Evaluating a d-dimensional Voronoi diagram algorithm and its applicability for linear interpolation

Pro Gradu
University of Turku
Department of Computing
Computer Science, Data Analytics
2024
Anton Nurmiaho

UNIVERSITY OF TURKU
Department of Computing

Anton Nurmiaho: Evaluating a d-dimensional Voronoi diagram algorithm and its applicability for linear interpolation

Pro Gradu, 59 p.
Computer Science, Data Analytics
March 2024

The Voronoi diagram (or graph) and Delaunay triangulation (or tessellation) are important geometric structures for space partitioning. The former is widely used in many fields as it generates patterns containing useful information on the intrinsic structure of the data. The latter generates a simplicial complex of d-simplices, which enables piecewise linear interpolation of scattered data. A simplex is the generalization of a triangle (2-simplex) to any dimension, for example, a three-dimensional simplex is a tetrahedron. Additionally, both structures are dual graphs of each other. This implies that by computing one, the other is also obtained during the process. In lower dimensions, it is possible to generate each structure efficiently by many popular algorithms, but in higher dimensions it becomes increasingly more infeasible as a d-dimensional Voronoi diagram can have $\mathcal{O}(n^{\lceil \frac{d}{2} \rceil})$ vertices and a d-dimensional Delaunay triangulation can have $\mathcal{O}(n^{\lceil \frac{d}{2} \rceil})$ simplices, which entails an alternative approach. The Voronoi graph traversal algorithm proposed by [1] in 2020 as a solution for computing d-dimensional Voronoi diagrams by incremental and stochastic exploration. In this thesis, the Voronoi graph traversal algorithm is surveyed and analyzed with regards to high-dimensional data. Later, components of the algorithm repurposed for piecewise linear interpolation in d-dimensions, which is also examined. The interpolation method searches for the simplex by performing numerous random walks to traverse the Voronoi diagram starting from a Voronoi generator point that is as close as possible to the given query point. When a simplex is found, the values located at its vertices can then be used to estimate the value of the query point. This new method can be applied to machine learning, and it resembles nearest neighbor (KNN) and radial basis function interpolation methods. It was found that the algorithm performed best in contrast to Qhull (a popular algorithm for generating these structures in any dimension) in extremely high dimensions $d > 10$, where computing and exploring a subset of vertices becomes the more practical approach. Conversely, piecewise linear interpolation performed best in dimensions between 2 and 8 since it became gradually more challenging to find simplices in $d > 8$ due to the random nature of the algorithm.

Keywords: Computational Geometry, Voronoi Diagram, Delaunay Triangulation, High-Dimensional Data, Piecewise Linear Interpolation

TURUN YLIOPISTO
Tietotekniikan laitos

ANTON NURMIAHO: Evaluating a d-dimensional Voronoi diagram algorithm and its applicability for linear interpolation

Pro Gradu, 59 s.
Tietojenkäsittelytieteet, Data-analytiikka
Maaliskuu 2024

Voronoi-diagrammi ja Delaunay-kolmiointi (tai tessellaatio) ovat tärkeitä geometrisia rakenteita avaruuden osioinnissa. Voronoi-diagrammia on hyödynnetty laajasti eri aloilla, sillä sen muodostamat alueet sisältävät hyödyllistä tietoa datan rakenteesta. Delaunay-kolmiointi tuottaa kompleksin, joka koostuu toisiinsa liitetyistä d-simplekseistä, jotka mahdollistavat paloittain lineaarisen interpolaation. Simpleksi on kolmion (2-simpleksin) yleistys mihin tahansa ulottuvuuteen, esimerkiksi kolmilotteinen simpleksi on tetraedri. Lisäksi molemmat rakenteet ovat toistensa geometrisia duaaleja. Tämä tarkoittaa, että laskemalla jompikumpi on mahdollista saada myös samalla toinen laskentaprosessin aikana. Matalissa dimensioissa on mahdollista generoida molemmat rakenteet tehokkaasti monilla yleisillä algoritmeilla, mutta korkeissa dimensioissa tästä tulee entistä hankalampaa, sillä d-dimensisellä Voronoin-diagrammissa voi olla $\mathcal{O}(n^{\lceil \frac{d}{2} \rceil})$ solmua ja d-dimensisellä Delaunayn-kolmioinnissa voi olla $\mathcal{O}(n^{\lceil \frac{d}{2} \rceil})$ simpleksejä, mikä edellyttää vaihtoehtoista lähestymistapaa. Vuonna 2020 julkaistussa lähteessä [1] on esitetty algoritmi, jolla voidaan generoida d-ulotteisia Voronoin-diagrammeja hyödyntämällä stokastista prosessia rakenteen vaiheittaiseen tutkimiseen. Tässä tutkielmassa tarkastellaan ja analysoidaan tätä algoritmia korkeadimensioisen datan osalta. Myöhemmin tarkastellaan myös algoritmin osia, jotka on uudelleenkäytetty paloittain lineaariseen interpolointiin d-dimensioissa. Interpolointimenetelmä etsii simpleksin suorittamalla useita satunnaiskulkuja Voronoi-diagrammin läpikäymiseksi Voronoi-generaattorin pisteestä, joka on mahdollisimman lähellä annettua pistettä, jonka arvoa halutaan selvittää. Kun simpleksi on löydetty, sen kärkipisteissä sijaitsevia arvoja voidaan käyttää annetun pisteen interpolointiin. Kyseistä uutta menetelmää voidaan soveltaa koneoppimiseen, ja se muistuttaa lähimmän naapurin ja radiaalisen interpoloinnin menetelmiä. On havaittu, että algoritmi toimi parhaiten Qhull:in (suosittu algoritmi näiden rakenteiden generointiin missä tahansa dimensiossa) verrattuna erittäin korkeissa dimensioissa $d > 10$, jolloin solmujen osajoukon laskeminen ja tutkiminen on käytännöllisempi lähestymistapa. Vastaavasti paloittain lineaarinen interpolointi suoriutui parhaiten 2 - 8 dimension välillä, sillä algoritmin satunnaisuuden seurauksena simpleksien etsiminen vaikeutui dimensioissa $d > 8$.

Asiasanat: Laskennallinen Geometria, Voronoi-Diagrammi, Delaunay-kolmiointi, Korkeaulotteinen Data, Palottain Lineaarinen Interpolaatio

Contents

1 Introduction

Piecewise linear interpolation of high dimensional data is an approach to implement predictors in machine learning, which have been sought after but has thus far been impossible due to high costs in partitioning the data space to such sensible compartments e.g. defined by the Voronoi diagrams. The Voronoi graph traversal algorithm [1] computes Voronoi diagrams and Delaunay triangulations in any dimension. The emphasis of the algorithm as well as the original research paper is on tackling the unique problems that come with high dimensionality, e.g., one being the impractical computation time, and providing a more realistic method for exploring these geometrical structures in high dimensions. To achieve this, instead of computing the complete structures, the algorithm relies heavily on randomness to incrementally discover individual pieces by unsystematically maneuvering throughout the structures during an execution. As more of these pieces are found, the closer the algorithm is to computing the full structure. And when a larger portion of the structures is ever required, then the algorithm can be executed repeatedly to find more and more pieces.

There are three main research aims of this thesis. The first goal is to implement the algorithm described in the research paper; carry out some tests using the algorithm and analyze its various components. The second goal is to give a general overview of the algorithm's foundational theory upon which it was developed and designed. And the final goal is to enable the algorithm to solve piecewise linear

interpolation problems by computing d-dimensional Delaunay triangulations. The Voronoi graph traversal algorithm was implemented in MATLAB and the subsequent testing as well as analysis was also performed in the MATLAB environment.

In Chapter 2, we delve into and give definitions for Voronoi diagrams and Delaunay triangulations. We also list a few popular algorithms for computing these structures. Some of them were intended to only construct 2-dimensional or 3-dimensional Voronoi diagrams or Delaunay triangulation, while others were designed for higher dimensions. Later in the chapter, the curse of dimensionality is briefly explained. This chapter will place emphasis on the usefulness of the Voronoi graph traversal algorithm.

In the first section of Chapter 3, some literature of the methods that the algorithm is based on is reviewed. In the subsequent sections, the algorithm is briefly presented, and its time complexity is calculated.

Chapter 4 is devoted exclusively to testing and analyzing the Voronoi graph traversal algorithm. In this chapter, a comparison is made between the algorithm and another notable algorithm used for computing both structures in any dimension called Qhull. Given that the Voronoi graph traversal algorithm generally computes a smaller portion of the full structure, this raises an interesting question about the number of times it's required for it or its underlying components to be performed to discover a satisfactory amount of the structure. This aspect is analyzed in the final section of this chapter.

In Chapter 5, a general overview of interpolation is presented, then a piecewise linear interpolation method that utilizes the Voronoi graph traversal algorithm is introduced and tested.

The final chapter, Chapter 6, will include a few remarks on the algorithm, a summary of the more notable conclusions made in this thesis, and ideas on how the algorithm can be taken further.

2 Voronoi Diagrams and Delaunay Triangulations

This chapter will serve as a preliminary to all the subsequent chapters. In this section the Voronoi diagram and the Delaunay triangulation will be examined and formal definitions as well as some basic geometric properties for both will be given. Later in the chapter a few prominent algorithms for generating the two structures are listed and compared side-by-side, which will help highlight the computational efficiency and importance of the Voronoi traversal algorithm, particularly in respect to high-dimensional data.

2.1 Voronoi Diagram

The Voronoi diagram (or graph) is an important geometrical concept, not unique to mathematics or computer science, that widely appears in many other academic, or even non-academic, disciplines, in manners that can be unforeseen [2]. This is probably attributable to a few factors: high prevalence of this kind of pattern intrinsically materializing in nature, useful mathematical properties, informative geometrical structure, and constructable as well as visualizable by means of computers [3]. Due to this multidisciplinary nature, it is thus beneficial to research the various properties of Voronoi diagrams and to invent efficient and fast algorithms to generate them. There are many applications of Voronoi diagrams in computer science;

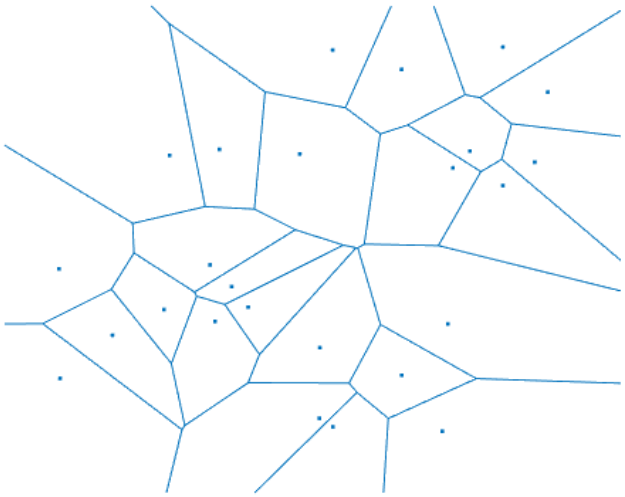

Figure 2.1: Planar Voronoi Diagram

used as graphs, data structures, or as tools for explorative data analysis. Here are a few examples of their possible uses: associative file searching, cluster analysis, scheduling record accesses, collision detection, distance problems, motion planning [3].

Let $S = \{x_1, \ldots, x_n\}$ be a finite set of points in Euclidean space $\mathbb{R}^d$. It is also assumed that the arrangement of the points in S are in general position, meaning that no three points are collinear. The Voronoi diagram is a partition of this Euclidean space into n Voronoi cells (regions) determined by a set of generators also known as sites. Each generator in S will always produce its own Voronoi cell, which arises from perpendicular bisectors of the generator itself and other closest generators (half-space intersections) and is a subset of $\mathbb{R}^d$. In this case, let S be the set of generators. Each point in a Voronoi cell is closer or at least as close to its corresponding generator than any other generator in S. More formally, the Voronoi diagram for the set S is the set of all Voronoi cells $V_i = \{y \in \mathbb{R}^d \mid \|x_i - y\| \leq \|x_j - y\| \mid \text{for all } x_i \in S\}$, and of course evidently $\cup_{i \in S}^{n} V_i = \mathbb{R}^d$. In Euclidean metric each generated Voronoi cell is either a convex polytope or possibly an unbounded convex polyhedron, there-

fore it's possible to use properties of polytopes (or polyhedrons), particularly facets. To define a facet, it is important to explain k-faces first. For example, 0-faces of a d-dimensional are the vertices, 1-faces are the edges connecting the vertices, and so on, until d-face, which is the polytope itself. Then a facet is a $(d-1)$-face of a d-dimensional polytope. Essentially, the whole Voronoi diagram or its individual separable parts such as cells can be constructed by using these combinatorial properties.

For a 2-dimensional Voronoi diagram with $n \geq 3$ generators, the maximum number of vertices is $2n-5$ and the maximum number of edges is $3n-6$, which can be proved using Euler's theorem. When this is generalized to higher dimensions $d > 2$ several properties are preserved such as the convexity of the Voronoi cells, while the linear number of vertices, edges and potentially other facets are lost [4], considering a d-dimensional Voronoi diagram can have at worst $\mathcal{O}(n^{\lceil \frac{d}{2} \rceil})$ Voronoi vertices [5].

Voronoi diagrams have been generalized, and extended for theoretical and practical purposes [3] [4] [2]. The type of Voronoi diagram that was defined above is called the ordinary (nearest-site) Voronoi diagram, and in this thesis, we'll work with this definition moving forward, but there exist many others as well considering they can be constructed under a different set of circumstances. Instead of using the Euclidean metric, alternative distance metrics could be applied to measure the distance between two points e.g., the Manhattan distance or the Chebyshev distance, which produce diagrams with a "grid-like" pattern. For instance, it has been shown that in high-dimensional data mining, it might be preferable to choose some distance metrics over others [6], which could be potentially meaningful for practical applications of Voronoi diagrams. Farthest-point Voronoi diagrams reverse the idea of each point in a Voronoi cell being closer to its corresponding generator, and as the name suggests generators create Voronoi cells of which points are farthest. Suppose

generators were given some numerical weight. These numerical values could then be associated with importance, as more significant generators with a higher weight are more dominant by producing bigger Voronoi cells superseding the distance between points as being the only metric by which to determine the cells. This concept is called the weighted Voronoi diagram. Lastly, a final example, the nth-order Voronoi diagrams, where rather than one generator generating its own Voronoi cell, a set of n-nearest neighbouring generators are responsible instead. All the aforementioned approaches to building these types of geometric structures are just a few extensively researched generalizations of the ordinary Voronoi diagram, but there exists many more.

