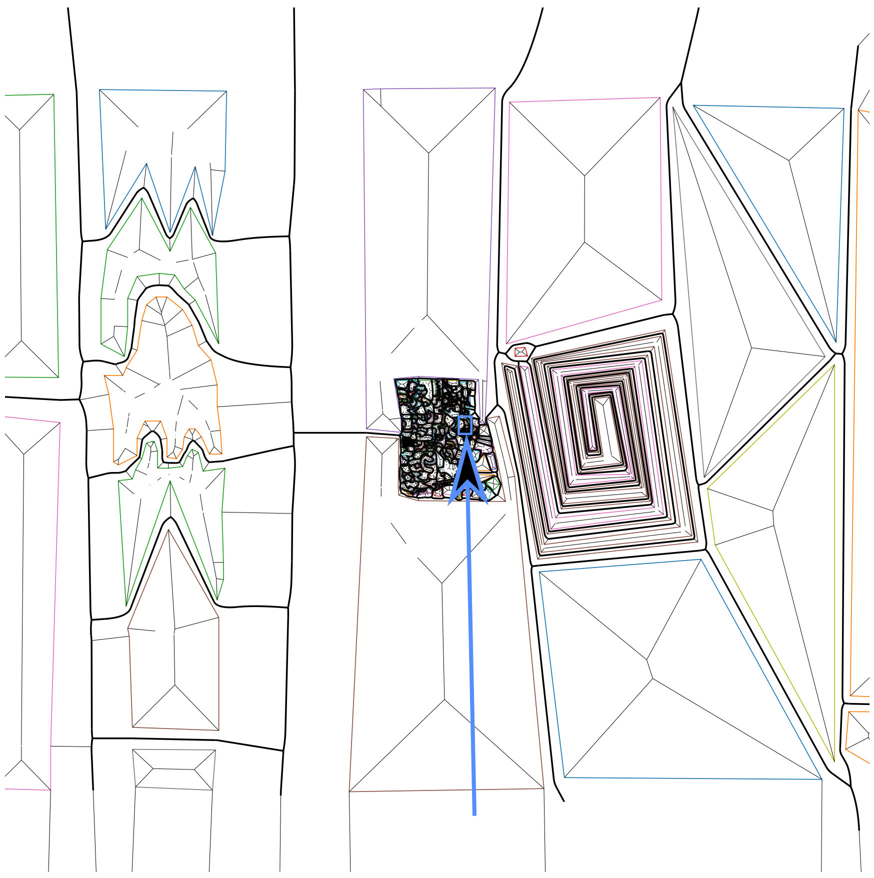
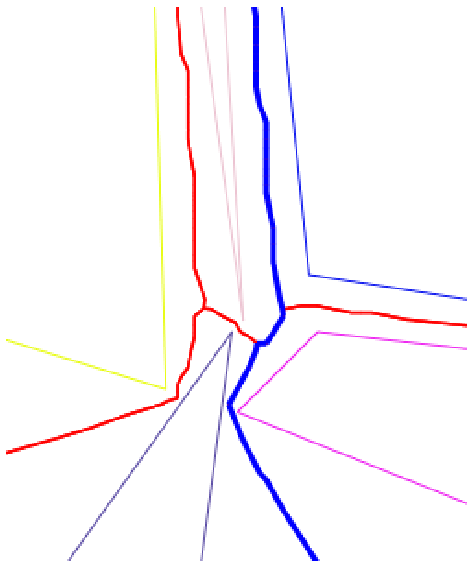


Fig. 5.2: Medial Axis. As a measure of robustness we show the computed GVD along with it's counterpart the medial axis. The medial axis is computed similarly to the GVD but encodes additional shape information. The GVD is show in the thicker lines while the medial axis is shown with thinner lines. Indeed, the medial axis is a complete shape descriptor. Here the medial axis of the Berlin city dataset [2] is shown using thin lines alongside the GVD indicated by thicker lines.

