Summer 2020

Novel approaches for constructing persistent Delaunay triangulations by applying different equations and different methods

Esraa Habeeb Khaleel Al-Juhaishi

NOVEL APPROACHES FOR CONSTRUCTING PERSISTENT DELAUNAY TRIANGULATIONS BY APPLYING DIFFERENT EQUATIONS AND DIFFERENT METHODS

by

ESRAA HABEEB KHALEEL AL-JUHAISHI

A DISSERTATION

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

2020

Approved by:

ABSTRACT

Delaunay triangulation and data structures are an essential field of study and research in computer science, for this reason, the correct choices, and an adequate design are essential for the development of algorithms for the efficient storage and/or retrieval of information. However, most structures are usually ephemeral, which means keeping all versions, in different copies, of the same data structure is expensive. The problem arises of developing data structures that are capable of maintaining different versions of themselves, minimizing the cost of memory, and keeping the performance of operations as close as possible to the original structure. Therefore, this research aims to aims to examine the feasibility concepts of Spatio-temporal structures such as persistence, to design a Delaunay triangulation algorithm so that it is possible to make queries and modifications at a certain time t, minimizing spatial and temporal complexity. Four new persistent data structures for Delaunay triangulation (Bowyer-Watson, Walk, Hybrid, and Graph) were proposed and developed. The results of using random images and vertex databases with different data (DAG and CGAL), proved that the data structure in its partial version is better than the other data structures that do not have persistence. Also, the full version data structures show an advance in the state of the technique. All the results will allow the algorithms to minimize the cost of memory.

.

ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Dr. George Markowsky, for his great advising to my doctoral work. Dr. George Markowsky Provided me with every bit of guidance, assistance, and expertise that I needed. this study would not have been as enjoyable as it was without his great help, advising, motivation, and encouragement. Words cannot express how grateful I am to have such an amazing advisor.

I would like to thank the members of my advisory committee: Dr. Ricardo Morales, Dr. Patrick Taylor, Dr. Siddhardh Nadendla, and Dr. Robert Paige. Each member provided valuable knowledge, great technical experience, and suggestions that continuously improved my knowledge and understanding.

I would like to thank all the staff in the Computer Science Department for help.

I would like to thank the most important people in my life my parents, whose love and guidance are with me in whatever I pursue. They are the ultimate role models. Most importantly, I wish to thank my loving, motivating, and supportive husband, Wesam, and my three fabulous children, Asawer, Aws, and Serein, who provide unending inspiration.

Last but not least, I would like to acknowledge the Ministry of Higher Education and Scientific Research of Iraq for the support during this study.

.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Page

LIST OF TABLES

1. INTRODUCTION

1.1. GENERAL INTRODUCTION

Data structures make up a fundamental field of study and research in computer science, for this reason, the correct choice and an adequate design are essential for the development of algorithms for the efficient storage and/or retrieval of information. However, most structures are usually ephemeral, which means that in each insertion, elimination, or modification operation, they lose the previous values or states. On the other hand, persistent data structures are known as temporary data structures [1], they allow them to maintain previous versions of the structure, so they retain previous states when they are modified. All this, taking into account the computational and spatial cost characteristics.

Delaunay triangulation is a fundamental data structure in computational geometry, graphing, engineering, and other areas of algorithmic application [2], [3]. For this reason, its usefulness and that of its dual (Voronoi diagrams) have various applications such as solution to the problem of finding the closest point [4], land modeling [5], collision detection [6], associative file search, a solution to the problem of data grouping, among others.

On the other hand, Delaunay triangulation, which is currently implemented in libraries such as CGAL (Computational Geometry Algorithms Library, [7]), does not have partial persistence or total persistence, which implies that if a set of operations is applied such as insertion, elimination or modification on the data structure in time, then, it loses the previous states. This also occurs in all traditional data structures such as lists, stacks, queues, balanced binary trees, B, B + trees, and many others. This type of structure has the problem of not being able to return to previous state i, after applying n modifications $(0 < i < n)$; it is for this reason that this type of classical structure is known as ephemeral data structures.

For example, if the value of a node in a simple linked list is modified n times, it is not possible to know the value that it had at a previous time i $(0 < i < n)$. This is because the changes, in this type of structure, are not stored. On the other hand, keeping all versions, in different copies, of the same data structure is expensive. Then, the problem arises of developing data structures that are capable of maintaining different versions of themselves, minimizing the cost of memory, and keeping the performance of operations as close as possible to the original structure. These types of data structures are called persistent data structures.

In general, data structures can be classified as partially persistent, fully persistent, confluent, and functional. The first are those structures in which it is only possible to make modifications in the current state, however, queries can be made in any state, that is, in any past time. On the other hand, completely persistent data structures (the definition in Section 2) are those to which modifications can be applied at any past time, and at the same time, it is possible to make queries at any time or in any state. Then, it is clear that the design of fully persistent data structures is more complex due to the amount of memory they can use.

In the current literature, there is no fully persistent Delaunay triangulation data structure yet. In this sense, this dissertation has a contribution: the proposal of a new persistent data structure to represent a Delaunay triangulation.

1.2. THE PROBLEM JUSTIFICATION AND MOTIVATION

There are many libraries for calculating Delaunay triangulation, such as CGAL, Fade2D, among others. These libraries have efficient algorithms for calculating Delaunay triangulations; however, they do not have persistence characteristics; that is, they do not have the possibility of maintaining versions of the structures every time operations are carried out on them.

Furthermore, an important aspect that data structures should have is the ability to maintain past versions of themselves, in order to return to a previous t version; so that it is feasible to evaluate the characteristics of the structure at that time.

In this sense, this research proposes the development of a persistent data structure for the management of a Delaunay triangulation. This data structure, in addition to maintaining past versions of the structure, will maintain computational complexity and minimize spatial complexity.

As mentioned above, in addition to the lack of a persistent data structure for Delaunay triangulation, the existence of such a data structure would be useful in different applications, such as on a two-dimensional map where different points have been marked, which indicate regions where different natural phenomena have originated. In this model, you want to perform simulations on the possible nearby areas that could be affected. In this sense, Delaunay's persistent triangulation results in a natural data structure for this problem, since it seeks to minimize RAM memory consumption in those computational problems that can be represented or solved using Delaunay triangulations and variable in time product of insertions, deletions and/or modifications.

On the other hand, let's imagine that we have a database of satellite images of a river [8], taken at different periods of time. We can represent the image of the river by means of a Delaunay triangulation. However, with the passage of time it is clear that the characteristics of the river bed change, increasing flow or modifying its shape, this means that several images of the same river are kept on different dates; therefore, different Delaunay triangulations should be constructed. However, our method generates a single Delaunay triangulation, which allows minimizing RAM memory and maintaining the reconstruction performance of the triangulation in a specific time.

1.3. THE PROBLEM STATEMENT AND THE OBJECTIVE

Ephemeral data structures do not have mechanisms to make queries in past times, in this sense, one of the most trivial ways of maintaining the information of the versions generated by the modifications in the data structures is through copies of themselves. the same after each modification. This method is clearly counterproductive when it comes to memory spending.

On the other hand, the data structures that are currently known and used to represent the Delaunay triangulation are ephemeral, that is, it is not feasible to perform operations or queries on the historical data of the triangulation at a certain time t, nor operations to perform changes in structure in a past time. However, there are currently methods for maintaining versions of some trivial data structures such as lists, stacks, queues, and trees [3]. These methods have the ability to maintain information from past versions, minimizing the amount of RAM memory, and maintaining the algorithmic complexity of its operations.

In this sense, in this dissertation, a new partially persistent and fully persistent data structure is proposed and implemented, which can query and execute operations (at any previous time t) minimizing computational cost and use of RAM memory.

Propose and implement a persistent Delaunay triangulation data structure, so that it is possible to query and modify previous versions, minimizing computational and spatial complexity.

1.4. THE CONTRIBUTIONS OF THE PROBLEM

The contributions of this dissertation are: A new algorithm for the generation of partially persistent and fully persistent versions of persistent Delaunay triangulation is proposed, maintaining the performance in memory and the algorithmic complexity in each operation.

1.5. THE LAYOUT OF THIS DISSERTATION

This study will be presented in five sections. Section 1 includes the general introduction of the problem and the motivation for the work. Section 2 contains a literature review on some definitions and on the previous related works, including types of persistence and mechanisms designed to maintain persistence in some basic data structures. Section 3 includes a description of algorithms proposed for the implementation of persistent Delaunay triangulations. Four algorithms, Bowyer-Watson, Walk, Hybrid, and Graph is proposed for the partial persistence insert operation. The vertex removal operation is also proposed for the Walk method. Subsequently, the Walk proposal is extended to create a fully persistent Delaunay triangulation. Section 4 contains a description explains the experimental results performed during the implementation. First, it is focusing on constructing a planar Delaunay triangulation that is modeled after the Bowyer-Watson algorithm. It also focuses on analyzing the temporal and spatial complexity of some methods (Walk, Hybrid, and Graph) that contrast to the CGAL library. Finally, the conclusions and recommendations for future work are presented in Section 5. Besides these five sections, one appendix was added to include the details of the algorithms which are using to obtain the results.