2.2 Delaunay Triangulation

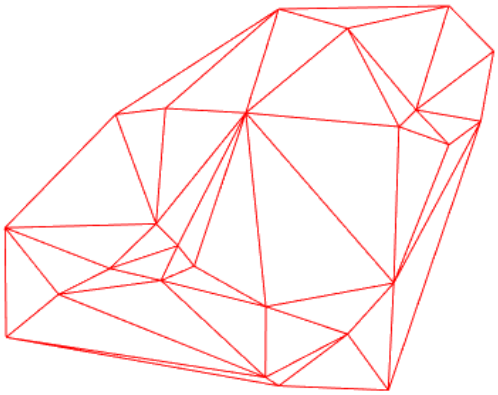

Figure 2.2: Delaunay Triangulation

Another notable geometric structure is the Delaunay triangulation depicted in Figure 2.2, which, just like the Voronoi diagram, has been vastly researched concurrently with it and has a lot of useful geometrical properties as well as applications in

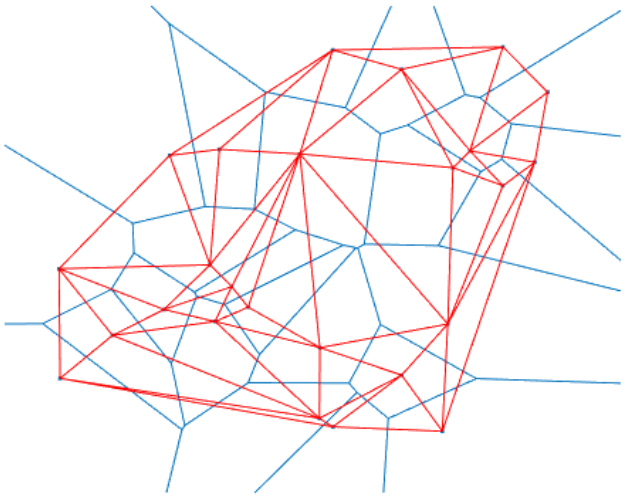

Figure 2.3: Duality

many fields. The Delaunay triangulation can be transformed to represent a graph, and several important and widely used graphs are subgraphs of this very triangulation, such as the relative neighbourhood, the minimum spanning tree, and beta skeletons [3] [4], which are all used in data mining and machine learning. These types of triangulations are also extensively used in representing the terrain of the earth's surface and generating either 2-dimensional or 3-dimensional meshes to solve differential equations [2], for example, in mathematical modelling. In this section the focus will be on the d-dimensional Delaunay triangulation.

The Delaunay triangulation and the Voronoi diagram are both dual graphs, meaning that one can be obtained from the other as there is a one-to-one correspondence between both structures [2] [7]. After computing a Voronoi diagram, a Delaunay triangulation can be constructed by connecting each Voronoi generator to its respective nearest generators that share a common boundary via edges to form a tessellation of nonoverlapping simplices; called, simplicial complex. Before delving into simplices, let's first define a convex hull. Suppose P is a finite set of points, then the convex hull of P is the smallest possible region that encompasses all points

in P. This region is, by definition, a polytope, and typically the boundary of the convex hull is what is computed. Now, a d-dimensional simplex is a simple polytope and a convex hull of a nonempty set of $d+1$ points in $\mathbb{R}^d$. Simplices can also be deconstructed down to its elements i.e., faces or facets just like Voronoi cells, where $(d-1)$-face of a n-simplex is likewise called a $(d-1)$-simplex. A 0-face is a vertex, 1-face is an edge, and so forth. For example, a 2-simplex, which represents a triangle, is combinatorically composed of three 0-faces and three 1-faces. A 2-dimensional Delaunay tessellation is typically called a Delaunay triangulation since the simplices are triangles. In a 3-dimensional environment this geometric structure is specified as a Delaunay tetrahedralization. In higher dimensions a more generalized term is applied, the Delaunay tessellation, but just Delaunay triangulation is also colloquially used to mean any d-dimensional Delaunay tessellation.

Let $S = \{x_1, \ldots, x_n\}$ be a finite set of points in Euclidean space $\mathbb{R}^d$, where $d+1 \leq n < \infty$, and let $CH(S)$ be the convex hull spanning generators in S. It is also assumed that the points in S are in general position i.e., there are no $d+2$ points lying on the same d-sphere and no $k+2$ points lying on the same k-subspace of $\mathbb{R}^d$, for $k < d$. A triangulation of S is where all points $x_i \in S$ is a vertex of some d-simplex, the union of simplices is bounded by $CH(S)$, and the intersection of any pair of simplices is an empty set. This triangulation is a Delaunay triangulation if and only if no points of S lie on the interiors of d-spheres inscribed by d-simplices. While a Voronoi diagram for the set of generators in S will always be unique, a Delaunay triangulation obtained from it will not necessarily be. For instance, the vertices of a quadrilateral formed by a subset of points can be triangulated by two different line segments. This property is easily extended to higher dimensions, where a given set of $d+2$ points in $\mathbb{R}^d$ may be triangulated in at most 2 different ways by d-simplices [8], but if the points are in general position, then the Delaunay triangulation will be unique. Since the Voronoi diagram and Delaunay tessellation are duals of each

other and the algorithm presented in the later sections obtains the Delaunay from the Voronoi diagram, let's define a d-dimensional Delaunay triangulation via the Voronoi diagram. Let $Vor(S)$ be a Voronoi diagram generated by $S = \{p_1, p_2, \ldots, p_n\} \in R^d$, where the generators are in general position, and let $V = \{v_1, v_2, \ldots, v_m\}$ be the set of Voronoi vertices in $Vor(S)$. Furthermore, let $F(p_{i1}), \ldots, F(p_{ik_i})$ be the $(d-1)$-faces sharing v_i, and let T_i be the convex hull spanning generators $p_{i1}, \ldots, p_{ik_i}$. If $k_i = d + 1$ for all $i \in I_m$, then $Del(S) = T_1, \ldots, T_m$ is a set of d-simplices and therefore is the d-dimensional Delaunay triangulation of $CH(S)$ spanning S.

Let S be a set of non-collinear points in $\mathbb{R}^2$ and let k denote the number of points in S that lie on the boundary of the convex hull, then a 2-dimensional Delaunay triangulation has at most $2n - 2 - k$ simplices and $3n - 3 - k$ edges. As in the case with planar Voronoi diagrams, both structures in the plane have a linear size, which can be proved using Euler's theorem. Similarly, because of the dual nature, the worst-case for the number of simplices for a d-dimensional Delaunay triangulation is $\mathcal{O}(n^{\lceil \frac{d}{2} \rceil})$, which corresponds with Voronoi vertices.

2.3 Algorithms

Numerous approaches to generate Voronoi diagrams and Delaunay triangulations exist either directly or indirectly through one's respective dual. In this section a brief description will be presented of a few of the most popular algorithms for Voronoi diagrams and Delaunay triangulations in any dimension, but the centre of attention will still be on higher dimensions. These algorithms have been categorized into 5 main groups [9]: incremental, divide and conquer, sweep-line, gift wrapping, and lift-up transformation. It's also good to say at this point that while all algorithms were designed to compute in any dimension, the unfortunate reality is that in truly high dimensions all these algorithms suffer with slow computation runtimes and high space complexity. This is known as the "curse of high dimensionality", which will

be discussed more in section 3. Higher dimensional data requires either discovering methods for reducing the dimensionality or other novel solutions.

2.3.1 Incremental

Incremental approach is the most conceptually straightforward, where the method starts with only a small number of generators in order to construct a simple structure first. After this more generators are iteratively added one by one; all the while updating the structure each time to make sure the structure satisfies either the Voronoi diagram or Delaunay triangulation criterion. One early example of such incremental algorithm is the Bowyer-Watson algorithm [10] [11], which was designed to compute any d-dimensional Delaunay triangulations mainly in Euclidean space but has a comparatively slow computation complexity in higher dimensions, where the running time can certainly reach $\mathcal{O}(n^{\lceil \frac{d}{2} \rceil})$.

2.3.2 Divide and Conquer

The fundamental design of a divide-and-conquer algorithm is recursively partitioning the original problem into smaller generally equal sub problems until a stopping condition is reached. This is subsequently followed by a process of merging each sub problem starting from the lowest depth of recursion. Let $S = \{x_1, \ldots, x_n\}$ be a set of generator points that are sorted in ascending order with respect to any coordinate axis and let's denote $DC(X)$ as the appropriate divide-and-conquer algorithm, where X is the input of a list of points. If $n \leq 3$, then a Voronoi diagram is computed directly and returned, otherwise S is divided into $S_L = \{x_1, \ldots, x_j\}$ and $S_R = \{x_{j+1}, \ldots, x_n\}$, where $j = 2/n$ and $j < n$. For a Delaunay triangulation the input must be $n \geq 3$, if $n = 3$, then there is only one simplex, otherwise proceed on with the same exact procedure as in the Voronoi diagram case. Afterwards a Voronoi diagram or Delaunay triangulation is computed separately for both parti-

tions S_L and S_R by recursively calling the algorithm again in the following manner: $DC(S_L)$ and $DC(S_R)$. The final step is the merging of S_L and S_R together to form a Voronoi diagram or Delaunay triangulation and then returning it. Some consideration for implementing a valid merging procedure is required to join all the earlier sub problems and the final "left" and "right" substructures.

The DeWall algorithm [12] is based on the divide-and-conquer technique for constructing Delaunay triangulations generalized to an arbitrary d-dimensional space. This algorithm uses a more efficient method for dividing the set of points in higher dimensions, which helps with subsequent merging. During the splitting phase the space is divided by a hyperplane into three disjoined subsets: the simplices intersected by the hyperplane called the "simplex wall", simplices on the left side of the simplex wall, and simplices on the right side of the simplex wall. The simplex wall is built by using an incremental construction approach removing the need to modify the triangulation. The first d-simplex is formed by triangulating the nearest points to the hyperplane in the following manner. Select the closest point to the hyperplane, let's denote it by p_1. Then from the other side of the hyperplane select the point that is closest to p_1, denoted by p_2. Both p_1 and p_2 now form a 1-face of the d-simplex, it's now possible to build a triangulation by joining the 1-face with some third point p_3. p_3 is determined by selecting the smallest possible 2-simplex (triangle) in terms of its radius, which produces a 2-face with points p_1, p_2, and p_3. This procedure is then repeated until a full d-simplex is generated. Afterwards other adjacent simplices are constructed in accordance with the Delaunay criterion.

In theory the worst-case time complexity for DeWall is $\mathcal{O}(n^{\lceil \frac{d}{2} \rceil + 1})$, but in practice the algorithm is quite efficient, especially in 3-dimensional space.

2.3.3 Gift Wrapping

Gift wrapping is an incremental algorithm to construct Delaunay triangulations by starting with an edge and then finding a point with which the edge forms a triangle. The triangle with the smallest circumcircle is selected and simultaneously creates a simplex of a Delaunay triangulation. When this is generalized to higher dimensions, then for each k-simplex a point is selected that forms the smallest possible circumcircle for the $k+1$-simplex. In order to not have to exhaustively check each possible point, it becomes critical to find a smaller search space, thus requiring optimal data structures.

Dwyer has proposed an algorithm that incorporates this idea and has also also showed that if all points are uniformly distributed within a unit d-ball, then the time complexity of this procedure is linear $\mathcal{O}(n)$ [13].

2.3.4 Sweep Line

The sweep line technique is computationally efficient for building Voronoi diagrams, and indirectly Delaunay triangulations. One notable algorithm of this kind is Fortune's algorithm [14] , which builds planar Voronoi diagrams and has an asymptotic bound of $\mathcal{O}(n \log n)$. This method creates either a horizontal or vertical line on the plane, which constitutes a boundary that is then "swept" from one end of the plane to the other. During this the algorithm encounters each generator point one by one in discrete steps and updates the data structures that hold the information about the combinational structure of the Voronoi diagram. This specific algorithm, of course, doesn't work in higher dimensions, but that is not the only thing to consider, as the result is not a direct Voronoi diagram but a geometric transformation of it.

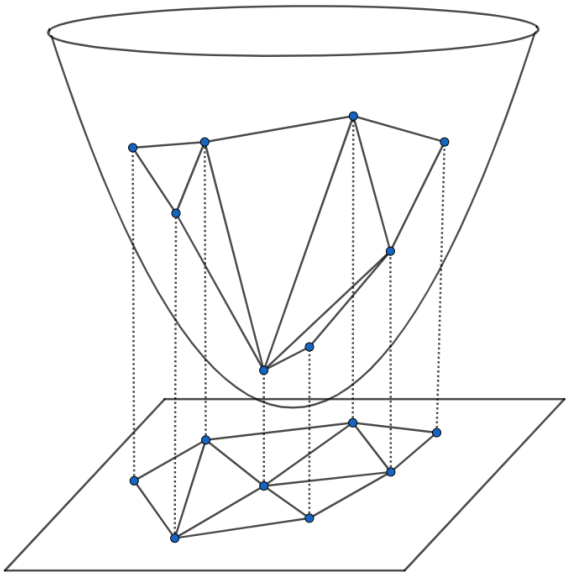

Figure 2.4: Lifting Transform

2.3.5 Lifting Transform (High-Dimensional Embedding)

There is an interesting connection between both Voronoi diagrams and Delaunay triangulations in d-dimensional space and convex hulls in $d+1$ dimensional space. which was initially discovered by Brown [15] for Voronoi diagrams. The first step is to "lift" or to transform the generator points in dimension d onto the surface of a paraboloid in dimension $d+1$. For example, a lifting function for points in 2-dimensional space is denoted by $f : \mathbb{R}^2 \rightarrow \mathbb{R}^3; (x,\ y) \rightarrow (x,\ y,\ x^2 + y^2)$. Subsequentially, a convex hull of the generator points is constructed by any applicable convex hull algorithm in the $(d+1)$-dimensional space.