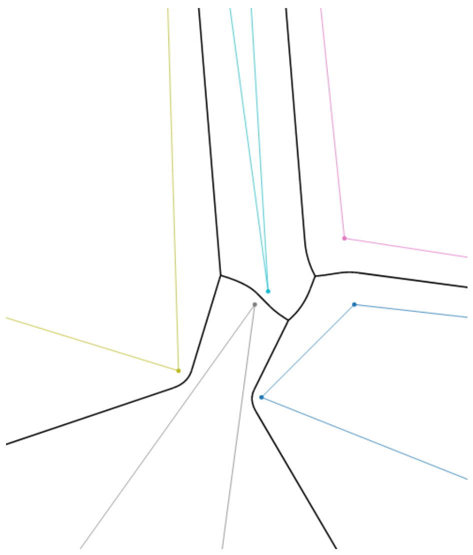
- Maze Dataset. In article "Approximating the Generalized Voronoi Diagram of Closely Spaced Objects" by Edwards *et al.* [3] the authors used a dataset to test algorithm robustness on densely packed objects. In figure 5.3b we compare their computed GVD to ours at roughly 80000 zoom resolution. Additionally we show our algorithm with the capability to find the shortest path through the maze in figure 5.8.



(a)



(b)



(c)

Fig. 5.3: Approximate GVD compared to the exact GVD. In figure 5.3a we show the context area to be zoomed in at roughly 80,000 zoom resolution. We use our results from the maze dataset [3] to compare the computed approximation to the GVD (figure 5.3b) and our computed exact GVD (figure 5.3c).

- RPG Dataset. In a recent article "On Generating Polygons: Introducing the Salzburg Database" by G. Eder *et al*, the authors released the Salzburg Database of Geometric Inputs and with it the random polygon generator (RPG) [21]. We use RPG for performance testing using the generated unit square (See Fig 5.6b). We also use RPG to for verification purposes of `webGVD` through the generation of the cluster dataset and star dataset in figures 5.4 and 5.5.

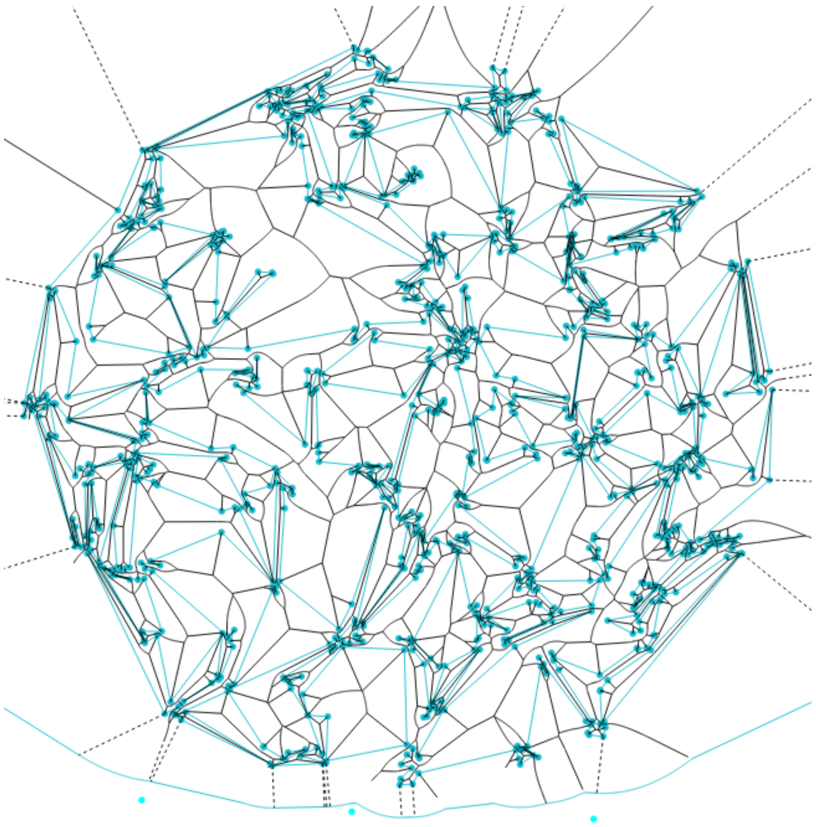


Fig. 5.4: RPG Cluster Dataset.