2. REVIEW OF LITERATURE

2.1. GENERAL INTRODUCTION

Delaunay triangulation is one of the most interesting triangulations because it is applicable for solving a multitude of apparently unrelated problems, due to its geometric properties, and because it has quite efficient algorithms for its calculation. All this also implies that there is a large amount of written material on this type of triangulation. Giving an idea of what this triangulation might look like could serve as an informal definition and could be done highlighting its most important characteristics: In a Delaunay triangulation. First, all the points are connected to each other and form as many triangles as possible without crossing their edges (essential for it to be a triangulation). Second, triangles are defined so that the closest points are connected to each other by an edge. This implies that the triangles formed are as regular as possible, that is, that their minor angles are maximized, and the length of their sides is minimized. These properties make this triangulation, as indicated at the beginning, interesting in several fields. Delaunay triangulation can be characterized in several ways, below some definitions that satisfy its properties.

The purpose of this section is to conduct a comprehensive literature review of previous and related research on the subject of Delaunay triangulation and persistent data structures. Some definitions will describe, and types of persistence and mechanisms developed to maintain persistence in some basic data structures will be explained.

2.2. DEFINITIONS AND CONCEPTS

The definition of the concept can be explained in the following.

2.2.1. Triangulations. Triangulation can be defined as a collection of triangles. Definition 1 represents the meaning of the tringle.

Definition 1: Triangle t in the plane can be defined by three points $p_1, p_2, and\ p_3, where\ (p_i = (x_i, y_i))$. Every two points has an edge connected between them, denoted by $e_{i,j}$ as shown in Figure 2.1 [9]. The tringle t obtained from the intersection of three half-planes, as shown in equation 1 [9]:

$$t = H_{1,2} \cap H_{2,3} \cap H_{3,1} \tag{1}$$

There are important properties that need to be checked for constructing triangulations, some of these properties are show below.

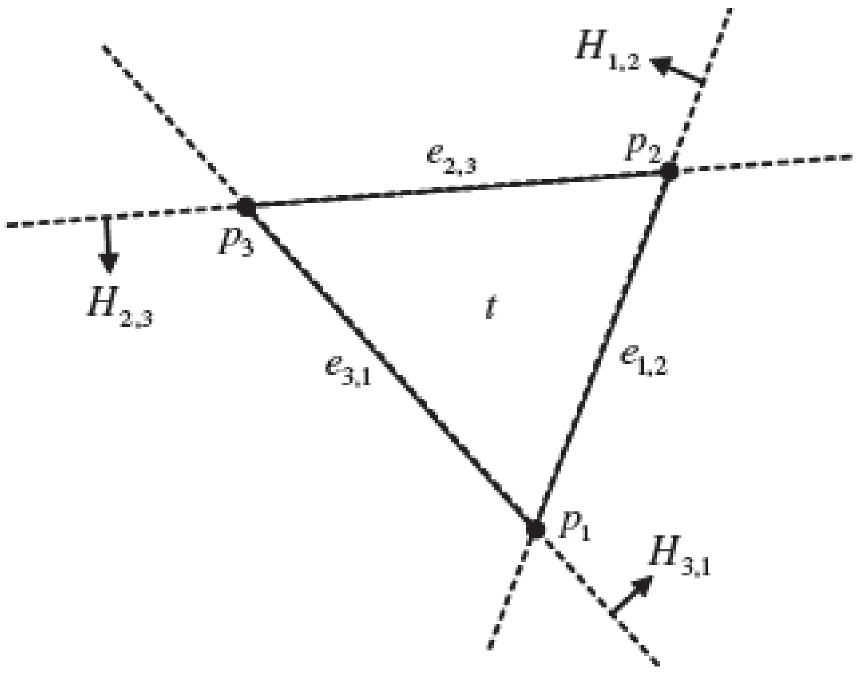

Figure 2.1 Example on Tringle That Obtained from Half-Planes.

1. The circumcircle property: This property represents a circle that drown through the tringle three points ($p_1, p_2, and\ p_3$). A point C is in the center of this circle in the plane that has an equal distance between the tringle three points ($p_1, p_2, and\ p_3$), as shown in Figure 2.2.

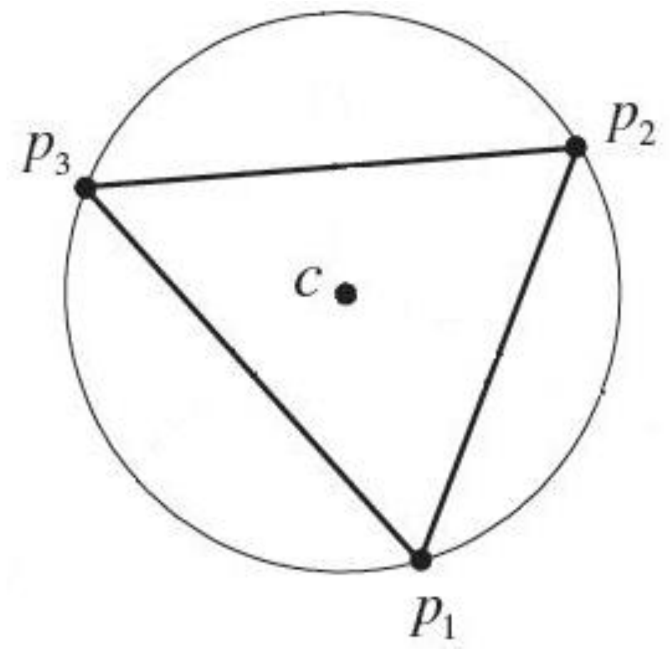

Figure 2.2 A Triangle with Circumcircle Property.

2. The triangle angle property: Calculating the angle of the tringle is another important property for the triangulations. In this work, we will be interested in the smallest angle in the triangle. The equations that will be used are the sines and cosines. Equations (2) explain an equation for getting the angle in a given triangle [9, 10]. Where $a, b, and\ c$ are represent the length of the edges. Also, the angles of the tringle are represented by $\alpha, \beta, and\ \gamma$, Figure 2.3.

$$\frac{sin\,\alpha}{a} = \frac{sin\,\beta}{b} = \frac{sin\,\gamma}{c} \tag{2}$$

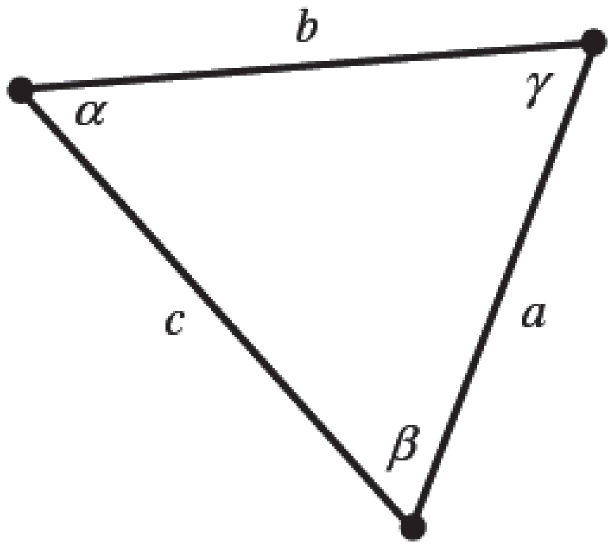

Figure 2.3 A Triangle with Edges and Angles.

3. The circle center property: The center of the circle can be found by intersected lines that split the angles. By using the corner points ($p_1, p_2, and\ p_3$) we will be able to calculate the area of the tringle, this can be seen in equation (3) [9, 11]:

$$A(p_1, p_2, p_3) = \frac{1}{2}\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = \frac{1}{2}(x_1y_2 + x_2y_3 + x_3y_1 - x_1y_3 - x_2y_1 - x_3y_2) \tag{3}$$

Each point p in the plane can be presented as a linear combination of the vertices (p_1, p_2, p_3) of the triangle, as shown in Equation (4):

$$p = b_1p_1 + b_2p_2 + b_3p_3 \tag{4}$$

To constructing triangulation for a given set of points $P = \{p_i\}, where\ i = 1, \cdots, N$, with the domain Ω. Where the boundary of Ω is closed simple polygons that is a polygon which is not self-intersect, as shown in Figure 2.4. Such that Ω is represents the convex hull (Definition 2) of the point set.

Definition 2 (Convex hull): The smallest convex set that containing P is called a convex hull of P. A set S is convex if any line segment joining two points in S lies entirely in S. Figure 2.5 [12], show an example of a convex domain Ω for asset of points in the plane with a corresponding triangulation of these points [13].

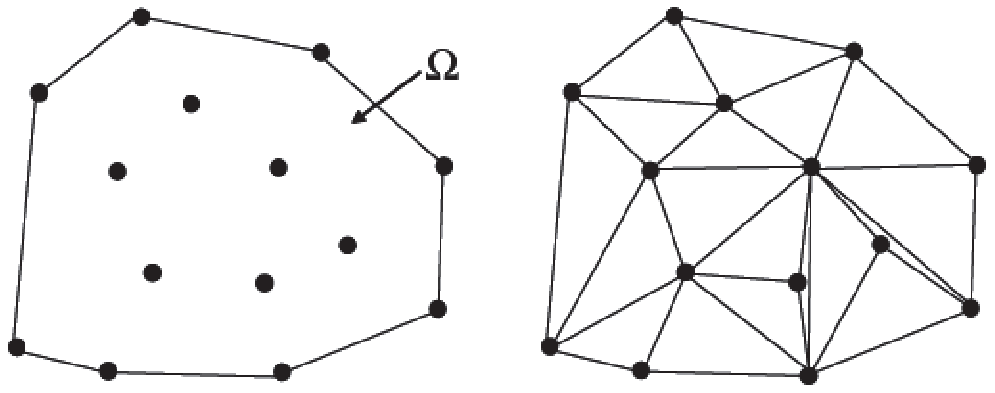

Figure 2.4 An Example of Convex Hull with Domain Ω.