There also exists a hyperplane that divides the $(d+1)$-dimensional space splitting the generator points. The points below this hyperplane form a lower d-face, and correspondingly the points above form an upper d-face. In the final step a part of the convex hull i.e., d-face is projected back onto the original dimension d. For the nearest-site Voronoi diagram and the Delaunay triangulation, the lower d-face is

projected, whereas for the farthest-site Voronoi diagram the upper d-face is projected [2].

The computation of the convex hull in $(d + 1)$-dimensional space is the step that is the most expensive. Therefore, finding an efficient algorithm for computing convex hulls in higher dimensions is crucial. One such robust algorithm is the Quickhull algorithm [16], which has, if the precision of the input is constricted to $\mathcal{O}(\log n)$, a conjectured worst-case complexity of $\mathcal{O}(n \log v)$, where n is the number of input points in d-dimensional space and v is the number of processed points. The Qhull code package [17] utilizes the Quickhull algorithm in combination with the aforementioned lifting transformation to compute nearest-site and farthest-site Voronoi diagrams and Delaunay triangulations. The more effective use of Qhull is still with input of up to 8-dimensions due to the running time growing rapidly with respect to dimension as computing a 16-dimensional convex hull with 1000 points will result in approximately 10^{24} facets.

3 Voronoi Graph Traversal

At the end of the previous chapter, it became quite evident that alternative approaches are required to deal with high-dimensional Voronoi diagrams and Delaunay triangulations. In this chapter, a brief examination of the behavior of points in high dimensions is presented and the effects of high-dimensional data, specifically as it pertains to the computation of the geometric structures mentioned above, is examined in Section 3.1, and some preliminaries are presented in Section 3.2. On the subsequent sections the center of attention is the Voronoi graph traversal algorithm put forward by Polianskii & Pokornoi [1]. A condensed description of the algorithm and a quick overview of its most significant components is presented in Section 3.3. And lastly in Section 3.4 the algorithm's time and space complexity is evaluated. In chapter 4 the Voronoi graph traversal algorithm, which was implemented in MATLAB for the purpose of this thesis, is tested with synthetic data.

3.1 High-Dimensionality

With efficient modern computers, a lot more data is constantly being collected than ever before. It almost seems like data is collected just for the sake of it, since it can be conveniently collected and stored without much difficulty. But large amounts of data with a lot of features, which are numerically represented as points, bring a new set of problems when exploratory data analysis is performed on the dataset or used in mathematical modelling or machine learning. High dimensional problems

compared to low-dimensional problems are obviously generally more computationally expensive. Data with a lot of features causes a combinatorial explosion and the space required to store the computed results as well as the input can easily reach the maximum memory limit. Additionally, there is another inherent problem with high-dimensional data called the curse of dimensionality, which conveys many definitions depending on the type of problem in statistics or computer science, but generally refers to the fact that as more dimensions are incrementally added, the natural metrics available cannot distinguish well between proximity and non-proximity of points, wherein points become inherently sparsely populated as opposed to in low dimensions.

The computational complexity of nearest neighbor searches, which is the calculation of distance to measure similarity or dissimilarity between pairs of points, becomes increasingly more expensive in high dimensions, requiring more efficient nearest neighbor search algorithms. Many factors affect the performance of nearest neighbor searches such as: data dimensionality, number of features, structure of the data, and the search precision, with data dimensionality having one of the biggest impacts [18]. Numerous algorithms have been designed, researched, analyzed, and compared side by side [18]–[20]. It has also been proposed to precalculate a solution space before performing nearest neighbor searches e.g., Voronoi cells, which when stored as an index structure enable faster search times in high-dimensional space [20]. Since finding exact matches requires brute forcing through each candidate point, exact nearest neighbor searches become increasingly more infeasible when the sample size or dimensionality of the data is high. Therefore, the reduction of the search space or the use of ϵ-approximate nearest neighbor searches is preferred. K-d tree and ball tree, which are popular tree-based data structures, are commonly used to reduce the search space of a query. While the k-d tree is an extremely efficient alternative, it is also vulnerable to the "curse of dimensionality" and becomes

just as bad as brute forcing when $d > 20$ [21]. In this case ball trees can be used, which out-perform k-d trees in higher dimensions but have a major drawback of potentially taking a longer time to construct itself. If finding an exact match isn't necessary, then using ϵ-approximate nearest neighbor searches to find a close enough match for a query point is also a good approach.

High-dimensional data does not, in practice, manifest as truly high-dimensional, as it can often be distributed on a lower-dimensional manifold. Let $S = \{p_1, p_2, \ldots, x_n\}$ be a finite set of points in $\mathbb{R}^D$ and let's assume that S lies on a d-dimensional manifold M embedded in Euclidean space $\mathbb{R}^D : M \in \mathbb{R}^D$. As a result, D is then called the extrinsic dimension of S and d the local intrinsic dimension of S [22]. For finite set of points, there can simultaneously be many lower dimensional manifolds, for example, a 1-dimensional manifold can exist as a continuous line segment passing through each point in a 2-dimensional space. The main challenge is the discovery of a manifold that retains as much of the original information as possible. By separating the space in this manner, some useful explicit mapping of a lower intrinsic dimension is then extracted and computed directly completely ignoring the extrinsic dimension $D - d$.

Dimension reduction techniques are used to combat the sets of problems that are brought by high dimensionality. One such reduction method, which is based on the idea above, is the projection of n points from high-dimensional space onto a lower-dimensional subspace when it's assumed that the points exist on some lower-dimensional manifold. The Johnson-Lindestrauss lemma states that if a set of points are projected onto $\mathcal{O}(\frac{\log n}{\epsilon^2})$-dimensional Euclidean space, then the distance between the points is almost fully preserved with a minor amount of distortion by a factor of $(1 \pm \epsilon)$, for any $0 < \epsilon < 1$ [23]. It is also quite likely that high-dimensional data has redundant features that do not offer necessary information, in which case feature selection is performed.

3.2 Preliminaries

Before introducing the Voronoi graph traversal algorithm, it's beneficial to be acquainted with a few more methods, which establish the basis of the algorithm. These include ray casting, the Monte Carlo method, Markov chains, and random walks.

3.2.1 Ray Casting

Ray casting is one of the earliest and most rudimentary rendering techniques in computer graphics and serves as an essential foundation for other more sophisticated rendering algorithms such as ray tracing. Ray tracing is a rendering method, which Andrew S. Glassner defines as "a technique for creating a 2-dimensional picture in a 3-dimensional world" [24]. This definition also applies well to ray casting and rendering in general. The first ray casting algorithm which was utilized for rendering images was introduced in 1968 by Arthul Appel [25].

To render a 3-dimensional object, multiple straight lines in a 3-dimensional space called rays are cast from an observational point called the eye (or camera). A ray has a direction and a distance and can additionally be defined as a unit vector. Between the eye and the object, there exists a plane onto which the resulting image of the object is rendered called the image plane. This plane is a rectangular grid graph, where each grid represents a single pixel. The object, rays, and the image plane exist in a 3-dimensional space referred to as the scene. Each ray passes the image plane through one of its grids towards the scene where the object is located. The next step is to calculate the point of intersection that occurs if an object is hit by a ray. If there is more than one object placed on the scene, then the ray hits the closest one blocking its trajectory. By calculating the intersection along the ray, the color of the pixel can then be determined.

A ray is defined as a set of points in a continuous function $r(t) = r_s + r_d t$, where $t > 0$. As for the variables in the equation of the ray: r_s is the starting point

(or initial point), r_d is the direction (or terminal point), and t is the length of the distance of the ray from the starting point to an intersection at which it collides with something.

3.2.2 Monte Carlo Method

The Monte Carlo method solves a deterministic system in a nondeterministic manner by repeatedly sampling random variables from a probability distribution. Repetition can be performed by either continuously drawing more random samples during a single simulation, increasing its runtime, or running the simulation multiple times consecutively. The goal is to utilize and rely on randomness to find a solution to a complex problem or analyze the behavior of a system.

One conventional use, which also serves as a good example, is in numerical integration, where the ratio of accepted points is used to calculate an approximated definite integral. Independent and identically distributed points are generated over an interval. Each point is then either accepted or rejected depending on its exact location. If it lies inside the area of the function, then it's accepted; otherwise, it lies outside and thus is rejected. As more points are generated, the ratio of accepted points starts to converge towards the expected value of the integral, due to the law of large numbers. Some identifiable reasons as to why the Monte Carlo method is used: efficiency, ease of implementation, insight into the structure of a system, and strong theoretical foundation [26].

3.2.3 Markov Chains

[27] is used as a reference for this subsection. A Markov chain is a sequence, represented as an index set, of random variables which correspond to states. Each state depends only on the previous state and provides information on the likelihood of a future state. A Markov chain creates a system, of which the value readjusts

each time a transition occurs from one state to another. The transition is based on some probability distribution expressed by the current state. A Markov chain is a type of stochastic process that satisfies the Markov property, which asserts that a future state is contingent only on the state before a transition happens. Typically, a stochastic process is measured in time, which can be discrete or continuous. Let's denote a stochastic process as $X(t) : t \in T$, where T is the index set. A discrete-time process is when the index set $T = 0, 1, 2, \ldots$, while a continuous-time process is when $T = [0, \infty)$. A state space, which is a set of all possible values of a Markov chain over some time interval of the process, can likewise be either discrete or continuous. The term Markov chain is generally used when the process is discrete-time, while the term Markov process conveys the meaning of a continuous-time process.

A stochastic process X is a Markov chain, if it satisfied the Markov property:

$$P(X_{n+1} = x | X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = P(X_{n+1} | X_n = x_n)$$

for all $n > 1$ and all states, $x_1, \ldots, x_n \in S$.

Since a transition from state i to j occurs with some probability p_{ij}, a matrix can be used to contain the probabilities of all possible pairs. A transition matrix:

$$\begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1j} \\ p_{21} & p_{22} & \cdots & p_{2j} \\ \vdots & \vdots & & \vdots \\ p_{i1} & p_{i2} & \cdots & p_{ij} \end{bmatrix}$$

$$p(i, j) = P(X_j | X_n = i), \text{where } p_{ij} >= 0$$

A Markov chain does not have to depend on time and can instead be time-homogeneous, where

$$P(X_n + 1 = x | X_n = y) = P(X_n = x | X_n - 1 = y)$$

for all n > 1.

3.2.4 Random Walks

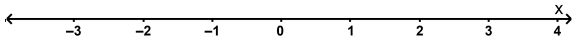

Figure 3.1: One Dimensional Lattice

[28] Consider a one-dimensional integer lattice $Z^1 = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ where the distance between any two neighboring integer points is 1. Let's perform a random walk on the lattice with n steps starting from the origin. In this case, the length of a singular step is the change of location on the lattice by an increment of ± 1. Hence, the walker's first step will be towards an adjacent point, either 1 or -1, based on some probability given to us by being on the origin. Let's also assume that the probability of taking a step towards any direction is uniform over the whole lattice. Therefore, the probability of the first step, or rather any step, towards a neighboring point is 0.5. Let's denote a step from an integer point i by Z_i, for which

$$Z_i = \begin{cases} +1, \text{with a probability of } 0.5 \\ -1, \text{with a probability of } 0.5 \end{cases}$$

Then, the final position of our random walk is the sum of all n steps $S_n = \sum_{i=1}^{n} Z_i$.

Markov chains can be thought of as random walks, but not all random walks are Markov chains. A random walk can be defined as a time-homogeneous Markov chain if it satisfies the Markov property, and then analyzed further by using the underlying theory of Markov chains. Let S_n denote the stochastic process of a random walk on Z^1, and let $Z_1, Z_2, \ldots$ be i.i.d random variables. Consequently, the random walk is defined by $S_n = \sum_{i=1}^{n} Z_i$, with $S_0 = 0$. Now, the set of integers on the one-dimensional lattice make up the whole state space of our random walk, and the probability of transitioning from state i to an accessible state is $p_{i+1} = p_{i-1} = 0.5$.

An accessible state j is any state whose transition probability from state i is $p_{ij} > 0$.

Random walks can also be deployed to explore graphs [29]. A grid-like structure like the lattice, used in the example above, can easily be redefined as an undirected graph $G = (V, E)$, where each vertex $v \in V$ corresponds to a coordinate point on the lattice and each edge $e \in E$ indicates which pairs of vertices are connected to each other. Now, if we consider a random walk on this graph, then we start on some vertex v_0 and move to another vertex v_i with some initial probability p_{0i} associated with the edge $v_0, v_i \in E$. Obviously, V can be thought of as a stochastic process $v_i : i = 0, 1, 2, \ldots$, and therefore a Markov chain whose state space is generated by V. Initiating a random walk from the vertex v_i alludes to triggering a transition from state i to some other state j in the Markov chain with some probability taken from the probability distribution P_i. All transition probabilities can thus be represented in a matrix.

When a random walk is performed on higher dimensional lattices or graphs with many edges or vertices, there is no guarantee that it does not revisit the same location/vertex multiple times during the full walk. In fact, a random walk on Z^1 will converge towards the starting location, and in lower dimensions the random walk will generally stay within proximity to the starting location. Pólya [30] proved that random walks are recurrent in Z^1 and Z^2, which is to say the probability of a random walk returning back to its starting position is 1, and transient in $d > 3$, meaning the probability of a random walk returning back to its starting point is less than 1. For $d = 3$ the probability was estimated numerically to be ~ 0.34 [31]. If this behavior is undesired, then a self-avoiding random walk can be utilized instead. And as the name suggests, this type of random walk will avoid visited vertices and take an alternative path. In the case of graphs, the probability of taking a step towards an already visited vertex decreases, as more vertices and edges are added.