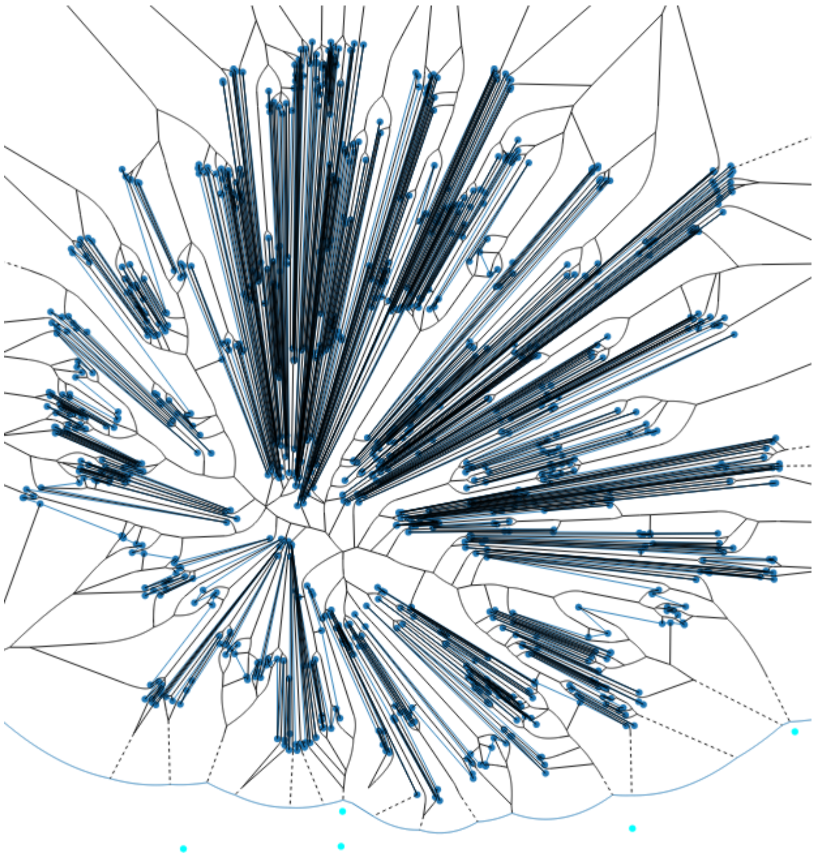
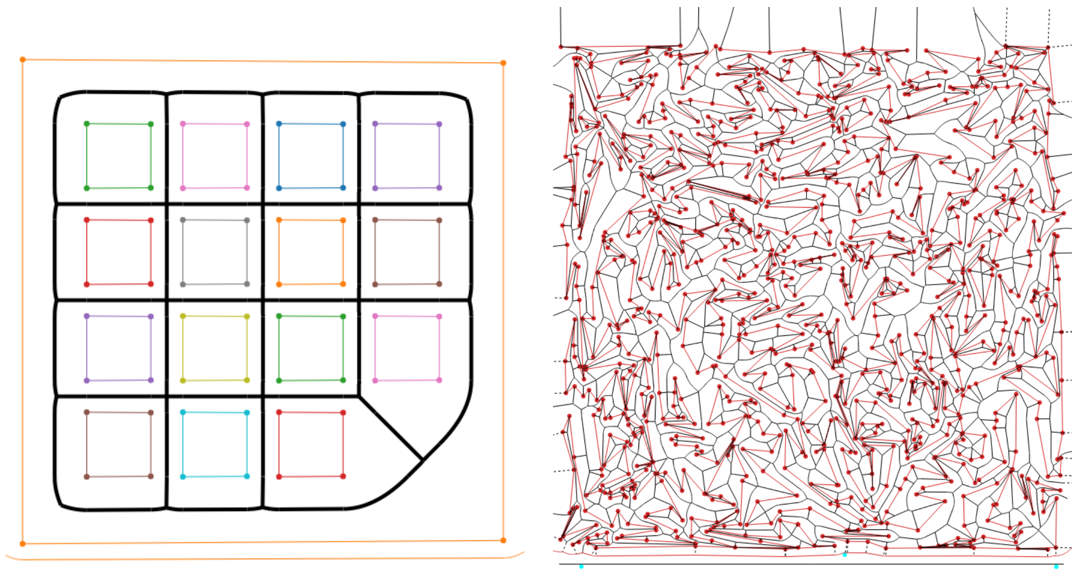