It is important to find an appropriate representation of a triangulation for all the computational applications, that will need a suitable data structure to use it in computing programs. It is important to find an appropriate representation of a triangulation for all the computational applications, that will need a suitable data structure to use it in computing programs. Triangulations are used in different sciences and different applications so in this work we present the triangulations with different types of data structures.

There are many interesting properties that are obtained from planar graphs properties as shown in elementary graph theory [14]. The size of triangulation is very important for computer programing and designing data structure. Below we will explain the relationship between the numbers of the triangles, edges, and vertices.

The relationship between the numbers of the triangles, edges, and vertices is important [15]. Suppose faces (triangles) are denoted by F and suppose edges are denoted by E, where the vertices are denoted by V. In the triangulations $V = V_B \cup V_I$, where V_B represents the boundary vertices and V_I represents the interior vertices. Equation (5), shows the relationship between triangles, edges, and vertices as presented in [9].

$$|F| = |V_B| + 2|V_I| - 2$$

$$|E| = 2|V_B| + 3|V_I| - 3 \tag{5}$$

In Equation (6), we can see a special type of the Euler Polyhedron Formula that also named as Euler-Poincare formula.

$$|F| = |E| - |V| + 1 \tag{6}$$

Equations (6) was obtained from Equation (5). The numbers of triangles and edges are fixed when the boundary of the triangulation is specified, and that happened by using Equation (5). Furthermore, the number of triangles is increased by two and the number of edges is increased by three, after inserting a point to an existing triangulation.

One other triangulation property is the degree $\left(deg(v_i)\right)$, it represents the number of the edges that is happening with the vertices vi in the tringles. The summation of these degree for the triangulation is satisfy Equation (7), [9].

$$\sum_{i=1}^{|V|} deg(v_i) = 2|E| \tag{7}$$

The number of vertices in the boundary is smaller than the total number of vertices when building a triangulations [16] from a large number of points such that the boundary of tringles is the convex hull of the points [9], as shown in Equation (8):

$$\sum_{i=1}^{|V|} deg(v_i) = 2|E| = 6|V_I| + 4|V_B| - 6 \approx 6|V| \quad (8)$$

Depends on what we said, we obtained that the average number of edges that happening with a vertex in a triangulation is six.

2.2.2. Delaunay Triangulations and Voronoi Diagrams. In computational geometry, there are two important constructs the Delaunay triangulation, and the Voronoi diagram. These are used in different applications, and studies for many years.

Voronoi diagrams have a close relationship with Delaunay triangulations. The first investigated for the Voronoi diagrams was made by René Descartes [17], and it used by Dirichlet for the quadratic forms. Voronoi was a Russian mathematician well known in number theory and his contributions with respect to continued fractions.

Definition 3: Suppose P is a set of points were $P = \{p_1, \cdots, p_N\}$ in the plane. Let the Euclidean distance denoted by $d(p_i, p_j)$, it is between p_i and p_j. The Voronoi region in the plane is shown in Equation (9) [9]:

$$V(p_i) = \{x: d(x, p_i) < d(x, p_j), where\ j = 1, \cdots N\} \quad (9)$$

Figure 2.5 [9] shows a Voronoi region point p_i. Voronoi diagrams have been used in many scientific corrections to organize convergence information between points distributed irregularly. For example, the climate scientist Thiessen used the regions of Voronoi to collect climatic data from unevenly distributed weather stations [18]. For this reason, the regions V (pi) are also called Thiessen regions.

Boris Nikolaevich Delaunay [19] presented the Delaunay triangulation of a point set P. It satisfies the property that says, for a set of points S there is no point inside the triangle's circumscribing disk.

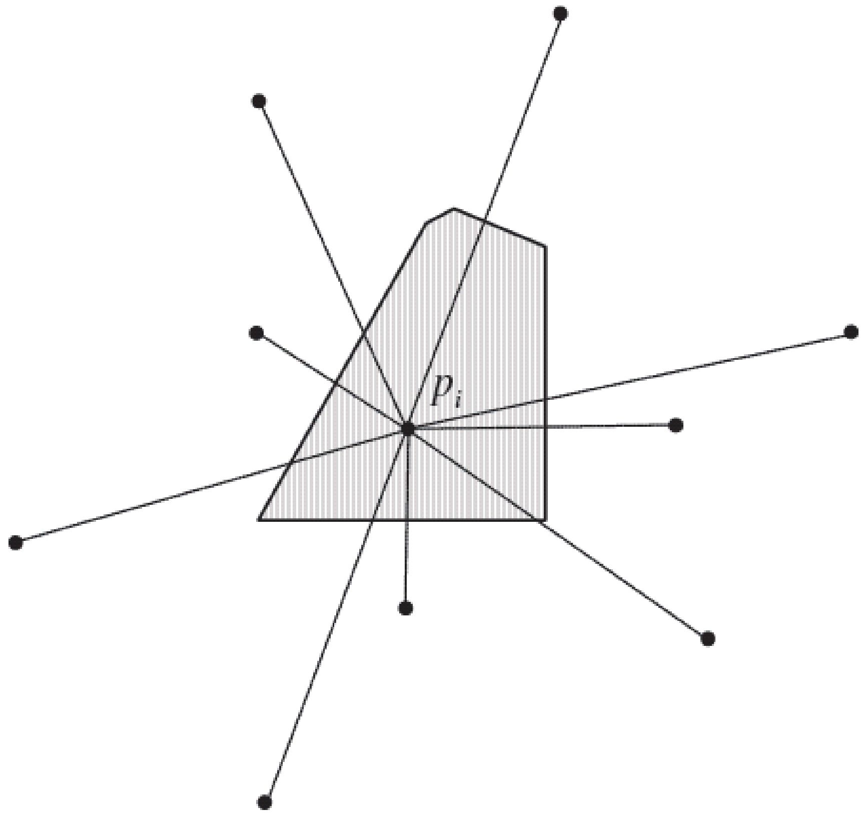

Figure 2.5 The Voronoi Region in the Plane.

Definition 4: A Delaunay triangulation for a set of points P in the plane is the triangulation operation which satisfies the empty circumcircle condition. That is means there is no point inside the circle from the set P.

This definition is more useful and geometric to use in a practical. In this work, this definition will use as a rule to construct a Delaunay triangulation algorithms.

Figure 2.6 shows an example of a Delaunay triangulation [20] for a set of six points. The five triangles satisfy the empty circle property. But the orange circle it not empty since it is not a circumcircle for any of the five triangles.

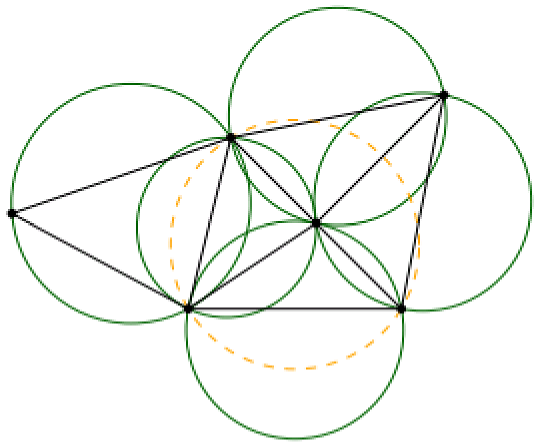

Figure 2.6 An Example of Delaunay Triangulation.

2.2.3. Persistent Data Structure. These data structures are subject to are not change effectively, as its operations do not (clearly) update the structure in place, but instead, always provide a new updated structure. These data structures are subject to are not change effectively, as its operations do not (clearly) update the structure in place, but instead, always provide a new updated structure. There are different types of persistent data structure, the partially persistent, confluently persistent, and the fully persistent data structure.

The persistent data structure [21] is called partially persistent if all the versions are able to access but the only version that can be modified is the newest version. The persistent data structure is fully persistent if all versions can be accessed and modified [22]. Confluently persistent is another type of persistent data structure, it is named confluently if all the operation can be merged to get a new version of the data from the previous versions. Were the ephemeral data means that the data structure is not persistent. Most of these results use construction, with the exception of the article by [23] and by [22].

2.3. DELAUNAY TRIANGULATIONS

This section describes what is a Delaunay triangulation, as well as its importance.

2.3.1. Triangulation for a Set of Points in the Plane. From a set $P = \{p_1, p_2, \cdots, p_n\}$ made up of n vertices in the plane, a triangulation T of P is defined as a planar maximum subdivision where no edge connecting two vertices can be added without change its planarity [24], in addition, there is always a triangulation of P since any polygon can be triangulated [25], on the other hand, the edges of the convex envelope [26] of P they belong to any triangulation T that is made up of P and that the limitless or infinite face infinite is always a compliment to the convex envelope.

By Theorem 1, the number of triangles is the same in any triangulation of P, as well as the number of edges, in addition, these depend on the number of vertices in P that are in the limit of the convex envelope of P (including the vertices that are in some edge of the convex envelope).

Theorem 1. Let P be a set of n vertices in the planar, not all collinear, and k denotes the number of vertices in P that are at the limit of the convex hull of P. So, any triangulation of P has $2n - 2 - k$ triangles and $3n - 3 - k$ edges [24].

If m is the number of triangles that make up T, any other triangulation of P will also be made up of m triangles, considering the $3m$ angles of the triangles of T, ordered by their value in increasing order, such as $\alpha_1, \alpha_2, \cdots, \alpha_{3m}$, therefore, $\alpha_i \leq \alpha_j, where\ i < j$, is called $A(T) = (\alpha_1, \alpha_2, \cdots, \alpha_{3m})$ the vector angle of T.