Some important parameters of a random walk are hitting time and commute

time [32]. Hitting time is the expected time of a random walk reaching a requested location. Commute time is the expected time of a random walk coming back from the requested location to the starting location with hitting time added. The number of steps can also be used as a measurement instead of time.

3.3 Overview of the Voronoi Graph Traversal Algorithm

In situations where dimension reduction is either unavailable or inefficient, there is a need for other strategies to address the problem of high dimensionality. Polianskii & Pokornoi proposed an algorithm for computing a Voronoi diagram in any dimension [1], which in this thesis will be referred to as the Voronoi graph traversal algorithm. For a more detailed description of the method as well as the pseudocode can be found in the original research paper.

The general idea of the algorithm is to compute smaller sections of the full Voronoi diagram instead of directly constructing it completely. This is done by finding a subset of Voronoi vertices by traversing through the diagram going from a higher-dimensional facet to a lower one until a 0-face (vertex) is reached. By combining the discovered vertices, all the other $(d-1)$-facets can also be constructed, which then materialize into a skeleton of the Voronoi diagram, from which a Delaunay triangulation can then be obtained. The worst-case time and space complexity of $\mathcal{O}(n)$ is therefore avoided when the focus is only on a portion of the Voronoi diagram, but it's still possible to reach $\mathcal{O}(n)$ by running the algorithm numerous times and storing the discovered facets during each iteration if the goal is to find the complete diagram.

In the following steps, the most important aspects of the Voronoi graph traversal algorithm are discussed: ray casting, intersection search, random walk, and the

obtainment of a Delaunay triangulation. The algorithm starts with generating a unit vector based on random sampling, and then directing a ray towards the direction of the vector. If the ray intersects the boundary of a Voronoi face, another randomized ray is then cast within the newly found face at the point of intersection. We'll continuously descend towards lower and lower dimensional faces. After a sufficient number of vertices have been discovered, the last step is the construction of the Delaunay triangulation.

Let's assume the data is represented as a set of finite points $S = \{x_1, \ldots, x_n\}$ in Euclidean space $\mathbb{R}^d$. We also assume that $n \geq 3$ and the points are in general position. Each point in S is a Voronoi generator and S generates the d-face of the Voronoi diagram. For each generator $x \in S$, a list $G_{x_i} \subset S$ is initialized to contain the generator x_i, $G_{x_i} = \{x_i\}$, where $\forall i \in \{1, \ldots, n\}$ and $x_i \in S$.

3.3.1 Ray Caster

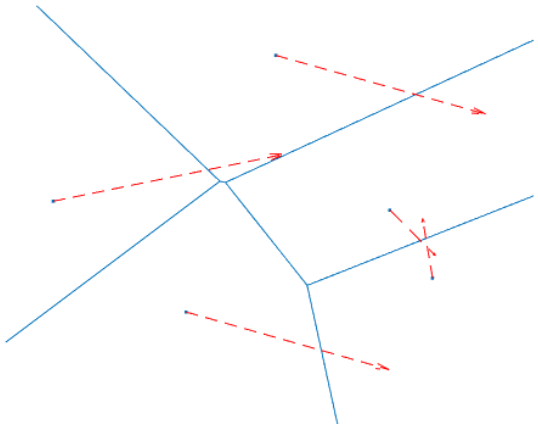

Figure 3.2: Ray Casting

To find a subset of $(d-1)$-faces, for a given reference point q, a uniformly distributed point $w \sim U(B(q,1))$, is generated on a d-sphere $B(q,1) = \{w \in \mathbb{R}^d \mid$

$\|w - q\| \leq 1\}$ to create a normalized direction pointing vector from q to its corresponding random point $\hat{u} = \dfrac{p_i - x_i}{\|p_i - x_i\|}$.

Consequently, a ray is then cast towards the direction of the unit vector $\hat{u}$ in the following manner $r(c) = q + c\hat{u}$, where $c \geq 0$ and c is the distance of the ray. In lower $(d-k)$-faces, where $d \geq k > 0$, the uniform random sampling is orthogonally projected onto the subspace of $(d-k)$-face, which then points towards some boundary of a $(d-k-1)$-face.

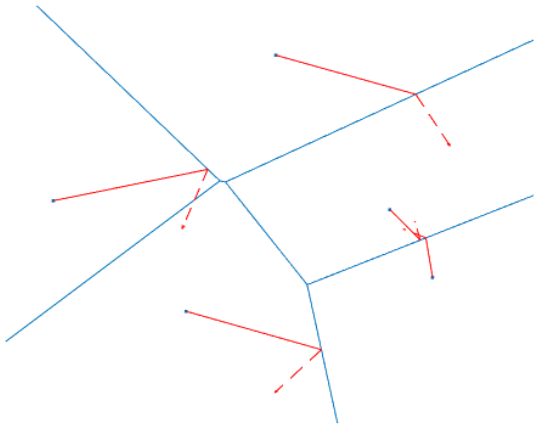

Figure 3.3: Random Ray Sampling

Before attempting to find the first $(d-1)$-face, at first, q will be the only generator in G_{x_i}, but as the algorithm travels to lower subspaces $k > 1$, the number of generators in G_{x_i} will increase and the reference point q will be some point located on the $(d-k)$-face generated by G_{x_i}.

3.3.2 Intersection Search

After ray casting, the ray will either strike the boundary of a lower-dimensional face or miss it completely as it travels infinitely inside an unbounded Voronoi cell. If it is the latter, then the direction of the ray is reversed at which point it is confirmed to hit a boundary component with a probability of 1 for S. In lower faces, there is also

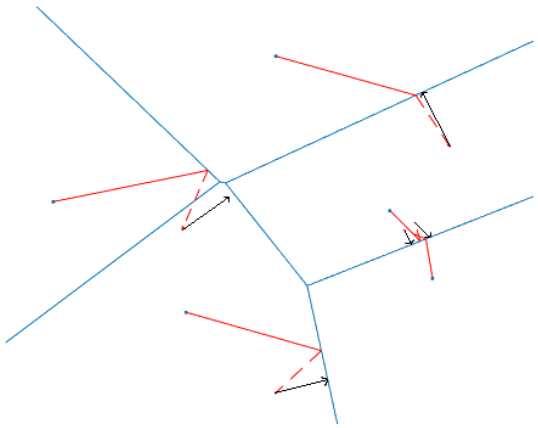

Figure 3.4: Orthogonal Projection

a possibility of the ray travelling towards the infinite direction of a boundary of the face. Likewise, in this case, the vector will be inverted to travel along the other side. The point of intersection $t = p + c\vec{u}$ is then calculated by determining the distance c by either a brute force method or performing an intersection search. Using the brute force method, the algorithm calculates the point of intersection for each generator in $S - G_{x_i}$, the intersection occurs when $f(c) = \|p + c\vec{u} - x_1\| - \|p + c\vec{u} - x_2\|$, with $x_1 \in G_{x_i}$ and $x_2 \notin G_{x_i}$, reaches a minimum $c^* = \min_{\mathbf{c}} f(c)$, where the smallest non-negative distance c is selected. It is known that the generators for the nearest boundary that the ray intersects will also be the ones that are nearest. Instead of computing all possible combinations one by one, we'll only consider the nearest generators via a nearest-neighbor search with respect to the distance parameter of the ray c. The generator responsible for the boundary of the half-space intersection is added to G_{x_i} and the determined intersection will constitute as a new reference point from which another ray is cast, so in a way we retract back to step 1 and carry on with the process.

By continuously executing steps 1 and 2 for each generator in S and then descending all the way until corresponding Voronoi vertex is reached, a partial Voronoi

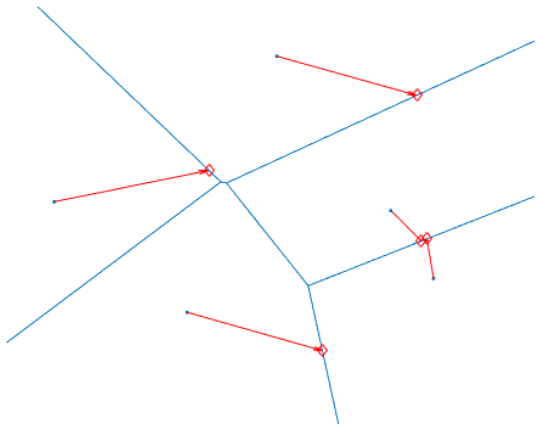

Figure 3.5: Initial Intersections

diagram can then be constructed with a subset of vertices that build up the structure. It is also possible to find the full Voronoi diagram with this method if the number of Voronoi generators is small enough, but in practice it will, most likely, be a d-skeleton. To find an even bigger subset of Voronoi vertices, the number of rays being cast can be increased. But instead, by using the knowledge of the generator set G_{x_i} for an arbitrary Voronoi vertex, other adjacent vertices can be discovered by directing a ray along other edges that connect to the vertex. This procedure is called random walk and it's done by selecting a random subset of G_{x_i} with d generators and then performing steps 1 and 2 for this subset.

3.3.3 Performing Random Walks and Obtaining Delaunay Triangulations

A Voronoi vertex v is an intersection generated by the set G_{x_i}. It is known that G_{x_i} has $d+1$ generators and it is also known that for each vertex $v \in S$, there is $d+1$ number of edges that connect v to other adjacent vertices. The information required for the edges is already provided in G_{x_i} except for their lengths. In result, the idea

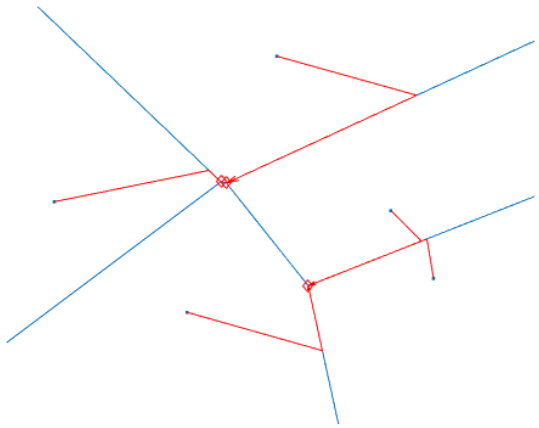

Figure 3.6: Final Intersections

is to utilize v as the reference point from which a ray is cast, where the direction will be randomly chosen by $G_{x_i} \setminus x_{any}$, where x_{any} is any generator from G_{x_i}. The point of intersection and the new generator is then determined by intersection search, where the discovered point is a Voronoi vertex v_{new}. Random walks can be done to incrementally explore an even bigger subset of the Voronoi diagram further away from v by casting a new ray from v_{new} towards any other vertex that isn't v and repeating this process.

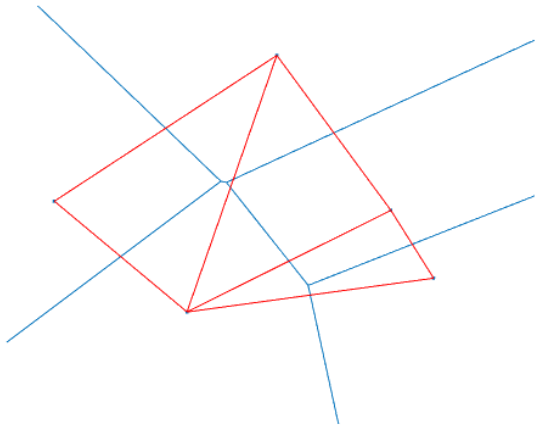

Figure 3.7: Delaunay Triangulation

A partial Delaunay triangulation can be obtained after a subset of Voronoi vertices have been computed. The generator sets stored during the runtime of the algorithm also contain information about the subset of Delaunay simplices, which are constructed by connecting the generators in S via edges.

3.4 Voronoi Graph Traversal Complexity

Given a dataset denoted as a set of finite points in general position $S = \{x_1, \ldots, x_n\}$ in Euclidean space $\mathbb{R}^d$, $n > 3$, what is the time and space complexity of the algorithm for input S? We know that ray casting and intersection search will initially be executed for each generator $x \in S$, where, as the algorithm descents towards 0-faces from d down to 0, both subroutines will be repeatedly called d times for all x. A data structure storing n lists will also be initialized and after termination each list will store $d+1$ generators and a single Voronoi vertex. The ray casting and intersection search will therefore be the most crucial components in terms of affecting computation efficiency. Later, the random walks will be examined separately, as they are performed after the computation of a subset of Voronoi vertices finishes.

Ray casting includes two steps to create a vector: random ray sampling and orthogonal projection. For random ray sampling, a uniformly distributed point on a d-sphere is sampled. One method is to generate $d+1$ normally distributed variables $\{p_1, p_2, \ldots, p_{d+1}\} \sim N(0,1)$ and then compute a unit vector $\hat{v} = \frac{(p_1, \ldots, p_{d+1})}{\sqrt{\sum_{i=1}^{d+1} p_i^2}}$ to obtain an independent uniformly distributed point v on the d-sphere [33]. This method is straightforward to implement and generalizes to any dimension. It takes $\mathcal{O}(d)$ to generate $d+1$ normally distributed variables assuming they can be generated in constant time. The calculation of the l^2-norm $\sqrt{\sum_{i=1}^{d+1} p_i^2}$ and the division operation are both done in $\mathcal{O}(d)$ time. An orthonormal basis spanning the $(d-k)$-face in $\mathbb{R}^d$, $k \leq d$, is required before $\hat{v}$ is orthogonally projected onto it. The Gram-Schmidt

orthonormalization process can then be implemented algorithmically to orthonormalize vectors of the generators $\{x_1, \ldots, x_{k+1}\} \subset S$ of the $(d-k)$-face, which has a $\mathcal{O}(dk^2)$ runtime. The vector $\hat{v}$ is projected onto each vector in the basis in $\mathcal{O}(k)$.