Fig. 5.5: RPG Star Dataset.

- Holes Dataset. The Holes dataset was introduced by Held [18]. We were able to generate it for performance and verification purposes as shown in figure 5.6a.

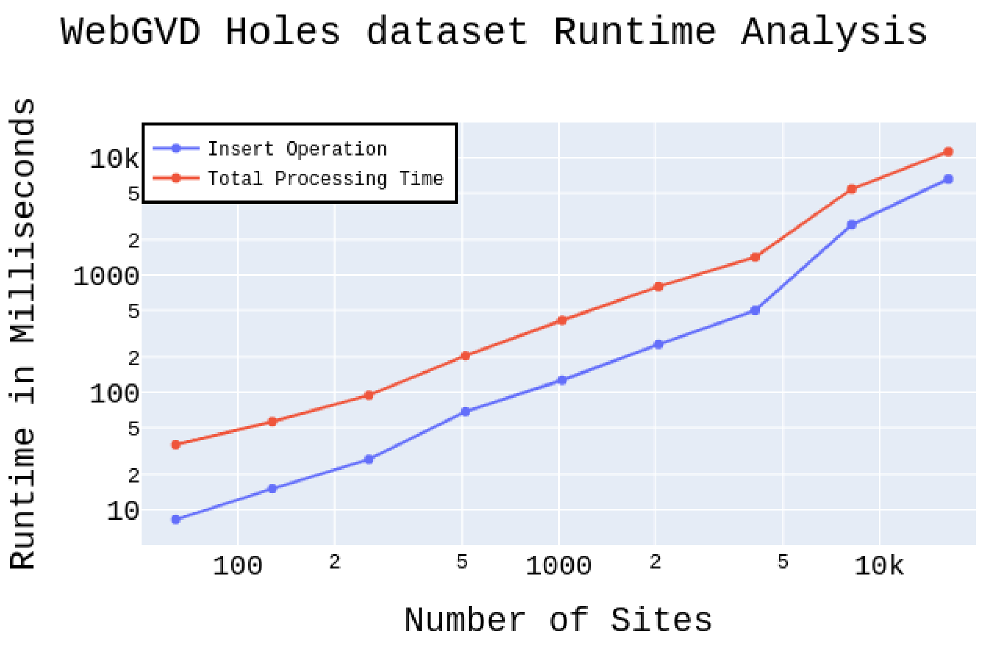
5.2 Run-time Analysis



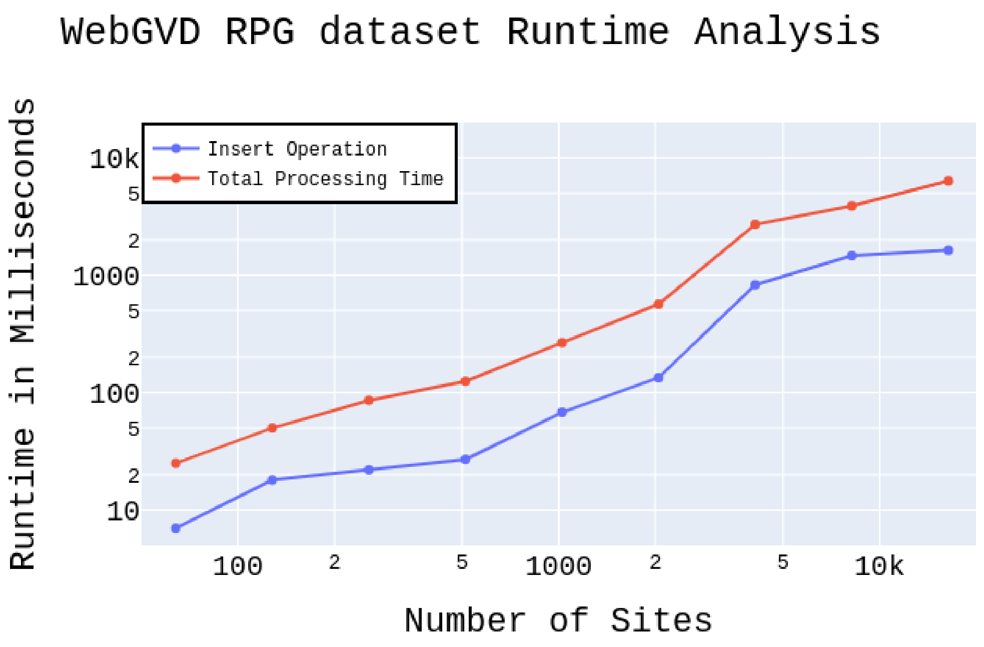
(a) (b)

Fig. 5.6: Generated Datasets. We illustrate the Holes with 64 segments and the RPG Unit Square with 1024 segments.

We use both the Holes [18] and RPG [21] datasets to gather our timing results to show scalability (See Fig 5.6). These datasets were chosen due to their flexibility to scale the size of n without generating invalid data. Fig. 5.7 shows results from a scaling test. With the two datasets, we measure the runtime as we scale up in the number of sites. The results show performance in practice is well within the $O(n \log n)$ worst-case bound. Table 5.1 shows runtimes from tests on various datasets with comparison to other algorithms where available. We expect that implementing our algorithm in a high-performance language would put our results closer to but not necessarily on par with VRONI [18].



(a)



(b)

Fig. 5.7: Run-time Analysis. The log-log plots show run-time spent on the insert operations and the total processing time of the algorithm.

5.3 Performance Vs Other Algortihms

For good measure we show recorded benchmarks from `webGVD`. We compare our results to similar algorithms. We note that `PLVOR`, `PVD`, `VRONI` were all implemented using optimized compilation [18].