Let T' be another triangulation of the same set of vertices P, and let $A(\acute{T}) = (\acute{\alpha}_1, \acute{\alpha}_2, \cdots, \alpha'_{3m})$ its vector angle, the vector angle of T is greater than the vector angle of T' if $A(T)$ is lexicographically greater than $A(T')$, that is, there is an index $1 \leq i \leq 3m$ such that $\alpha_j = \acute{\alpha}_j$ for all $j < i$ and $\alpha_i > \acute{\alpha}_i$, which is denoted as $A(T) > A(T')$, in addition **an optimal angle** of triangulation T can also be called if $A(T) \geq A(T')$ for all triangulations T' of P. To see if triangulation has **an optimal angle**, Theorem 2 is used, it called Thales's Theorem, where the smallest angle defined by three vertices p, q, r is denoted by $\measuredangle pqr$.

Theorem 2. Let C be a circle, as shown in Figure 2.7, l a line intersecting C at vertices $a\ and\ b$, let $p, q, r\ and\ s$ be vertices that are on the same side of l. It is assumed that $p\ and\ q$ are found in C, and that r is inside C, and that s is outside C. Then:

$$\measuredangle arb > \measuredangle apb = \measuredangle aqb > \measuredangle asb \tag{10}$$

From Theorem 2, an edge $e = p_ip_j$ of a triangulation T of P is considered that is not an edge that is in the convex hull, that is, it is incident to two triangles $p_ip_jp_k$ and $p_ip_jp_l$. If these two triangles form a convex quadrilateral, we can obtain a new T' triangulation by removing p_ip_j from T and inserting p_kp_l in its place. This operation is called a flip [27].

The only difference in the vector angle of T and T' is the six angles $(\alpha_1, \alpha_2, \cdots, \alpha_6)$ in $A\ (T)$, which are replaced by $(\acute{\alpha}_1, \acute{\alpha}_2, \cdots, \acute{\alpha}_6)$ in $A\ (T')$. Figure 2.8 shows this behavior. On the other hand, said edge (p_i, p_j) is called illegal if Equation 2 is fulfilled, that is if by performing the flip operation the value of the smallest angle can be increased.

$$\min_{1 \leq \iota \leq 6} \alpha_\iota \acute{<} \min_{1 \leq \iota \leq 6} \acute{\alpha}_1 \tag{11}$$

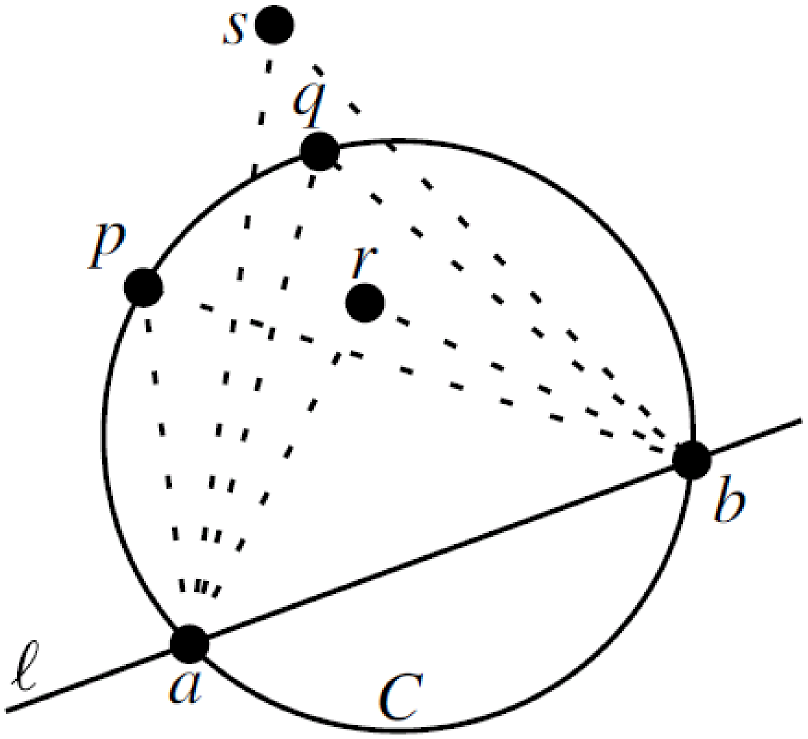

Figure 2.7 Thales's Theorem.

Otherwise, the following lemma can be considered in order to corroborate if an edge is illegal.

Lemma 1: Let the $p_i p_j$ edge be incident to the triangles $p_i p_j p_k$ and $p_i p_j p_l$ and let C be the circle that passes through $p_i p_j p_k$; then the $p_i p_j$ edge is illegal if and only if the vertex pl is inside C. On the other hand, if the vertices $p_i p_j p_k p_l$ form a convex quadrilateral and are not in a common circle, so $p_i p_j$ or $p_k p_l$ is an illegal edge.

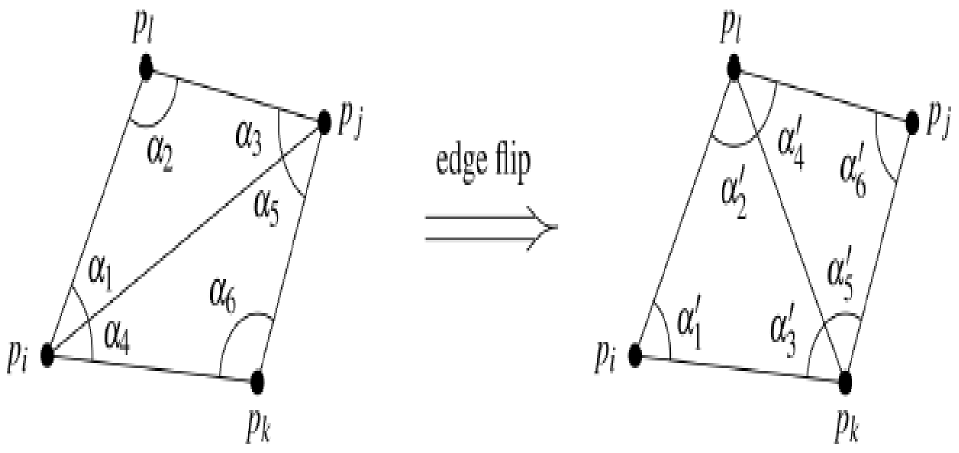

Figure 2.8 The Flip Edge Operation.

Also, when the four vertices meet in a circle, both p_ip_j and p_kp_l are legal. On the other hand, the two triangles incident to an illegal edge must form a convex quadrilateral, so it is always possible to flip an illegal edge.

In theory, a legal triangulation is defined as a triangulation that does not contain any illegal edge, also from Equation 2 it is concluded that any optimal angle in the triangulation is legal. Furthermore, a Delaunay triangulation is always legal as shown in Theorem 3.

Theorem 3. Let P be a set of vertices in the planar. A triangulation T of P is legal, if and only if, T is a Delaunay triangulation of P.

Since any an optimal angle triangulation must be legal, Theorem 2 implies that any optimal angle triangulation of P is a Delaunay triangulation of P. When P is in a general position (P is in general position if it contains no 4 points on a circle), there is only one legal triangulation, which is then the only optimal angle triangulation, that is, the only Delaunay triangulation. When P is not in general position, there are many legal triangulations of P, not all these Delaunay triangulations will be of optimal angle, however, all these triangulations will have the same value for the minimum angle of their angle-vectors, this can be seen in theorem 4 and be verified by Thales' theorem.

Theorem 4. Let P be a set of vertices in the plane. Any angle-optimal triangulation of P is a Delaunay triangulation of P. In addition, any Delaunay triangulation of P maximizes the minimum angle over all triangulations of P.

From the above, to obtain a Delaunay triangulation of T from any triangulation T' both triangulations of the set of vertices P, It is enough to verify if all edges are legal and if not, perform flip operations on illegal edges, until there are no illegal edges in T'.

2.3.2. Constrained Delaunay Triangulations. In some cases, it is necessary to force the creation of certain edges within a triangulation, these are called Constrained Delaunay Triangulations [28].

2.3.3. Algorithms to Construct Delaunay Triangulations. The duality between Delaunay triangulation and Voronoi diagrams is widely known [29], [30], and, therefore, the algorithms for the construction of Voronoi diagrams serve to calculate Delaunay triangulation. However, direct construction methods are generally more efficient because the Voronoi diagram does not have to be calculated and stored. Algorithms for constructing Delaunay triangulations [6], [31] can be classified as follows:

2.3.3.1. Local improvements algorithm. From an arbitrary triangulation, these algorithms locally modify the edges of the adjacent vertex pairs through flip edge operations until obtaining a Delaunay triangulation.

2.3.3.2. Incremental insertion algorithm. These algorithms insert the vertices one at a time. The triangle that contains the new vertex is divided by inserting new triangles adjacent to the new vertex. Then, the circle criteria are tested on all adjacent vertices and, if necessary, the flip operation is applied to the edges.

2.3.3.3. Gradual construction algorithms. These algorithms transform the vertices in the space of $d\ to\ d\ +\ 1$ dimensions and then calculate the convex hull of the transformed vertices. Delaunay triangulations is obtained with the projection of the convex hull in d dimensions [32].

2.3.3.4. Divide and conquer algorithm. Divide and conquer: They are based on the repeated partition and the local triangulation of a set of vertices and then carry out a combination phase, where the resulting triangulations are joined.

2.4. PERSISTENT DATA STRUCTURES

A data structure is persistent if it can maintain versions of itself as a result of operations such as insertion, deletion or modification, which change the structure over time [1].