A brute force method or intersection search is utilized to determine the distance of the vector and the intersection. During the brute force approach, the distance from the reference point is calculated to the boundary of a Voronoi face generated by each $x_i \in S - G$. S is self-evidently the global set of generators and G is a subset of generators $G - S$, which contains the generators of faces the algorithm has already visited. Since the maximum number of generators in G, before a Voronoi vertex is reached, is d, $S - G$ contains at least $n - d$ generators, and therefore $S - G = \{x_1, \ldots, x_n - k\}$, where $k = 1, \ldots, d-1$, d with regard to the $(d-k)$-face the ray is being cast in the direction of. Consequently, brute forcing necessitates $(n-k) \times d$ operations for a single point $x \in G$, hence the worst-case time complexity for this approach becomes $\mathcal{O}(nd)$ and for all n it would be $\mathcal{O}(dn^2)$. This is quite a slow computation time, and the more efficient and realistic approach is using intersection search instead.

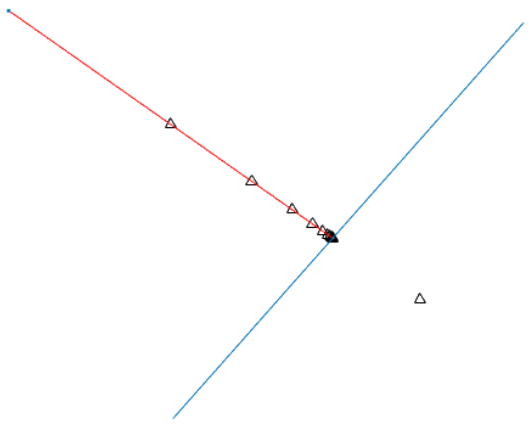

Figure 3.8: Intersection Search with Nearest-Neighbor Searches

The two determining factors of intersection search are choosing the distances of

the ray at which the nearest-neighbor search should occur and the nearest-neighbor search itself. As for the former, a ray can be cast to a certain terminal point p_{end} from an initial point p_{start}. Another point is initialized by dividing the distance between p_{end} and p_{start} by 2, $p_{\text{mid}} = \frac{p_{\text{start}} + p_{\text{end}}}{2}$. If at p_{end} the nearest-neighboring generator $x \notin G$, then the point of intersection between the boundary of a Voronoi face generated by x and the ray cast from a higher dimensional k-face is calculated and p_{end} is assigned to it. If the nearest-neighboring generator x is in G, then $p_{\text{start}} = p_{\text{mid}}$. This procedure is repeated until the distance between p_{start} and p_{mid} converges sufficiently small, $\sqrt{\|(p_{\text{end}} - p_{\text{start}})^2\|} < \epsilon, \epsilon > 0$. The time complexity of this is $\mathcal{O}(\log \frac{r_{length}}{\epsilon})$ and depends solely on the maximum distance of the ray r_{length} and precision ϵ. Concerning the nearest neighbor search, time complexity grows with respect to dimensions as previously briefly mentioned in Section 3.1, the selection of an appropriate algorithm has a considerable role in terms of efficiency. Hence, $\mathcal{O}(\log \frac{r_{length}}{\epsilon} \cdot NN(\cdot))$ is the time complexity of intersection search, where $NN(\cdot)$ is the cost of a nearest-neighbor search.

Figure 3.8 portrays the process of intersection search with nearest-neighbor lookup, where p_{mid} is represented as a black triangle after each iteration as the length of the ray between p_{start} and p_{end} converges. The initial value of p_{mid} goes over the boundary and is the rightmost black triangle in the figure. Now its closest generator is not the one from which the ray was cast, thus p_{end} is set to the same value as p_{mid}. p_{mid} is then set to the midpoint of a new line with its updated endpoints p_{start} and p_{end}. The next p_{mid} value is the leftmost black triangle in the figure and is located closest to the generator from which the ray was cast, thus p_{start} is set to the value of p_{mid} and p_{mid} is updated again. This process is repeated, and from the figure we can see how each updated p_{mid} value is continuously located closer and closer toward the boundary.

A single random walk consists of ray casting and intersection search, so its total

runtime is dependent on both. In order to retrieve a Delaunay triangulation, d-simplices are constructed by adding $\binom{n+1}{2}$– edges to each $i \in G_i$, where i comprises of $d+1$ generators. As a result, this requires $\mathcal{O}(i * \binom{n+1}{2})$ operations.

4 VGT Algorithm Validation & Performance

In the previous chapter, we went through an overview of the VGT algorithm and calculated its time complexity. This chapter will delve into the expected behavior and the performance of the VGT algorithm with respect to input. The algorithm has been implemented in MATLAB as well as the testing environment.

In Section 4.1, the runtime of the VGT algorithm is compared to Qhull, where both will be tested to process synthetic data of differing sample sizes and number of features (dimensions). Since the VGT algorithm computes a subset of vertices through repeated random sampling, Section 4.2 will cover the number of times the algorithm needs to be executed in order to potentially discover an even higher number of vertices and how this pertains to eventually constructing a complete Voronoi graph.

4.1 VGT vs. Qhull

The quickhull algorithm, which was briefly discussed in Section 2.3.5, is an efficient approach for computing convex hulls, Voronoi diagrams and Delaunay triangulations for high dimensional data. In practice, however, the algorithm operates ideally with dimensions between 2 and 10, as it starts to become computationally and memory-wise more expensive to calculate and store all the facets or simplices in higher

dimensions. The core idea of the VGT algorithm is not to immediately calculate all possible facets and simplices, but rather find them incrementally by performing random walks on the Voronoi graph to discover an even greater number of Voronoi vertices starting from some initial smaller subset of vertices. Many algorithms, such as the quickhull algorithm, compute the full Voronoi graph or Delaunay triangulation, but there might be circumstances when it's more sensible or practical to compute a smaller subset of the complete structure instead. To demonstrate this, in this section, the runtimes of the VGT algorithm and the quickhull algorithm will be analyzed and compared side by side, where the focus will be on the efficiency of above-mentioned algorithms with respect to dimensionality of the input. The quickhull algorithm was selected, because of its general prevalent use for efficiently processing high dimensional data and for the convenience of the Qhull package already existing as a default library inside MATLAB, which helps with achieving a more impartial comparison.

Regarding the VGT algorithm, both brute force and the nearest neighbor approaches are included in the runtime comparison. The number of steps in a random walk is set differently depending on the dimensions. After an initial subset of Voronoi vertices are computed, a random walk is initiated from each of the computed vertices that will traverse the undirected graph until the maximum length of the walk is reached, or for self-avoiding random walks, when a good deal of discovered vertices has been encountered. During a self-avoiding random walk, newly found vertices are stored, ensuring that the current and subsequent random walks avoid traversing towards an already explored vertex and instead take another path. For each random walk, there exists a unique counter variable, and each time a random walk travels towards an already stored vertex or along an unbounded edge, this counter variable is incrementally increased. If the counter variable becomes arbitrarily large, then the random walk is cut short and the next random walk is launched early. The idea

of this is to not waste time finding vertices in a region of the Voronoi graph that was largely explored. If the goal is to construct the full graph, then this setting can be turned off; however, merely finding a subset of vertices will serve the purpose of the algorithm and this test. Random walks can be set up differently depending on the task and size of the data: for example, with the method above; with the method above, except we let the walk traverse towards an already known vertex but keep count of it with the counter variable; or disable counting altogether and let the walk reach the maximum number of steps regardless of potentially storing duplicates.

All synthetic datasets consist of samples generated from a uniform distribution [0, 1], and each dataset will have a different sample size as well as dimension. The test will proceed with the lowest dimensional dataset and continue until the highest is processed. The dimensions included in the experiment range from 2 to 20, and during each test run, the time is recorded. In higher dimensions, the sample size will be smaller, since the excess computation time is nonessential to the experiment, and it becomes infeasible to store the graph in memory. In MATLAB there is an already built-in method for determining the time required to perform a function called "timeit". The method executes a function repeatedly and then calculates the median of the measurements, which is used to measure the performance of the implemented VGT algorithm and Qhull.

There are a few observations that can be made from this experiment. We can see that as the dimension grows, the running time of Qhull slows down considerably. According to the table, dimensionality of the data seems to have a greater effect on the performance of Qhull than sample size. While the VGT algorithm performs slower in lower dimensions $d < 12$, the benefits of it become much clearer in dimensions higher than 12. It should still be pointed out again that unlike Qhull, which computes all vertices of the full Voronoi graph, the VGT algorithm generally computes a subset of vertices. So, for dimensions $d < 10$, Qhull should still be the

VGT vs. Qhull					
Dimension	Data Points	Qhull (s)	VGT (s)	VGT init (s)	Random steps
2	1000	0.003	8.81	1.47	10
3	1000	0.013	10.5	2.596	10
5	1000	0.86	19.3	4.726	10
7	1000	69.092	130	8.237	100
10	100	65.321	82	14.818	100
12	50	39.781	37.1	0.429	1000
15	50	583.241	38.4	0.568	1000

Table 4.1: VGT Runtime vs. Qhull Runtime

algorithm of choice, which offers an efficient solution for datasets with large number of samples and directly returns the complete graph. In high dimensions, it's probably better to find an alternative more practical solution like the VGT algorithm. There is also an added benefit of the VGT algorithm, which allows for eventually more vertices to be discovered. Computed vertices can be stored long-term, and then be used to find more by performing random walks. Regarding the nearest neighbor search methods, intersection search outperforms brute forcing when the sample size is high, but when the sample size is low in any dimension, brute force might be a fine option. We can also see that just finding an initial subset of vertices takes way less time when looking at "VGT init", the largest portion of the runtime was taken up by performing random walks. At this point, we will also make an assumption that the number of steps required to discover an adequate amount of the graph depends heavily on the dimensionality of the dataset. For example, for dimension 2, random walks with 10 steps or less should suffice, but this will be investigated more in 4.2.

The time to find an initial subset of vertices takes considerably less time than

performing subsequent random walks when the number of steps is reasonably high. The number of random walks and steps required is heavily influenced by the dimensionality of the data. Even if the input size is small, the number of lower dimensional facets increase with respect to dimensionality. The sufficient number of steps in random walks for desirable results can be experimented with. Since this is a Monte Carlo method, it might take a lot of computation time to discover all vertices. self-avoiding random walks can speed up this process, but there is a possibility for a random walk of becoming stuck.

Due to random walks exploring in an incremental manner, it is possible to perform multiple runs of random walks to search separate parts of the structure at the same time in parallel.

4.2 Random Walk Analysis

To find a complete Voronoi graph with the VGT algorithm, it's possible to do two things; either increase the number of rays being cast from the initial generator points or increase the number of steps to perform with random walks. The benefit of the random walk is that it allows for the discovering of vertices that are further away from the starting vertex from which the random walk is initiated. With this approach, vertices belonging to other Voronoi generators, which contribute to building up their Voronoi sites, can be located. While with the increasing the number of rays approach, it's only possible to find the vertices belonging to the singular Voronoi generator from which the rays were cast. This then raises an interesting question; how many random walks are needed to find all Voronoi vertices and as a result construct the full Voronoi graph?

Qhull is used to compute the full graph with all vertices for a given dataset. The total number is then employed to compare the number of vertices that are discovered during a random walk. A test is performed by running the VGT algorithm several

times on the same dataset but with an ascending order of steps for the random walk. The test starts from 1 step and continues until a stopping criterion is reached. During this, each random walk setup is executed 5 times, and a median of the number of discovered unique vertices is taken. The stopping criterion is triggered when the percentage of unique vertices discovered has stagnated. Now, it's possible with this approach, for a dataset with a large sample size, the specific number of steps to build up the complete Voronoi graph won't be attained. But, in practice, it's probably more reasonable to find at which point the rate of change becomes too small and initiate a new round of random walks from all vertices, which include the newly discovered vertices as well. We'll also test self-avoiding random walks. The synthetic datasets comprise of points independently drawn from a uniform distribution on the interval [0, 1]. In higher dimensions, we expect that the number of steps in a random walk to be higher in order to obtain a reasonable portion of the Voronoi graph.

Random Walks in d = 2 With 1000 Points				
	Normal Random Walk		Self-Avoiding Random Walk	
Steps	Vertices Found %	Runtime (s)	Vertices Found %	Runtime (s)
1	∼66% (1303)	1.63	∼65% (1293)	2.97
10	∼85% (1692)	9.11	∼98% (1946)	13.9
100	∼86% (1696)	72.97	∼99% (1966)	17
500	∼86% (1696)	110.85	∼99% (1974)	17.5

[h]

Table 4.2: Random Walks in d = 2 With 1000 Points

In d = 2 (Table 4.2), we can see that with only one step, the algorithm finds more than half of the total number of vertices. This is intuitively clear, as without the initiation of the random walks, the algorithm would still find, in the best case, 1000 vertices because of the rays being cast from each generator (1000 in our case). And since a 2-dimensional Voronoi graph of n points (generators) can have at most $2n - 5$ vertices, it's possible to discover about a half of all the vertices. About 66% of vertices can be acquired by performing random walks with only 1 step, and with 10 steps per walk, approximately 85% of vertices were discovered. Increasing the number of steps past 10 does not result in any significant changes, and, in fact, has an adverse effect of prolonging the algorithm's runtime. Therefore, for d = 2, 1 to 10 steps would suffice. An alternative approach is utilizing self-avoiding random walks with 10 steps per walk, as it discovered almost the full Voronoi graph, around 98% of vertices, but with an increased runtime of ∼53% compared to normal random walks with 10 steps. Likewise, increasing the number of steps past 10 doesn't result in significantly more vertices.