Dataset	Our time (s)	Other time (s)	# sites
Maze [3]	2.0	2.0 [3]	4,456
Sydney [2]	1.3	-	3,260
Berlin [2]	1.7	-	4,239
Boston [2]	3.0	-	7,813
Moscow [2]	1.5	-	4,081

Table 5.1: Results of timing tests. Where available, timings on the same dataset using a different algorithm are also provided.

Size	`PLVOR`	`PVD`	`VRONI`	`webGVD`
64	1.359	0.164	0.109	0.390
128	0.922	0.160	0.107	0.392
256	0.657	0.162	0.108	0.336
512	0.565	0.158	0.115	0.244
1024	0.517	0.167	0.114	0.259
2048	0.511	0.167	0.117	0.276
4096	0.561	0.175	0.126	0.661
8192	0.643	0.185	0.136	0.474
16384	0.898	0.209	0.148	0.387

Table 5.2: RPG Dataset Results (time per segment in ms)

Size	PLVOR	PVD	VRONI	webGVD
64	1.250	0.469	0.156	0.516
128	0.859	0.625	0.078	0.501
256	0.625	1.289	0.078	0.367
512	0.508	3.477	0.117	0.336
1024	0.449	15.53	0.098	0.381
2048	0.444	99.23	0.098	0.314
4096	0.532	732.2	0.105	0.365
8192	0.621	5671.0	0.117	0.438
16384	0.982	n/a	0.132	0.504

Table 5.3: Holes Dataset Results (time per segment in ms)

5.4 GVD Applications

We discuss applications of the GVD including shortest path applications and path planning with path diameter requirements. The GVD allows for elegant solutions to these types of problems. By coupling our algorithm with Dijkstra's algorithm we can analyse paths through each dataset. Figure 5.8 shows our capability to navigate the maze dataset using the GVD.