For example, in the case of a binary tree, when performing an operation on the structure, it increases, decreases, or changes the keys of the nodes according to the type of operation: insertion, deletion or modification. However, in order to know the state of the structure in a past time t, it would be necessary to store each version before a modification, the simplest way would be to save the entire data structure in RAM before undergoing a modification. In this way, there would be n different copies of the structure, one for each modification in time, so that recovering the state of the structure in a time t, would imply instantiating a copy r, in that time t. However, this would imply sacrificing memory space.

In this sense, persistent data structures seek to develop strategies to minimize the space required by the new versions that are generated over time.

2.5. TYPES OF PERSISTENCE DATA STRUCTURES

The data structures can be classified as follow.

2.5.1. Partial Persistence Data Structures. A data structure is partially persistent, it does allow modifications only to the current data structure, but it is possible to query any previous version and at any past time t, [33].

These versions can be accessed because each node stores additional information. This extra information refers to the creation time of that version, which indicates, to the time where modification was made. The time, of each version, is known as a timestamp (It is a sequence of characters information identifying when a certain event occurred, usually giving date and time of day, sometimes accurate to a small fraction of a second). An example of the behavior of this structure can be seen in Figure 2.9, where each node represents a version of the data structure.

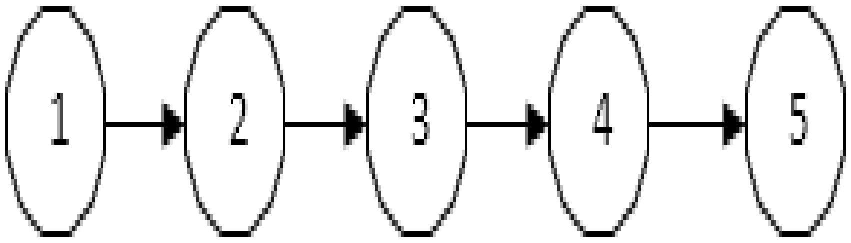

Figure 2.9 Example of Partial Persistence.

2.5.2. Full Persistence Data Structures. With this type of persistence data structures, the versions do not form a simple linear trajectory, furthermore, they form a tree version [33], as explained in Figure 2.10.

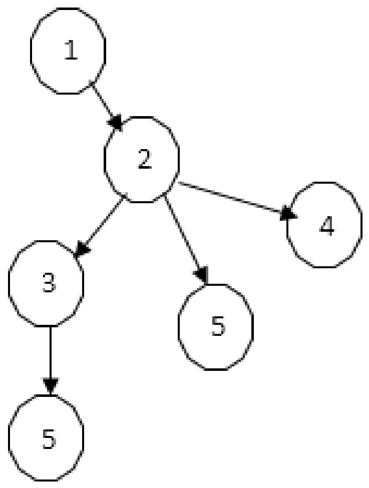

Figure 2.10 Example of Full Persistence.

2.5.3. Confluent Persistence Data Structures. This type of persistence allows to modification and querying the versions in the past time. It also allows joining two or more previous versions to create a new version. This version takes the form of a DAG (direct acyclic graph), which can be seen in Figure 2.11.

Figures 2.9, 2.10, and 2.11 are shows the different types of persistence. In Figure 2.9 partial persistence can be seen. The first nodes are earlier versions, while the last node is the only one that can be modified, in this way, a linear structure of persistence can maintain. In Figure 2.10 a full persistent data structure can be seen. This structure can modify any of the versions in time and, therefore, the versions make up a tree so that any version in time can be modified. Where in Figure 2.11 shows the confluent persistence that presents a group of versions induce a DAG structure on the version graph.

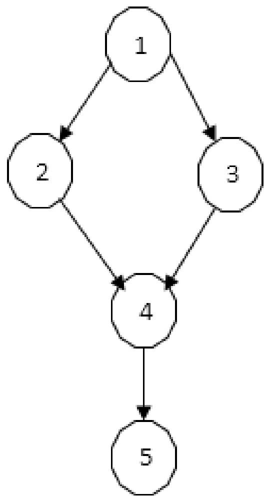

Figure 2.11 Example on Confluent Persistence.

Considering that the obvious way to provide persistence is to make a copy of the data structure every time a modification operation is performed on the structure. Using this way would have the disadvantage of requiring space and time proportional to the space occupied by the original data structure. For that, a method is required to minimize spatial and temporal complexity.

2.6. METHODS TO CREATE A PERSISTENT DATA STRUCTURES

Various methods have been proposed in the literature to make a persistent data structure [22], below are general schemes to convert an ephemeral data structure into a persistent data structure.

2.6.1. Fat Nodes. In this scheme, all changes made to the node fields are recorded without deleting the old values of the fields. This makes the nodes of the structure called Fat Nodes. Where each one contains the same information fields and pointers as an ephemeral node, but they keep extra space to add additional values such as old data and timestamp in which modifications were made [1].

Each additional data has an associated value and a timestamp that indicates the version in which the field was changed by a specific value. In addition, each Fat Node has its own timestamp, which indicates the version in which the node was created. Figure 2.12 [3] shows Fat Nodes in a simple linked list. Each node has, in addition to its data and pointers, an additional field which stores new versions.

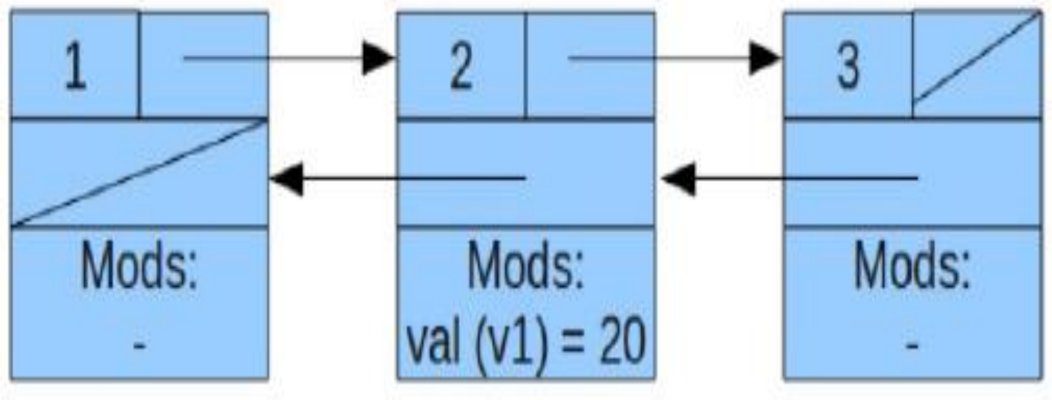

Figure 2.12 Fat Node Example of a Simple Linked List.

On the other hand, because Fat Nodes only have additional space limited to a fixed number of modifications, then, in the case that a node has all its information fields filled, it is necessary to create a new node. This new node must keep pointers to the previous and next node in the list. In addition, it must be pointed by the previous and next nodes, without losing the address of the previous version node.

2.6.2. Path Copying Method. Path Copying is a method that consists of making a copy of all the nodes of the route, which contains the node to be modified. This operation is performed in cascade to modify all those nodes so that all the nodes that pointed to the old node should now be modified to point to the new node.

These modifications generate changes in a cascade until reaching the root of the tree. Which makes each version have its own root, and in that sense, a series of roots indexed by time must be maintained as shown in Figure 2.13. The data structure pointed to the root of time t is exactly the structure time t [1].

There are more complex ways to make persistence, such as confluent persistence [34], [35]. This structure allows the three versions of a data structure to be joined in a new update operation. This research focuses on partial, full, and confluent persistence.

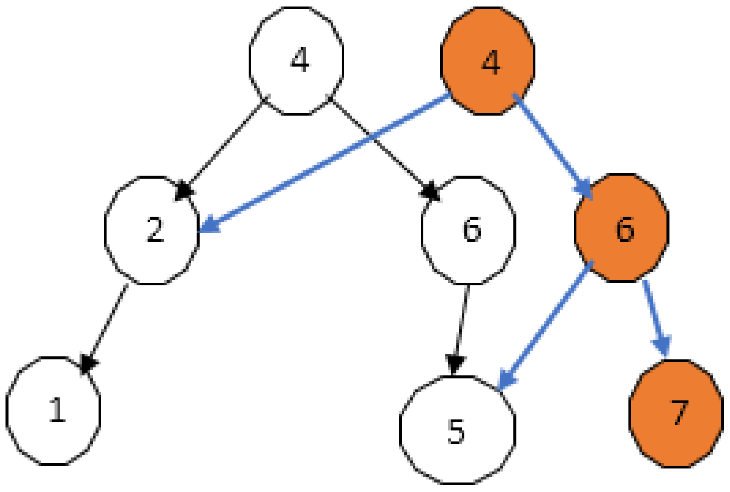

Figure 2.13 Path Copying in a Binary Search Tree.

2.7. THE INCREMENTAL ALGORITHM

A Delaunay triangulation of a set of vertices P is calculated by inserting one vertex after another, that is, the triangulation of the set of vertices P inserted until R is maintained, where $R \subseteq P$. For the insertion of the next vertex s, the triangulation is updated to obtain the Delaunay triangulation of $R \cup \{S\}$.

To avoid special cases, three artificial vertices (a triangle) are added to the vertex set P that must wrap around all the vertices, that is, the convex envelope of the resulting vertex set is a triangle. The incremental algorithm begins with the Delaunay triangulation of the three artificial vertices. At each insertion of a vertex s, the triangle $\triangle = \triangle (p, q, r)\ for\ DT(R)$ containing s must be found and replaced by the three resulting triangles connected to the three vertices p, q, r as seen in Figure 2.14 [36].