Random Walks in d = 3 With 1000 Points				
	Normal Random Walk		Self-Avoiding Random Walk	
Steps	Vertices Found %	Runtime (s)	Vertices Found %	Runtime (s)
1	∼26% (1670)	3.65	∼26% (1648)	4.76
10	∼60% (3808)	10.9	∼74% (4711)	16.57
100	∼74% (4719)	84.8	∼88% (5604)	27.79
500	∼74% (4686)	395.04	∼87% (5543)	27.1
1000	∼72% (4573)	774.22	∼87% (5557)	27.58

Table 4.3: Random Walks in d = 3 With 1000 Points

In d = 3 (Table 4.3), we can see that the percentage of unique vertices found for

normal random walks starts stagnating after 10 steps, and the largest difference was the jump from 1 step to 10 steps, meaning that around 10 steps would be enough to find a large majority of vertices. The important aspect of this test is to find the best ratio of number of vertices found to the total runtime of the algorithm. Alternatively, for self-avoiding random walks, the sufficient number of steps is in the range of 10 to 100. Additionally, according to the table, self-avoiding random walks begin to become "stuck" after 10 - 100 steps, since the number of discovered vertices and runtime past 100 steps stays consistent and remains unchanged. A self-avoidant random walk is deemed stuck when the walker has taken too many steps towards previously visited vertices. This is a parameter that can be adjusted depending on the task, but also increases the runtime considerably if the maximum number of tries is raised, as it might result in more redundant searches.

Random Walks in d = 6 With 275 Points		
	Normal Random Walk	
Steps	Vertices Found %	Runtime (s)
1	<1% (532)	1.63
10	∼2% (2405)	3.68
100	∼16% (16066)	23.3
500	∼30% (30560)	110.85
1000	∼33% (33363)	230.68
5000	∼39% (39491)	1179.78

Table 4.4: Random Walks in d = 6 With 275 Points

The same trend continues, for d = 6 (Table 4.4), we also plotted the table in Figure 4.1. We can see that the number of unique vertices stagnates as the number of steps increases, while the runtime increases linearly with respect to the number

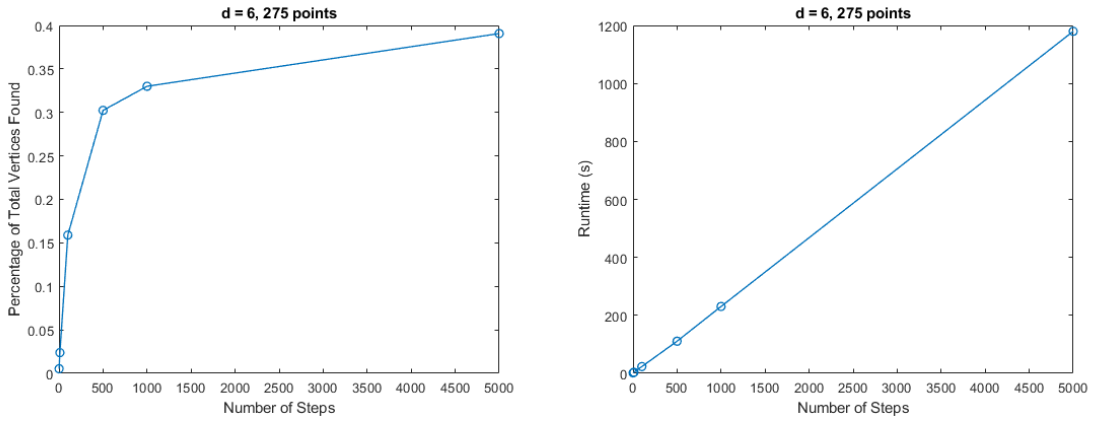

Figure 4.1: Relationship Between Number of Steps and Vertices Found and Runtime

of steps. This behavior will also be present in higher dimensions. In result, to find all vertices, more subsequent random walks need to be performed from newly found vertices.

Self-avoiding random walks were also performed in d = 6, but these got stuck almost immediately. As mentioned in the d = 3 case, it's possible to increase the maximum number of tries parameter in hopes of finding a bigger subset. But when this parameter was increased slightly, it resulted in impractical runtimes with only a slight increase of unique vertices. The look-up in self-avoiding random walks was set up naively to search a list of discovered vertices in $\mathcal{O}(n)$. In higher dimensions, where the number of facets is high, a more efficient search data structure is essential to utilize self-avoiding random walks. Because of this, we won't test self-avoiding random walks any further in higher dimensions.

42 points in d = 10 (Table 4.5) and 30 points in d = 15 (Table 4.6) both resulted in the same overall number of Voronoi vertices, which is around 100 thousand. 10000 random steps in d = 10 and d = 15 discovered ∼50% and ∼66% of the total number of vertices, respectively. This is quite strange, as intuitively initiating more rounds of random walks should result in more vertices to be found. This can also be seen from the runtimes, where d = 10 has longer runtimes than d = 15 with the same number of steps, but this is because in d = 10, 42 rounds of random walks are

Random Walks in d = 10 With 42 Points		
	Normal Random Walk	
Steps	Vertices Found %	Runtime (s)
1	<1% (80)	0.54
10	<1% (357)	0.82
100	∼3% (3178)	3.99
500	∼13% (13310)	16.52
1000	∼22% (22419)	32.6
5000	∼43% (43742)	170.42
10000	∼50% (51177)	314.13

Table 4.5: Random Walks in d = 10 With 42 Points

Random Walks in d = 15 With 30 Points		
	Normal Random Walk	
Steps	Vertices Found %	Runtime (s)
1	<1% (55)	0.49
10	<1% (280)	0.68
100	∼2% (2480)	2.91
500	∼9% (11398)	12.89
1000	∼17% (21201)	24.27
5000	∼53% (64913)	130.36
10000	∼66% (80778)	229.62

Table 4.6: Random Walks in d = 15 With 30 Points

executed (from each generator) compared to 30 rounds in d = 15.

If we were to increase the number of generators in d = 15 to 42 and compare the results to d = 10, then based on the behavior of random walks, the runtime would be the same for both given the number of steps is the same. Obviously, in the d = 15 instance, there would be more vertices requiring longer random walks.

It is not known why more vertices were found in d = 15, one reason could be that random walks in d = 10 found more redundant vertices and did not explore the more significant portions of the Voronoi graph. This also suggests an idea, instead of starting a random walk from each generator, it could be good to select a portion of the total number of generators at random and initiate random walks from these selected generators, which would decrease the overall runtime.

5 Piecewise Linear Interpolation

In this chapter, we will discuss how the simplicial complex of a Delaunay triangulation can be used for piecewise linear interpolation. A brief description of interpolation will be presented in Section 5.1. In Section 5.2, a method for finding a simplex for a given query point without the need to compute the full Delaunay triangulation is presented. And in the last Section 5.3, this method will be tested and analyzed.

5.1 Interpolation

A Delaunay triangulation forms a mesh of simplices which are useful for the linear interpolation of scattered data [34]. Interpolation is the estimation of the value of a new data point by known data points. Estimation is typically based on the values of surrounding points to calculate the interpolated value. The Delaunay triangulation also provides a model for triangulated irregular networks used for representing surfaces and offers a mesh of predetermined data points over which interpolation of unknown data points can be performed [35]. [36] is used as a reference for this section.

Suppose that two points given by their coordinates (x_1, y_1) and (x_2, y_2) are used in interpolation. x_i are the x-coordinates and y_i are the corresponding values. Let's also suppose that the points are in ascending order with respect to the x-axis, which is to say that $x_1 < x_2$. Suppose we are given an x-coordinate x that lies between x_1 and x_2 and for which the corresponding value or the y-coordinate y is unknown.

Now that the position of the point on the x-axis is known, we will have to estimate its position on the y-axis by utilizing the two known points (x_1, y_1) and (x_2, y_2). If we map the x values to the y values, we can define it as a function $y = L(x)$, where x is the independent variable and y is the dependent variable. Then for a given input value x, the function returns some output value y, which in relation to interpolation is the estimated value. The target function denoted by $f(x)$ is the function that outputs the exact value y. Realistically it might be too difficult to formulate $f(x)$, so an approximate function $L(x)$ is for that reason used to calculate a good estimate $\hat{y}$ while minimizing the error as much as possible. At this point, the exact form of the function f is not known. There are many interpolation methods, and one simple method is constant interpolation, where $L(x)$ is a step function. The interval $[x_1, x_2]$ is split into two smaller intervals $[x_1, a]$ and $[a, x_2]$ determined by proximity, where x values that fall within the interval $[x_1, a]$ are closer to x_1 than x_2 and additionally are assigned a y value of y_1, and vice versa with the interval $[a, x_2]$. To approximate the value of some input x, we check the interval within which x lies. For example, if x is on $[x_1, a]$, then its interpolated value is y_1. Another method is to use linear interpolation. Clearly, a straight line passes through both points (x_1, y_1) and (x_2, y_2). The idea of linear interpolation is to use this line as the interpolant $y = L(x)$. Therefore, the estimated value y is determined by:

$$y = L(x) = y_1 + \frac{(y_2 - y_1)}{(x_2 - x_1)}(x - x_1)$$

As a side note, if x does not fall under $[x_1, x_2]$ and is either $x < x_1$ or $x > x_2$, then the estimation of x is called extrapolation.

Suppose that we are given a finite number of data points by their coordinates $(x_0, y_0), \ldots, (x_n, y_n)$, where $x_i < x_{i+1}$ for each $i \in 0, 1, \ldots, n-1$. We are also given an x value, and the objective is to estimate $f(x)$ by making use of the known points over the whole interval $[x_0, x_n]$. An interpolant $L(x)$ is required to calculate $\hat{y}$, since

the exact function f is not known. For this problem, we shall only be concerned with linear interpolation with polynomials of degree 1 i.e., straight lines. And to reiterate, there are several types of interpolation [34] e.g., for smoother interpolants the degree of polynomials can be increased.

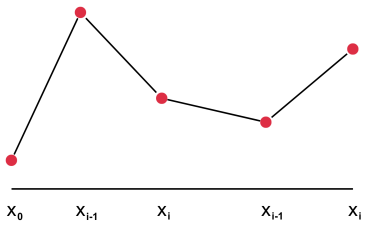

Figure 5.1: Piecewise Linear Interpolation

In the last example, linear interpolation was determined by two known points. For this example, the estimated value will be determined by an identical method except before interpolation, we will have to find the two best points from the whole set of points. We do this by subdividing the interval $[x_0, x_n]$ into $n-1$ segments, where each segment is an interval in the form of $[x_i, x_{i+1}]$, for each $i \in 0, ..., n-1$. The two points x_i and x_{i+1} in each segment create a straight line, and the collection of these lines can now be used to build a piecewise linear function which will be our interpolant. We then piecewise linearly interpolate the input value x in two steps: finding the interval $[x_i, x_{i+1}]$ under which x falls under; and estimating the output value y with x_i and x_{i+1}.

Until now, we have only been concerned with univariate functions, but interpolation on functions of more than one variable is also possible. For example, in bilinear interpolation, interpolation is performed along two lines and therefore requires at least 4 points. Interpolation on a d-dimensional Delaunay triangulation is also multivariate interpolation, where the first step is to determine the d-simplex with vertices $\{v_1, v_2, \ldots, v_{d+1}\}$ in which the query point lies within, and then estimate its value by using the values at the vertices of the d-simplex $f(v_i), i = 1, \ldots, d+1$. Conse-

quently, the interpolant is given by $\sum_{i=1}^{d+1} = w_i f(v_i)$, where w_i are the barycentric coordinates, which determine the location of the query point within the simplex (more on this in Section 5.2).

It has been shown that in $\mathbb{R}^2$, the shape of a triangle used in interpolation can have a drastic effect on the quality of the interpolation [37]. It is assumed that this is also generalizable for higher dimensions $\mathbb{R}^d$ with simplices. Typically, to improve linear interpolation, triangles that are deemed as bad candidates are avoided, and different triangulations on the data points are tried to provide better results. Triangulations, which are also dependent on the values at the data points, have been shown to produce better surfaces for piecewise linear interpolation [38].

5.2 Visibility Walk

In the original paper [1], the researchers proposed a method for estimating a query point called the visibility walk, which incorporates components of the VGT algorithm. For this thesis, a slightly modified visibility walk was implemented and assessed.

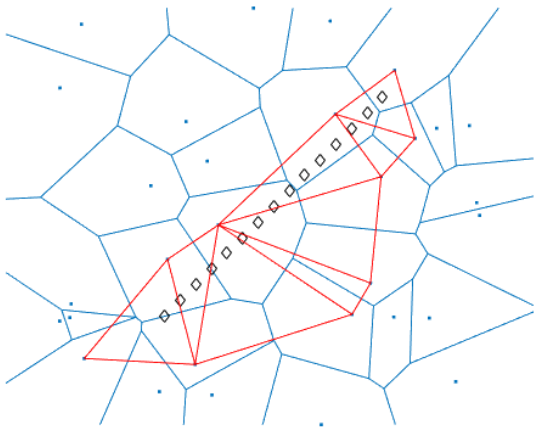

Figure 5.2: Simplex Interpolation

A Voronoi graph can be perceived as a visibility graph, where a vertex represents a location, and each edge represents a traversable path from one vertex (location) to another. At each vertex, other vertices are visible indicating that a walk toward them is possible. In the case of a d-dimensional Voronoi graph, there are $d+1$ paths for each vertex. As for some examples of practical applications of visibility graphs, these include path planning, time series representation and analysis.

Let's denote a query point by q and a set of simplices of a d-dimensional Delaunay triangulation by S. The first step is to determine whether $q \in \mathbb{R}^d$ lies in any simplex $s \in S$. This can be done by calculating the barycentric coordinates $(a_0 : \ldots : a_d)$ of q with respect to the vertices of s. If $a_i > 0$, for all $i = (0, \ldots, d)$, and $\sum a_i <= 1$, then q lies inside s. It is also fine for some of the numbers in the barycentric coordinates to be zero, in which event q is on a lower dimensional face of s. If every number is 0 except for one, which should be 1, then q is on one of the vertices of s. q lies outside the convex hull of S if there does not exist an appropriate s from S for q. But once an appropriate s is found, the values provided by the vertices of s or the weights determined by the barycentric coordinates can be used to estimate the value of q.