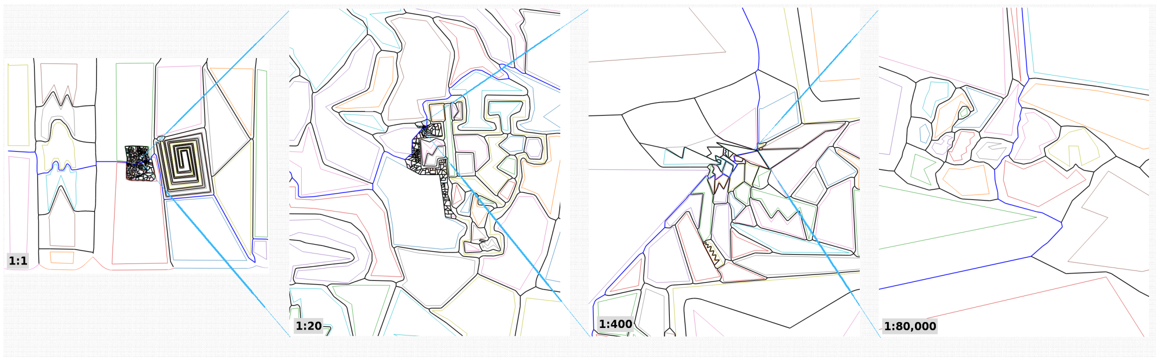
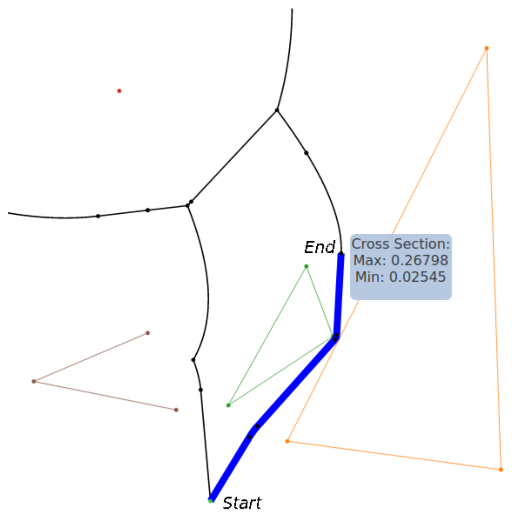


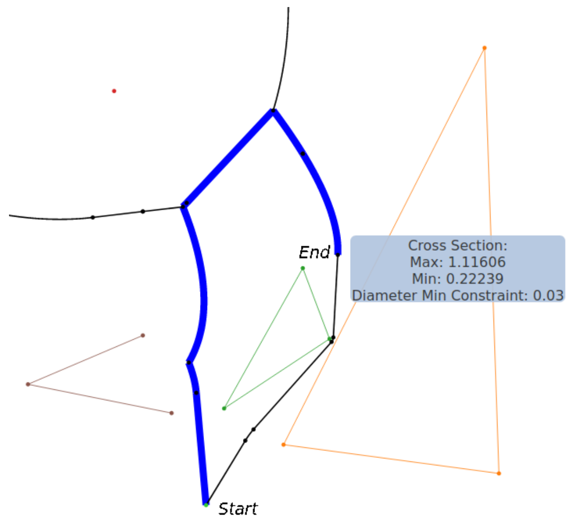
Fig. 5.8: Shortest Path Capability. We illustrate our ability to determine the shortest path through the maze dataset [3] which tests the algorithm's ability to compute the GVD at multiple scales.