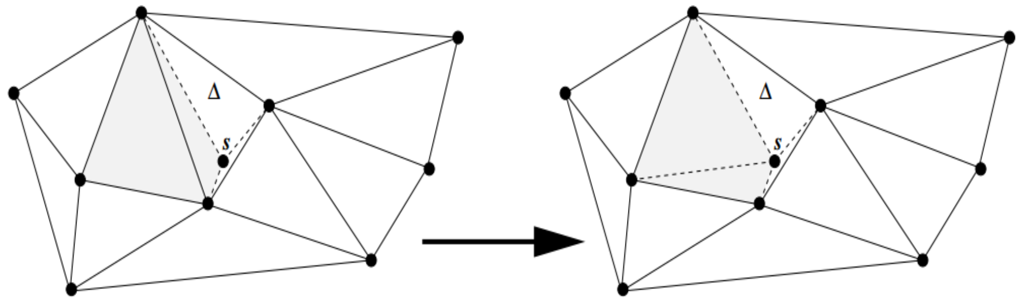

Figure 2.14 Incremental Insertion Algorithm.

To find the triangle that contains the vertex to be inserted, different algorithms and/or data structures are proposed:

1. Data structures for inquiries from the nearest neighbor [37].
2. Algorithms for walking in triangulations [38].

Given the triangulation T of $R \cup \{s\}$, flips are performed in T until obtaining $DT(R \cup \{s\})$ is obtained as in Figure 2.15.

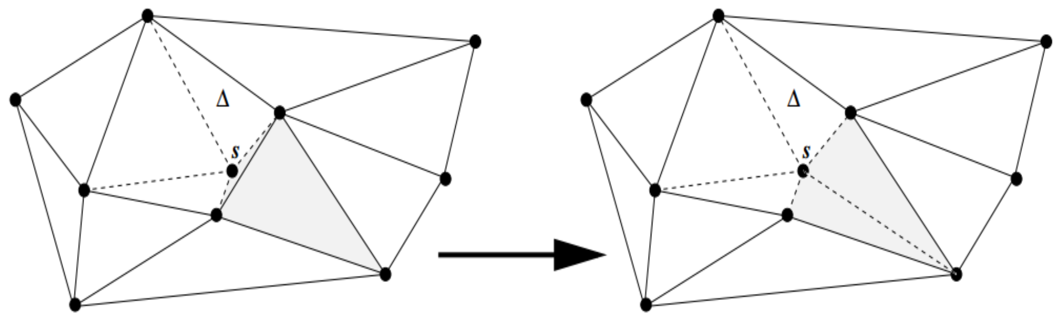

Figure 2.15 Incremental Insert Result.

Each incident s edge that is created during the update is an edge of the Delaunay graph of $R \cup \{s\}$ and therefore an edge that will be in $DT(R \cup \{s\})$.

2.8. RELATED WORKS

The use of persistent data structures is involved in many areas of computer science such as functional programming, computational geometry, and other algorithm application areas [1], most of these applications require being able to perform queries in previous states of the data structure.

There is a simple scheme to make any data structure persistent. In this scheme, the operations are carried out as they have been ephemerally but before each update operation, it is performed in the current version. New copies are created on which will perform the required operation, keeping both versions in memory independently of each other. This is obviously inefficient since the consumption of time and space is equal to the number of elements of each version.

In general, persistent data structures are divided into two types: The first, seen in the previous section, attempts to create general ways that would make any ephemeral data structure persistent while keeping computational and spatial costs low. The second, which presents specific solutions to generate persistent data structures, some of which are described below.

2.8.1. Location of Vertex in the Plane. "One of the best-known geometric applications is the algorithm for the location of a vertex in the plane that triggered the development of the entire area. In the problem of locating a vertex in a plane, we are given the subdivision of the Euclidean plane into polygons of n line segments that intersect only at their ends " [39].

First, processing must be done on the line segments, so that given a query vertex p, it is possible to efficiently determine which polygons contain it. The plane is divided into vertical stripes by drawing a vertical line that passes through each vertex, that is, the intersections of the line segments in the subdivision of the plane [1], the intersection of the line segments with subdivision of a fragment are arranged, then it is possible to answer each query using two binary searches. One binary search locates the fragment containing the query, and another binary search locates the segment before the query vertex within the fragment.

If we have the set of a specific segment, we can obtain the set of segments by removing the segments that end at the limit between them and inserting segments that start at that limit. As all the segments are swept from left to right, we have that in total there are n eliminations and n insertions; one deletes and one insert for each line segment, thus reducing the problem of keeping partially persistent search trees to query.

2.8.2. The Persistent Arrays. Dietz's article [40] shows a general technique for creating persistent arrangements, which takes time $O\ (loglog\ (m))$ to access the arrangement and $O\ (loglog\ (m))$ amortized [41] to change the content of an entry, where m is the total number of changes being the size of space used m.

Otherwise, n indicates the size of the array and it is assumed that $n\ < m$, the array is considered a Fat Node with n fields. The list of values-versions that describe the assignments to each entry in the array is represented in a data structure proposed by [42]. This data structure has spatial complexity linear to the number of elements, making use of perfect dynamic hashing [43].

2.8.3. Persistent Triangulations. Blelloch [44] describes a method to construct Convex Hull in 3 dimensions, to subsequently obtain the Delaunay triangulation from the edges of the convex envelope, to obtain the Delaunay Triangulation partially persistent, information is maintained about the edges, which they are stored in data structures with an optimal query time, in particular, making use of balanced search trees, to be able to query the data structure at an earlier time t.

Many algorithms, including the one proposed by Blelloch [44], are based on the construction of the convex polygon in the dimension $d\ +\ 1$. That is, given a vertex p outside the convex polygon of a set of vertices, said polygon can be extended to include this vertex as follows. Consider the vertex p outside as a subset of the faces for the polygon. These faces are eliminated in the construction.

The boundary of these faces is a set of edges called the horizon, then a pyramidal polyhedron is created as in Figure 2.16 [45], whose vertex is the outer vertex and whose base is the horizon. This construction the vertex p as a New vertex of the related component, the inner vertices to the polygon, are discarded.

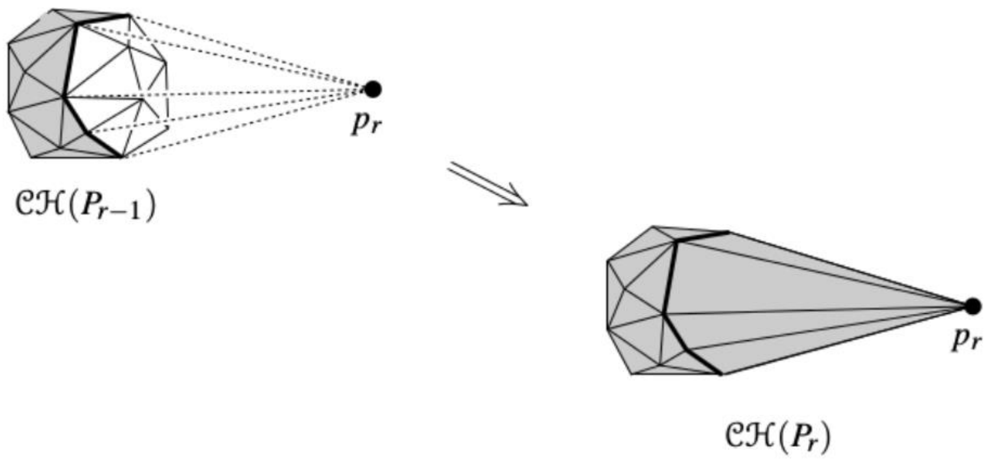

Figure 2.16 The Incremental Algorithm, Convex Polygon in 3 Dimensions.

Otherwise, to obtain Delaunay Triangulations partially persistent, information is kept about the adjacencies, which are stored in data structures with an optimal query time, in particular the use of balanced search trees.

3. PERSISTENT DELAUNAY TRIANGULATIONS

This section explains the algorithms proposed for the implementation of persistent Delaunay triangulation. Four algorithms, Bowyer-Watson, Walk, Hybrid and Graph are proposed for the insertion operation in partial persistence, the vertex removal operation is also proposed for the Walk method. Later the Walk proposal for the full persistent Delaunay triangulation is extended.

3.1. THE WALK METHOD

The proposal consists of a structure, which is divided into two fundamental parts:

1. The first corresponds to the data structures, which will support the persistent Delaunay triangulation and whose stage we will call the internal structure.
2. The second describes the persistent insertion and deletion that will support the proposed Delaunay triangulation.

3.1.1. Internal Structures. The internal structure, unlike libraries such as CGAL, considers the persistence of previous versions. It allows storing previous states, which were generated as a result of some modification operation. The storage cost of previous versions in the internal data structure will minimize the temporary cost and maintain computational complexity.

One of the simplest, but least efficient ways of maintaining versions would be by storing full copies of all past versions and considering that every time an insertion or deletion operation is performed. A new state or version is generated, then implementing this method would be counterproductive in time and space.

In this sense, the internal structure aims to minimize the amount of memory necessary to maintain the versions of the structure and, in addition, maintain the algorithmic complexity in each of the operations. Considering that each operation of insertion or elimination, in the Delaunay triangulation, modifies the edges of one or several vertices. Then, it is proposed to use an additional data structure, which will allow storing the changes, only of the edges and the affected vertices in a certain time t.