The next question is how to find such a simplex? One way is to calculate the full or a sufficient portion of the d-dimensional Delaunay triangulation and do a search over the whole set of discovered simplices, which has a worst-case time complexity of $\mathcal{O}(n)$. But, with this approach, it's going to take a much longer time before linear interpolation can be performed. In situations in which the construction of a Delaunay triangulation is not of interest, it might be preferable to reduce the amount of time required to acquire the desired simplex. Here is where the components of the VGT algorithm can be reutilized for this problem.

Let's denote the set of Voronoi generators by $P = \{x_1, x_2, \ldots, x_n\}$ in Euclidean space $\mathbb{R}^d$. The general idea is to "travel" the portion of the Voronoi graph that is closer to the query point q and compute simplices until the desired one is found.

The desired simplex is calculated by using barycentric coordinates in the way it is described above. Before a simplex can be constructed using the VGT algorithm, Voronoi vertices are required to begin with. Therefore, a nearest neighbor search is performed for the query point q to find the nearest generator $x_q \in P$. We then follow a similar procedure described in section 3.3; a ray is generated with a random direction from x_q and orthogonally projected onto the facet on which we wish the ray to travel, the intersection of the ray hitting a boundary component of a lower dimensional facet is calculated, and this is repeated until a Voronoi vertex (0-facet) is reached. Now that an initial Voronoi vertex is computed, we have secondarily gathered one of the simplices of a Delaunay triangulation. We can also utilize random walks from the Voronoi vertex to discover more Voronoi vertices and simplices. We check whether q lies inside the simplex, if not, then q either lies some other simplex or outside the convex hull of the simplicial complex. In other words, we require more simplices to be discovered to make this determination. Instead of letting the random walk traverse the graph too randomly and finding simplices that are obviously too far away from q, we would like to have a bit more control over the path the walker takes. Since the nearest Voronoi generator x_q is known, by keeping x_q and removing other generators during the random walk, we can make sure that the walker never steps out of the Voronoi cell generated by x_q. The walker is then let traversing this cell until all its Voronoi vertices are discovered or the simplex that triangulates q is found. If the desired simplex is not found within the cell, then more random walks are initiated from the discovered Voronoi vertices towards the adjacent cells by removing x_q. The behavior of the random walks on the new cells will remain the same as in the case with the cell generated by x_q, so the walker will only search "locally". At this point, if the desired simplex is still not found, then we can determine that q probably lies outside the convex hull.

Regarding the time complexity of this method, since the components of the VGT

algorithm are used, the time is dependent on the time complexities of each of the components, which were presented in Section 3.4. Nevertheless, we will list the components and their time complexities, as well as the overall time complexity required to run the modified visibility walk. Of course, the exact total runtime depends on the query point and the path a random walk takes, which will be discussed more in the next Section 5.3.

By using a nearest neighbor search algorithm to find the closest Voronoi generator x_q for a given query point q, it will take $\mathcal{O}(NN(\cdot))$ time. Then we "descent" from a higher dimensional facet towards the lowest dimensional facet (vertex) by executing a series of random ray sampling and intersection searches d times. Random ray sampling also consists of orthogonal projection, so its overall time complexity is $\mathcal{O}((f) + dk^2)$, where $f = 1, 2, \ldots, d+1$, while intersection search takes $\mathcal{O}(\log \frac{r_{length}}{\epsilon} \cdot NN(\cdot))$. Subsequently, a random walk with s steps is performed from discovered vertex. One step in a random walk incorporates random ray sampling to randomly choose an edge and casting a ray along it and a single intersection search to determine the intersection point which is a new vertex. Therefore, the time complexity of a singular step in a random walk is $\mathcal{O}((d) + dk^2) + \mathcal{O}(\log \frac{r_{length}}{\epsilon} \cdot NN(\cdot))$. Each time an unexplored vertex is found, its dual – a simplex of a Delaunay triangulation – is also discovered. For the duration of the random walk the barycentric coordinates are calculated for each new simplex, which takes $\mathcal{O}(1)$ time.

5.3 Testing

The most vital aspect of this piecewise linear interpolation method, which is based on the idea of the VGT algorithm, is the process of searching and finding the desired simplex for interpolation. This section is devoted exclusively to evaluating this characteristic, which is accomplished in two stages: validation and analysis.

To demonstrate the validity of interpolation, a synthetic dataset of random points

drawn from an elementary isotropic manifold and a random query point that lies inside the manifold are generated. The points in the dataset form a Delaunay triangulation, on which the search for a simplex and interpolation is performed. Since the intrinsic shape of the manifold is known beforehand, it becomes easy to determine the desired simplex and whether a simplex exists inside the convex hull of the Delaunay triangulation.

Subsequently, regarding the analysis part, the behavior of the interpolation method is examined when it is deployed in a more realistic environment. The Delaunay triangulations for interpolation are generated in the same manner as in the VGT algorithm testing section in 4.1. The interpolation is performed on multiple synthetic datasets consisting of randomly generated points that vary in dimensions. With this approach, it is possible to observe the average time to discover the desired simplex and the average length of a random walk (number of steps) in d-dimensions.

5.3.1 Validation

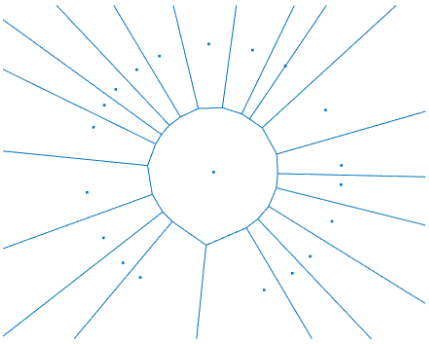

Figure 5.3: Voronoi Diagram Generated for Validation

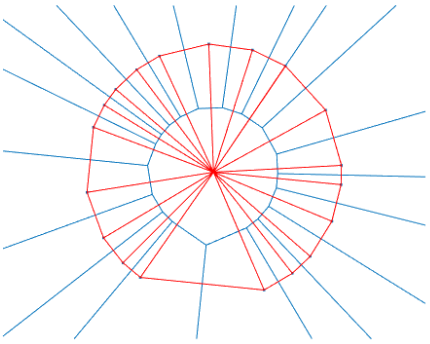

Figure 5.4: Delaunay Triangulation Generated for Validation

Let's denote a dataset as $S = \{x_1, \ldots, x_n\}$, which is a finite set of points in

Euclidean space $\mathbb{R}^d$, where each $x \in S$ is a uniformly generated point on the hypersurface of a d-sphere $S = \{x \in \mathbb{R}^{d+1} \mid \|x\| = 1\}$. The d-sphere is centered at the origin. Let's also denote a particular query point we wish to estimate by q and an extra point located at the origin by O. O is then added to be a part of the dataset S before interpolation. This is done to ensure that no accuracy issues associated with floating point arithmetic are present during the process. Clearly, with sufficient number of points the volume of the convex hull of the Delaunay triangulation will approximate the volume of the d-sphere. This also lets us know if q lies inside the convex hull and on which spherical sector, which simplifies the validation of interpolation.

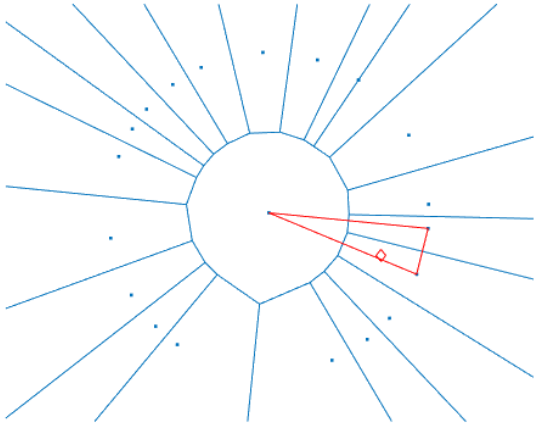

Figure 5.5: Interpolating an Arbitrary Point that Lies Within the Convex Hull of the Delaunay Triangulation

While it's theoretically possible for 3 points to be collinear under these conditions, where two randomly generated points x_i and x_j and O form a diameter of the d-sphere, the occurrence of this transpiring is highly improbable with randomly generated points. For this reason, it is assumed that the points are in general position. Regarding the query point q, it is generated on the hypersurface of another d-sphere $\{q \in \mathbb{R}^{d+1} \mid \|q\| = 0.75\}$ that is still centered at the origin but has a slightly

Piecewise Linear Interpolation Validation Test			
Dimension	Runtime (s)	Number of Data Points	Number of Steps
2	0.008	1000	1000
3	0.034	...	...
4	0.08	...	...
6	2.566	100 000	10 000
8	24.83	...	...

Table 5.1: Piecewise Linear Interpolation Validation Test Results

smaller radius. This is done because if the distance between q and the origin is 0.5, then it lies on the inside of the cell belonging to O. With the way this validation test is set up, the boundary of the cell generated by O will include all vertices of the whole Voronoi diagram. As a result, the initial ray will be cast from within the cell generated by O, precisely from the origin, and will hit a part of the Voronoi diagram that is far away from the desired simplex with a high degree of probability. And since the interpolation method is set up in such a way to first search the initial closest cell with regards to the query point and all the nearby cells after with which the initial cell shares a vertex, the search would take a rather large amount of time. This is best illustrated in Figure 5.3.

For this test, 10 query points were used as input, and the time it took to interpolate each query point is calculated. This is then executed 10 times for each dimension with the same exact setup, subsequently the mean is calculated of all runtimes obtained from each individual execution to get the average runtime.

There are a few interesting remarks that can be made from Table 5.1. By this point in the thesis, it has become clear how the curse of dimensionality manifests itself. So, it is not a surprise to see that the discovery of a desired simplex in higher

dimensions takes the interpolation algorithm a longer time to run as there are more vertices to visit. We also kept count of how many times the algorithm was able to interpolate query points in some dimension d. In other words, interpolation was successful if the desired simplex was found during the random walk for a given query point. To get meaningful results, it was required to increase the number of steps of random walks in higher dimensions, but there were still failed attempts. This means that the number of steps is a crucial parameter for locating the desired simplices.

Due to the nature of a random walk, the simplex search time even for the same query point can differ by quite a lot. The best case is obviously when the initial vertex before starting the walker holds the desired simplex right from the start. The worst case is that simplex is found at the last step or not found at all. In Section 3.2.4, it was mentioned that random walks on lattices, typically consisting of less paths, tend to stay localized while with more paths tend to go further away. Therefore, for higher dimensions it would probably be a good idea to increase the number of rays and search for the simplex sequentially.

Overall, the algorithm provides a great method for interpolating points in lower dimensions while also having the option for higher dimensions without the need to precalculate the whole or portions of the Voronoi diagram or Delaunay triangulation beforehand.

5.3.2 Analysis

In this subsection, the performance of the piecewise linear interpolation method is measured with regards to runtime with scattered data. The algorithm is executed multiple times during a single test run with some configuration (data, number of query points, type of random walk) and the final runtime is then calculated by taking the average of all the runtimes occurred in the test run. During the search, the number of steps the walker takes will also be tracked. Since the algorithm is based on

the Monte Carlo method, it's entirely possible to find the desired simplex for a given query point earlier or later in the random walk's pathing. Thus, multiple runs of the algorithm will be executed for the same query point with the same predetermined Delaunay triangulation. Later, a set of query points will be interpolated with random Voronoi and Delaunay structures tessellated by synthetic datasets. Since the number of vertices in a Voronoi cell increases exponentially with respect to dimensionality, we can expect the interpolation method to take a longer time to explore a cell for the desired simplex in higher dimensions. To speed up this process, it is possible to utilize self-avoiding random walks or increase the number of rays being cast from within the cell to travel its boundary. For future work, there is also an option to run this interpolation algorithm in parallel as random walks can be quite effortlessly parallelized. But nevertheless, for this test the synthetic data will be processed sequentially.

The synthetic data is identically generated as in Section 4.1, where data points are sampled from a uniform distribution $[0, 1]$ in some dimension d, but the query points will be randomly drawn from a uniform distribution $U([0.25, 0.75])$ to avoid query points lying outside the convex hull of the Delaunay triangulation. The triangulation for each dimension is generated with 1000 data points, and the total number of query points that that are interpolated is 10, likewise, for each dimension. We also adjusted the algorithm to only search one Voronoi cell, the cell the query point belongs to, and not to search nearby cells in the event of an unsuccessful simplex search. This is done to avoid situations when, though, a query point is inside the convex hull, the Voronoi vertex exists outside the cell in which the query point lies. We have observed that in a vast number of cases the vertex that holds the desired simplex is found on the first cell, thus for this test these situations would increase the runtime drastically and could count as outliers.

Average Runtime of Successful Interpolations w.r.t Dimension		
Dimension	Number of Random Steps	Average Runtime (s)
2	1000	0.018
3	1000	0.024
4	10 000	0.58
6	100 000	7.34
8	300 000	214.07

Table 5.2: Average Runtime of Successful Interpolations w.r.t Dimension

Table 5.2 consists the results of this test, and just likes with the validation test, it takes the interpolation algorithm a longer time to search for the desired simplex with higher dimensional data than it is with lower dimensional, which coincides with previous assumptions and realities of the curse of dimensionality. But the more interesting aspect is the number of steps that is at minimum required to find vertices, and as a result discover the simplex to use for interpolation. This has been mentioned in the validation section, but we will test it for dimensions 2 and 6 with this same setup with 100 query points. We will also repeat this test 5 times for both dimensions and then take the average number of successful searches. The results of this test can be seen in Table 5.3. A different strategy for higher dimensions could probably be useful, for example, executing random walks at different locations or rather vertices of the Voronoi cell and running the algorithm in parallel. A self-avoiding random walk with an efficient search data structure can also be useful in such circumstances.