In Dijkstra's path algorithm, a start and end are known. We begin by evaluating all paths leading outward from the start position. Paths with the lowest cost are chosen before those with higher costs. The algorithm finishes when the end is reached. Using the GVD we retrieve the exact cost or length of each edge between sites. The computed length is based on the equidistant path which is the path taken by someone traveling down the very center of the path.

We can use the GVD to compute new routes when under constraints such as a required minimum path diameter. Consider a scenario where an object is transported through a maze. Depending on the size of the object it might be impossible to take the shortest path if at any point the path that is smaller than the object itself. In this case, a new route is computed to allows for the object to pass freely through the maze. Figure 5.9 illustrates this scenario where a longer but less narrow path is chosen.



(a)



(b)

Fig. 5.9: Path Planning with Diameter Requirements. We show the computed GVD as the black line and the computed shortest path with the blue line. Figure a shows the shortest path through the objects with no minimum size requirements. All units are in relations to unit grid comprising the scene from -1.0 to 1.0. The shortest path passes through a narrow section. When the shortest path is limited only to routes with a minimum diameter of at least 0.03 units a different route that is less narrow must be chosen (See Fig b).

To do this, we modify Dijkstra's algorithm slightly. We pass the requirement criteria into the algorithm. In this case, we pass in d_{min} as the minimum diameter. We modify the algorithm to check the minimum path diameter P_{min} before choosing which path to take. Since the GVD marks all points equidistant between sites the P_{min} is of any path is known. If $P_{min} < d_{min}$, we add an infinitely high cost to the potential path. Since the cost is too high, the algorithm will naturally ignore the invalid path and find an alternative cheaper route. The GVD is well-suited to handle these types of problems. Exact path requirements are easily stored and retrieved in the result itself.

CHAPTER 6
Conclusion

6.1 Conclusion

This thesis presents our work to advance the capability of the sweep line method introduced by Fortune [1]. We have presented a sweep line algorithm to compute the exact GVD in the Euclidean plane for point, line segment, polyline, and polygon sites. Despite no mention of polygons in our algorithm description, the implementation works as polygons are handled naturally. We have derived identical time and space bounds to Fortune's original algorithm and shown experimental behavior consistent with those bounds in tests. We have also shown robustness across a variety of datasets. Our method scales well with large datasets without extra runtime configurations or variables.

Our work is different from previous methods because we focus on the utilization of a sweep line to maintain optimal runtime complexity. The large majority of previous methods use incremental algorithms to compute the GVD. With these methods, it is challenging to avoid a $O(n^2)$ complexity for nearest neighbor discovery.

In our results, we present practical applications of the GVD. We show how the GVD can be exploited to compute the shortest path through a maze of closely spaced objects. We illustrate how our method is well suited in finding solutions for path planning problems with path diameter requirements (See Fig 5.9).

6.1.1 Future Work

Our work brings a sweep line approach forward as a possible framework for handling higher-order sites. Whether our approach would be suitable for curved or circular sites is a topic for future study. We have shown that the sweep line method can be extended and generalized to handle higher-order data naturally. The sweep line approach is well suited to handle general input as long as event order and bisector segment mapping is handled correctly. Whether or not the sweep line approach may prove effective in generating the 3D Voronoi diagram is another topic for future study. The Voronoi diagram in 3D remains largely unexplored in general.

6.1.2 Implementation Benefits

The benefit of implementing our algorithm in JavaScript allows it to be easily adaptable in a variety of environments since it is run in the browser. The source code of our work is hosted publicly. Future researchers or contributors can easily submit changes and enhancements. Additionally, we provide public access to our application on a hosted web page [1]. Our hope is that is can be used as a platform for getting more people interested in Voronoi diagrams, the sweep line method, and computational geometry concepts in general.

[1] https://edwardsjohnmartin.github.io/gvd-fortune/

Bibliography

[1] S. Fortune, “A sweepline algorithm for voronoi diagrams,” *Algorithmica*, vol. 2, p. 153, 1987.