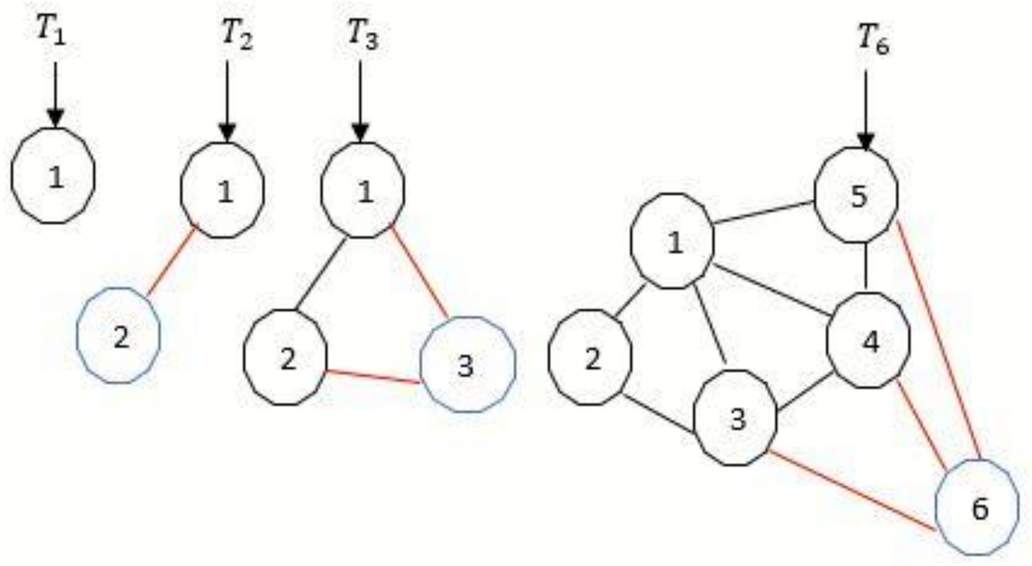

Figure 3.1 The Triangulation Over Time.

In Figure 3.1, the changes in triangulation overtime after inserting operations can be seen. Where $T1$ represents the initial Delaunay triangulation and for the other triangulations, Ti represents the Delaunay triangulation of $T_{i-1} \cup \{vertex\ i\}$. In the same figure, the black nodes represent vertices that were entered in a past time and, on the other hand, the blue nodes are those that are being entered in the T_i time. Each node stores information of the time T_i in which it was inserted, the edges store information of the time in which they were created, similarly the edges that connecting with the blue nodes represent the edges created in the time T_i.

The internal structure of the persistent Delaunay triangulation is made up of a set of data structures, which allow keeping the information of the changes made on the triangulation. These additional structures have the main objective to minimize the amount of RAM, which will allow storing any modification. And, to make consultations in the past of the structure, maintaining the computational cost of the regeneration of the triangulation in a previous time t.

On the other hand, it is proposed to store information about the neighborhood of each vertex over time (vertices that at some time had some edge in common). An example is observed in Figure 3.1, where each edge has an integer associated, which indicates the time in which it was created. In addition, each vertex stores a set of data, which represent the historical information about vertices with which there was some relationship (edge) in the past.

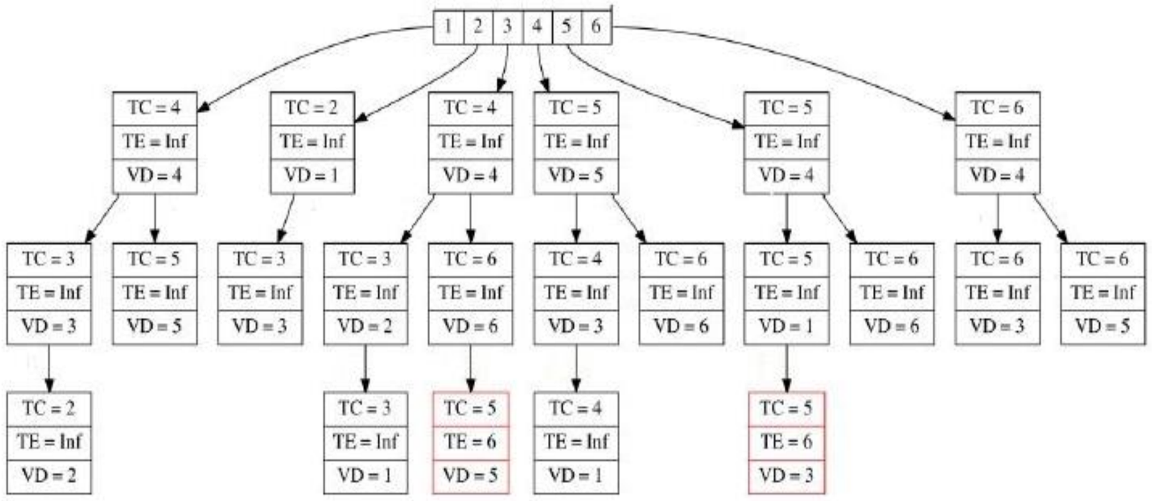

Figure 3.2 Internal Representation for the Triangulation Over Time.

In Figure 3.2 the persistent representation of Delaunay's triangulation of Figure 3.1 can be seen. Figure 3.2 shows the vertices of the Delaunay triangulation. Each vertex has a balanced binary search tree associated and, in addition, each node of the tree symbolizes an edge between the vertex p and the destination vertex q in T. Even, it maintains information of the creation time (TC) and the elimination time (TE) of the edge between $p\ and\ q$. For example, the first tree indicates that vertex 1 is connected to vertices (VD) 2, 3, 4 and 5 and the root of this tree, that is, node [TC=4, TE=Inf, VD= 4], indicates that the edge was created at time 4, which has not removed and is connected to vertex 4 (respectively).

In other words, each node of the tree has the structure [TC, TE, VD], where TC represents the creation time of the edge. TE is the elimination time of the edge. The value Inf represents the infinite value, which means that the edge has not been eliminated, and finally the adjacent vertex. An example of an edge that was eliminated in time 6 can be seen, in this same Figure 3.2, in the red nodes.

3.1.2. Insert Operations. In the present investigation, triangulation is represented as a set of vertices and faces. Each face or triangle provides access to its three vertices and its three neighboring faces clockwise cw or counterclockwise ccw. Each face has three-pointers to its three vertices and three-pointers to the three adjacent faces.

These pointers are indexed by 0,1 and 2 counterclockwise, on each face. The vertex indexed by i is opposite to the adjacent face with the same index; as shown in Figure 3.3 [7].

For the insertion operation, the incremental algorithm seen in Section 2.5 will be used. This method is used because the persistence in time requires storing information every time an operation is performed. That is, it is not feasible to insert n vertices together, as the divide and conquer method would do.

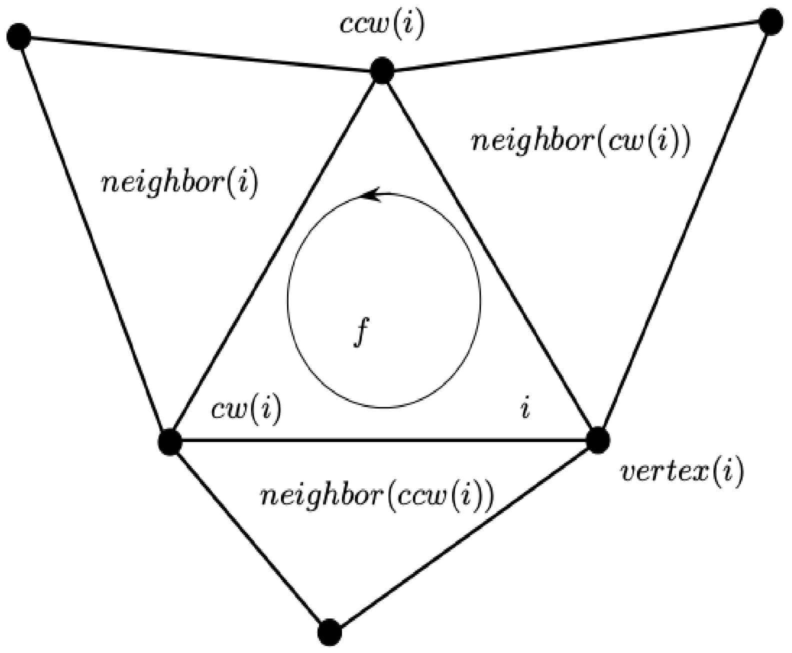

Figure 3.3 Representation of a Triangulation.

As stated in the incremental algorithm, the first step to being able to insert into the Delaunay triangulation is to find the triangle that contains the vertex to be inserted. See Figure 3.4. To find this triangle, a modification to the algorithm proposed by Brown [38] was implemented whose computational cost is $O\left(\sqrt{(n)}\right)$. This is because the algorithm had to be adapted to work in our internal data structure. Modification to this algorithm is necessary because the search for the triangle that contains the vertex depends not only on the structure but also on the version in which the vertex is to be inserted. Algorithm 1 (Appendix A), which calls PS-Search, performs this procedure.

The algorithm shows the proposed method to find a near vertex in persistent triangulation. For this, the input data are the vertex p to be inserted, the persistent data structure of the Delaunay triangulation and an integer value t, which indicates the time at which the insertion operation will be performed.