In 2 dimensions, we see that the range of adequate number of steps seems to be between 10 and 100 at which point we can reliably find the simplex on which to interpolate our query point. As we go higher in dimension, the number of steps also

Number of Random Steps vs. Successful Searches w.r.t Dimension		
Dimension	Number of Random Steps	Average Number of Successful Searches
2	1	15.2
	10	70.8
	100	96.98
	1000	95
6	1	0
	10	0.1
	100	0.9
	1000	12.2
	10 000	63
	30 000	91.4

Table 5.3: Number of Random Steps vs. Successful Searches w.r.t Dimension

grows to get the intended results.

Since only the Voronoi cell to which the query point is closest to was explored, there is a possibility of the desired vertex and its dual simplex existing in a different location of the Voronoi diagram. This would then necessitate searching more exhaustively by performing random walks on other nearby cells.

6 Conclusions

Throughout this thesis, it became evident some compromises are necessary in order to compute space partitioning structures like the Voronoi diagram or Delaunay triangulation or to perform piecewise linear interpolation, e.g. with simplices, in high dimensions. There are many algorithms and code packages such as Qhull which offer fast and efficient methods for constructing either structure, but in high-dimensional space many approaches become impractical or infeasible with respect to time and space complexity. Many of the solutions focus on the construction of the complete structure with all its lower dimensional faces intact, but the VGT algorithm compromises on this by computing a subset of vertices instead to obtain access to a portion of the structure faster. This approach still allows for further exploration of the structures with random walks to discover repeatedly more and more vertices or simplices if needed. But if the intent is the computation of the complete structure, then the VGT algorithm probably isn't going to be considerably more efficient than other popular algorithms, and in fact, could be slower because of the randomized behavior, but can still be used to find a vast majority of vertices or simplices. Ultimately, this is the trade-off the algorithm makes for quicker computation time, as it seems there is no escaping the curse of dimensionality. If the complete Voronoi diagram or its dual, the Delaunay triangulation, is not in consideration, then this method provides a good solution for exploring both structures in any dimension, but largely for dimensions higher than 10.

Approximation methods for scattered data are necessary in many fields, where a Delaunay triangulation becomes useful for piecewise linear interpolation. In [39], a parameter performance prediction framework for a big data analytics engine was built by constructing a d-dimensional Delaunay triangulation over the whole feature space to estimate the runtime of parameter configurations. In [40], a data compression method using a Delaunay triangulation is presented for a geographic information system application for visual terrain representation. In [41], Delaunay triangulation and a singularly weighted k-nearest neighbor scheme are examined to provide insight of interpolating classifiers to explain strong generalization performance of optimal machine learning models. In [42], a reversible data hiding method for embedding data into images is introduced. This method, first, constructs a 3-dimensional Delaunay triangulation containing the pixel values of an image, which is then utilized to acquire an interpolated image of the original and insert information into it. Performing piecewise linear interpolation with the VGT algorithm enables us to find and interpolate a given query point without the need to precompute every simplex of a Delaunay triangulation. During testing, this method was quite efficient for dimensions between 2 and 8.

As for future work, the algorithm could be expanded to compute other types of Voronoi diagrams such as higher order or weighted Voronoi diagrams. The original algorithm by the authors of the paper can be run in parallel utilizing a parallel computation library, therefore parallelization could also be added as a feature for the MATLAB version by taking advantage of the build-in parallel computing toolbox.

References

[1] V. Polianskii and F. Pokorny, "Voronoi graph traversal in high dimensions with applications to topological data analysis and piecewise linear interpolation", pp. 2154–2164, Aug. 2020. DOI: 10.1145/3394486.3403266.

[2] A. Okabe, "Spatial tessellations", in *International Encyclopedia of Geography*. John Wiley Sons, Ltd, 2017, ISBN: 9781118786352. DOI: https://doi.org/10.1002/9781118786352.wbieg0601. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118786352.wbieg0601. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118786352.wbieg0601.

[3] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure", *ACM Comput. Surv.*, vol. 23, no. 3, pp. 345–405, Sep. 1991, ISSN: 0360-0300. DOI: 10.1145/116873.116880. [Online]. Available: https://doi.org/10.1145/116873.116880.

[4] F. Aurenhammer and R. Klein, *Voronoi Diagrams* (Informatik-Berichte), en. Hagen: FernUniversität in Hagen, 1996, vol. 198. [Online]. Available: https://ub-deposit.fernuni-hagen.de/receive/mir_mods_00001064.

[5] V. Klee, "On the complexity of d-dimensional voronoi diagrams", *Archiv der Mathematik*, vol. 34, pp. 75–80, 1979. [Online]. Available: https://api.semanticscholar.org/CorpusID:118753732.

[6] C. Aggarwal, A. Hinneburg, and D. Keim, "On the surprising behavior of distance metric in high-dimensional space", *First publ. in: Database theory, ICDT 200, 8th International Conference, London, UK, January 4 - 6, 2001 / Jan Van den Bussche ... (eds.). Berlin: Springer, 2001, pp. 420-434 (=Lecture notes in computer science ; 1973)*, Feb. 2002.

[7] S. Fortune, "Voronoi diagrams and delaunay triangulations", pp. 225–265, DOI: 10.1142/9789812831699_0007. eprint: https://www.worldscientific.com/doi/pdf/10.1142/9789812831699_0007. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/9789812831699_0007.

[8] C. L. Lawson, "Properties of n-dimensional triangulations", *Computer Aided Geometric Design*, vol. 3, no. 4, pp. 231–246, 1986, ISSN: 0167-8396. DOI: https://doi.org/10.1016/0167-8396(86)90001-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167839686900014.

[9] P. Su and R. L. Scot Drysdale, "A comparison of sequential delaunay triangulation algorithms", *Computational Geometry*, vol. 7, no. 5, pp. 361–385, 1997, 11th ACM Symposium on Computational Geometry, ISSN: 0925-7721. DOI: https://doi.org/10.1016/S0925-7721(96)00025-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925772196000259.

[10] A. Bowyer, "Computing Dirichlet tessellations*", *The Computer Journal*, vol. 24, no. 2, pp. 162–166, Jan. 1981, ISSN: 0010-4620. DOI: 10.1093/comjnl/24.2.162. eprint: https://academic.oup.com/comjnl/article-pdf/24/2/162/967239/240162.pdf. [Online]. Available: https://doi.org/10.1093/comjnl/24.2.162.

[11] D. F. Watson, "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes*", *The Computer Journal*, vol. 24, no. 2, pp. 167–

172, Jan. 1981, ISSN: 0010-4620. DOI: 10.1093/comjnl/24.2.167. eprint: https://academic.oup.com/comjnl/article-pdf/24/2/167/967258/240167.pdf. [Online]. Available: https://doi.org/10.1093/comjnl/24.2.167.

[12] P. Cignoni, C. Montani, and R. Scopigno, "Dewall: A fast divide and conquer delaunay triangulation algorithm in ed", *Computer-Aided Design*, vol. 30, no. 5, pp. 333–341, 1998, ISSN: 0010-4485. DOI: https://doi.org/10.1016/S0010-4485(97)00082-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010448597000821.

[13] R. A. Dwyer, "Higher-dimensional voronoi diagrams in linear expected time", SCG '89, pp. 326–333, 1989. DOI: 10.1145/73833.73869. [Online]. Available: https://doi.org/10.1145/73833.73869.

[14] S. Fortune, "A sweepline algorithm for voronoi diagrams", *Algorithmica*, vol. 2, pp. 153–174, Nov. 1987. [Online]. Available: https://doi.org/10.1007/BF01840357.

[15] K. Q. Brown, "Voronoi diagrams from convex hulls", *Information Processing Letters*, vol. 9, no. 5, pp. 223–228, 1979, ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(79)90074-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020019079900747.

[16] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls", *ACM Trans. Math. Softw.*, vol. 22, no. 4, pp. 469–483, Dec. 1996, ISSN: 0098-3500. DOI: 10.1145/235815.235821. [Online]. Available: https://doi.org/10.1145/235815.235821.

[17] C. Barber, *Qhull*, http://www.qhull.org, Accessed: 2023-07-3.

[18] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data", *IEEE Transactions on Pattern Analysis and Machine Intel-*

ligence, vol. 36, no. 11, pp. 2227–2240, 2014. DOI: 10.1109/TPAMI.2014.2321376.

[19] W. Li, Y. Zhang, Y. Sun, W. Wang, W. Zhang, and X. Lin, "Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement (v1.0)", 2016. arXiv: 1610.02455 [cs.DB].

[20] R. Weber, H.-J. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces", VLDB '98, pp. 194–205, 1998.

[21] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality", *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, vol. 604-613, Oct. 2000. DOI: 10.1145/276698.276876.

[22] J. Wang, "Geometric structure of high-dimensional data", in *Geometric Structure of High-Dimensional Data and Dimensionality Reduction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 51–77, ISBN: 978-3-642-27497-8. DOI: 10.1007/978-3-642-27497-8_3. [Online]. Available: https://doi.org/10.1007/978-3-642-27497-8_3.

[23] S. Dasgupta and A. Gupta, "An elementary proof of a theorem of johnson and lindenstrauss", *Random Structures & Algorithms*, vol. 22, no. 1, pp. 60–65, 2003. DOI: https://doi.org/10.1002/rsa.10073. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rsa.10073. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rsa.10073.

[24] A. S. Glassner, Ed., *An introduction to ray tracing*. GBR: Academic Press Ltd., 1989, ISBN: 0122861604.

[25] A. Appel, "Some techniques for shading machine renderings of solids", AFIPS '68 (Spring), pp. 37–45, 1968. DOI: 10.1145/1468075.1468082. [Online]. Available: https://doi.org/10.1145/1468075.1468082.

[26] D. P. Kroese, T. Brereton, T. Taimre, and Z. I. Botev, "Why the monte carlo method is so important today", *WIREs Computational Statistics*, vol. 6, no. 6, pp. 386–392, 2014. DOI: https://doi.org/10.1002/wics.1314. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/wics.1314. [Online]. Available: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/wics.1314.

[27] D. R. S. Geoffre R. Grimmett, *Probability and Random Perocesses*. Oxford University Press, Oxford; New York, 2001.

[28] G. F. Lawler and V. Limic, "Random walk: A modern introduction", 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:117107657.

[29] L. László, L. Lov, and O. Erdos, "Random walks on graphs: A survey", May 1994, pp. 1–46.

[30] G. Pólya, "Über eine aufgabe der wahrscheinlichkeitsrechnung betreffend die irrfahrt im straßennetz", *Mathematische Annalen*, vol. 84, pp. 149–160, 1921. [Online]. Available: http://eudml.org/doc/158886.

[31] E. W. Montroll, "Random walks in multidimensional spaces, especially on periodic lattices", *Journal of the Society for Industrial and Applied Mathematics*, vol. 4, no. 4, pp. 241–260, 1956. DOI: 10.1137/0104014. eprint: https://doi.org/10.1137/0104014. [Online]. Available: https://doi.org/10.1137/0104014.

[32] F. Xia, J. Liu, H. Nie, Y. Fu, L. Wan, and X. Kong, "Random walks: A review of algorithms and applications", *IEEE Transactions on Emerging Topics in*

Computational Intelligence, vol. PP, pp. 1–13, Nov. 2019. DOI: 10.1109/TETCI.2019.2952908.

[33] M. E. Muller, "A note on a method for generating points uniformly on n-dimensional spheres", *Commun. ACM*, vol. 2, no. 4, pp. 19–20, Apr. 1959, ISSN: 0001-0782. DOI: 10.1145/377939.377946. [Online]. Available: https://doi.org/10.1145/377939.377946.

[34] I. Amidror, "Scattered data interpolation methods for electronic imaging systems: a survey", *Journal of Electronic Imaging*, vol. 11, no. 2, pp. 157–176, 2002. DOI: 10.1117/1.1455013. [Online]. Available: https://doi.org/10.1117/1.1455013.

[35] L. Mitas and H. Mitasova, "Spatial interpolation", *Geographical information systems: principles, techniques, management and applications*, vol. 1, no. 2, 1999.

[36] M. Yano, "Chapter 2: Interpolation", *Math, Numerics, & Programming (for Mechanical Engineers)*, pp. 17–39, 2013, MIT OpenCourseWare.

[37] S. Rippa, "Long and thin triangles can be good for linear interpolation", *Siam Journal on Numerical Analysis - SIAM J NUMER ANAL*, vol. 29, Feb. 1992. DOI: 10.1137/0729017.

[38] N. Dyn, D. Lewis, and S. Rippa, "Data dependent triangulations for piecewise linear interpolation", *IMA Journal of Numerical Analysis*, vol. 10, pp. 137–154, Jan. 1990. DOI: 10.1093/imanum/10.1.137.

[39] Y. Chen, P. Goetsch, M. A. Hoque, J. Lu, and S. Tarkoma, "*d*-simplexed: Adaptive delaunay triangulation for performance modeling and prediction on big data analytics", *IEEE Transactions on Big Data*, vol. 8, no. 2, pp. 458–469, 2022. DOI: 10.1109/TBDATA.2019.2948338.

[40] B. Pradhan, K. Sandeep, S. Mansor, A. Rahman Ramli, and A. R. B. M. Sharif, "Second generation wavelets based gis terrain data compression using delaunay triangulation", *Engineering Computations*, vol. 24, no. 2, pp. 200–213, Jan. 2007, ISSN: 0264-4401. DOI: 10.1108/02644400710729572. [Online]. Available: https://doi.org/10.1108/02644400710729572.

[41] M. Belkin, D. Hsu, and P. Mitra, *Overfitting or perfect fitting? risk bounds for classification and regression rules that interpolate*, 2018. arXiv: 1806.05161 [stat.ML].

[42] W. Hong, T.-S. Chen, and J. Chen, "Reversible data hiding using delaunay triangulation and selective embedment", *Information Sciences*, vol. 308, pp. 140–154, 2015, ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2014.03.030. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025514003144.