[2] N. Sturtevant, “Benchmarks for grid-based pathfinding,” *Transactions on Computational Intelligence and AI in Games*, vol. 4, pp. 144 – 148, 2012.

[3] J. Edwards, E. Daniel, V. Pascucci, and C. Bajaj, “Approximating the generalized voronoi diagram of closely spaced objects,” *Comput. Graph. Forum*, vol. 34, pp. 299–309, 2015.

[4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry*, 1997, pp. 147–148, 254.

[5] P. Bhattacharya and M. L. Gavrilova, “Roadmap-based path planning - using the voronoi diagram for a clearance-based shortest path,” *IEEE Robotics Automation Magazine*, vol. 15, pp. 58–66, 2008.

[6] F. Aurenhammer, R. Klein, and D.-T. Lee, *Voronoi Diagrams and Delaunay Triangulations*, 2013.

[7] K. Hoff, T. Culver, J. Keyser, M. C. Lin, and D. Manocha, “Interactive motion planning using hardware-accelerated computation of generalized voronoi diagrams,” in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, vol. 3. IEEE, 2000, pp. 2931–2937.

[8] S. Garrido, L. Moreno, M. Abderrahim, and F. Martin, “Path planning for mobile robot navigation using voronoi diagram and fast marching,” in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2006, pp. 2376–2381.

[9] K. Sugihara, “Approximation of generalized voronoi diagrams by ordinary voronoi diagrams,” *CVGIP: Graphical Models and Image Processing*, vol. 55, pp. 522 – 531, 1993.

[10] N. Coll, I. Boada, and J. A. Sellarès, “The voronoi-quadtree: construction and visualization,” 2002.

[11] I. Boada, N. Coll, N. Madern, and J. A. Sellarès, “Approximations of 2d and 3d generalized voronoi diagrams,” *International Journal of Computer Mathematics*, vol. 85, pp. 1003–1022, 2008.

[12] I. Fischer and C. Gotsman, “Fast approximation of high-order voronoi diagrams and distance transforms on the gpu,” *Journal of Graphics Tools*, vol. 11, no. 4, pp. 39–60, 2006.

[13] H.-H. Hsieh and W.-K. Tai, “A simple gpu-based approach for 3d voronoi diagram construction and visualization,” *Simulation modelling practice and theory*, vol. 13, no. 8, pp. 681–692, 2005.

[14] M. Teichmann and S. Teller, "Polygonal approximation of voronoi diagrams of a set of triangles in three dimensions," in *Tech Rep 766, Lab of Comp. Sci., MIT*, 1997.

[15] J. M. Vleugels and M. H. Overmars, *Approximating generalized Voronoi diagrams in any dimension.* Utrecht University, 1995, vol. 1995.

[16] C. Yap, "Ano(n logn) algorithm for the voronoi diagram of a set of simple curve segments," *Discrete and Computational Geometry*, vol. 2, pp. 365–393, 1987.

[17] T. Imai, "A topology oriented algorithm for the voronoi diagram of polygons," 1996, pp. 107–112.

[18] M. Held, "Vroni: An engineering approach to the reliable and efficient computation of voronoi diagrams of points and line segments," *Computational Geometry*, vol. 18, pp. 95 – 123, 2001.

[19] S. Sethia, M. Held, and J. S. B. Mitchell, "Pvd: A stable implementation for computing voronoi diagrams of polygonal pockets," vol. 2153, 2001, pp. 105–116.

[20] M. Zavershynskyi and E. Papadopoulou, "A sweepline algorithm for higher order voronoi diagrams," in *2013 10th International Symposium on Voronoi Diagrams in Science and Engineering.* IEEE, 2013, pp. 16–22.

[21] G. Eder, M. Held, S. Jasonarson, P. Mayer, and P. Palfrader, "On generating polygons: Introducing the salzburg database," *International Journal of Computational Geometry & Applications*, vol. 28, no. 4, pp. 309–340, 2019.