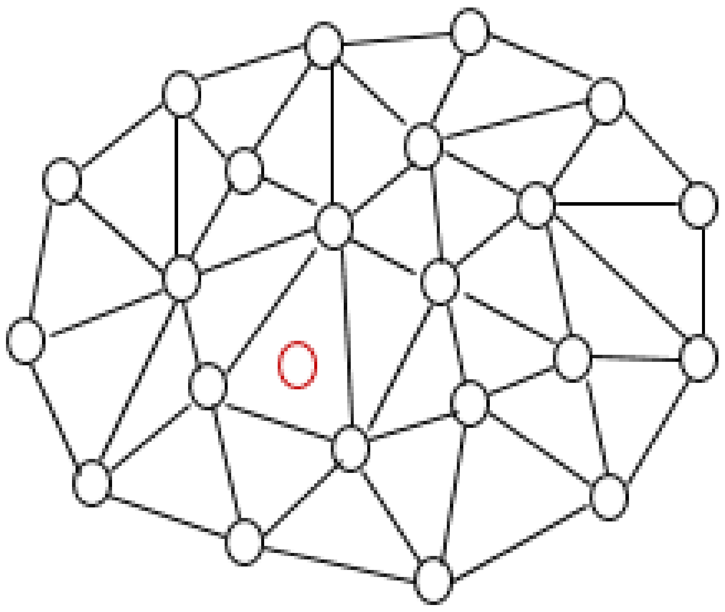

Figure 3.4 The Location of the Triangle.

The algorithm shows the proposed method to find a near vertex in persistent triangulation. For this, the input data are the vertex p to be inserted, the persistent data structure of the Delaunay triangulation and an integer value t, which indicates the time at which the insertion operation will be performed.

The computational cost required in the worst case is $\left(O\left(\sqrt{n} * k * \log(k)\right)\right)$; where n indicates the number of vertices in the entire triangulation, k represents the maximum degree of triangulation. Finally, since in practice the set of edges adjacent to a vertex v is usually very small in relation to n, then $\sqrt{n} * k * \log k \leq C\sqrt{n}$, from which it is concluded that the proposed algorithm has a computational complexity of $O(\sqrt{n})$. It should be considered that Algorithm 1 (Appendix) returns one of the edges of the triangle containing p. Next, it is necessary to find the other two vertices of the triangle containing p. This is achieved by finding the neighboring edge so that both edges form a triangle that contains p.

With the 3 vertices that make up the triangle that encloses the vertex p, the triangulation is loaded. This procedure is performed in the internal structure and loading only the triangles that will be modified, those that must be affected by a flip operation. This allows us to strictly keep the information on the changes that will be made. In Figure 3.5, only the vertices and their respective edges are marked burble, which is loaded from the internal structure. Next, 3 edges are created adjacent to the vertex to insert p with the vertices of the triangle that contains that vertex. Then, the edges of the triangle containing p are evaluated and if there is any illegal edge (that does not comply with Delaunay's condition). The flip operation on that edge must be carried out, to subsequently evaluate the 2 opposite edges.

In Figure 3.6, the purple edges that is related to the purple nodes represent the edges that will be evaluated in order to verify whether they are legal or illegal edges. If illegal, the flip operation is applied.

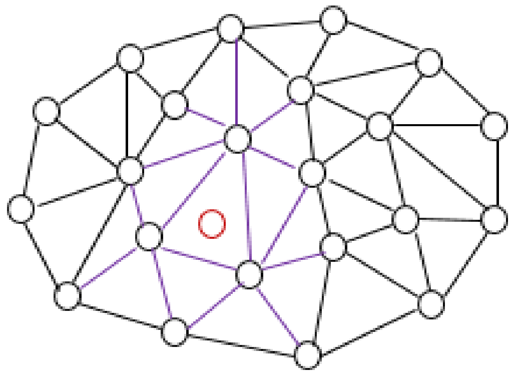

Figure 3.5 Load Triangles That Contains At Least One Vertex.

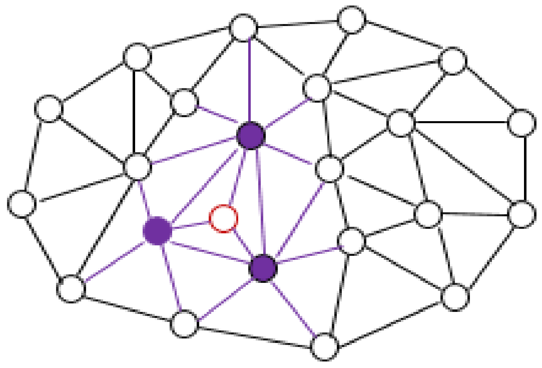

Figure 3.6 Edges to Be Evaluated to Perform Flip Operations.

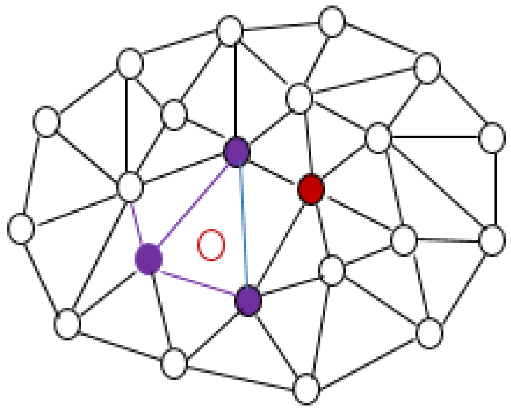

Figure 3.7 An Illegal Node Will be Loaded to Execute the Flip Operation.

In Figure 3.7, the blue edge is an illegal edge. Then, to apply the flip operation, the opposite vertex of that edge is loaded into memory. The read node, in the same figure, represents this behavior.

After applying these operations, the edges adjacent to the red node are loaded, and the algorithm continues to perform the same process as long as an illegal edge is found. Figure 3.8 shows in red, the set of edges that will be evaluated after applying the first flip. It is important to note that in the persistent Delaunay triangulation only the set of vertices and edges that will be evaluated after an insertion operation is loaded. This means that the amount of RAM is limited only by the number of vertices and edges loaded in the evaluation process.

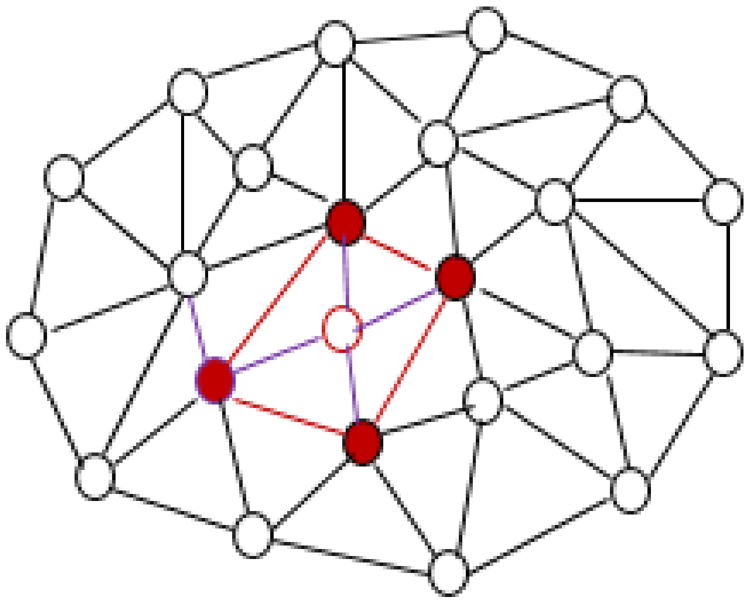

Figure 3.8 Edges to Be Evaluated to Perform the Red Flip Operation.

Algorithm 2 (Appendix), presents the modification to the classic incremental algorithm for Delaunay triangulations. The function in the internal data structure is part of the proposal for persistent Delaunay triangulation.

Correctness test for the insertion operation is following the steps below.

1. Initialization: Initially there is a temporary Delaunay triangulation, which contains all the triangles adjacent to the triangle that contains x. This state can be seen in Figure 3.5. In this figure, only the purple triangulation is loaded into a temporary Delaunay structure.
2. A temporary Delaunay triangulation is maintained in each iteration. In each iteration, the following edge $E \in A$ is evaluated. If a flip operation is executed, then the pair of edges that could be affected and can give rise to new flip operations is loaded. Then, a temporary triangulation is maintained, according to the classic incremental algorithm, until it is not possible to do more flip operations. Here, it is important to note that, in practice, the number of flip operations in a Delaunay triangulation is constant, including $O(1)$ in random vertices distributions [46]. Therefore, its effect on the performance of the proposed algorithm is minimal.
3. Termination: Since the number of flip operations is constant and finite, and that the algorithm ends when there are no more edges in the array, then it is guaranteed that the algorithm ends.

3.1.3. The Operation for Loading Triangulations. The internal data structure does not store information about faces, order, and orientation, as well as information from neighboring triangles. This in order to minimize RAM. In this sense, a procedure must be carried out that allows loading the triangles but considering the correct orientation and information of the neighborhood. For this, Algorithm 3 is used. This algorithm receives a vertex p as input and returns all its adjacent vertices, at a certain time t, arranged in a counterclockwise direction.

In Algorithm 3 all the vertices are obtained within the search binary tree associated with p, taking into account that, for each node of the tree, the argument t of the function, be within the initial range $t_{initial} \leq t \leq t_{final}$. This function reconstructs the set of triangles, which are necessary, to perform the flip operations only on the vertices and edges that exist at that time t.

3.1.3.1. The correctness test for loading triangulations.

1. Initialization: Initially there is only one vertex, which is the vertex p.
2. Given $p \in T$ then all vertices q adjacent to p are loaded. Once all vertices adjacent to p are loaded, they are ordered counterclockwise to p, the correctness of this procedure is given by the correctness of the sorting algorithm to use.
3. Completion: it is guaranteed that this algorithm ends since different vertices are inserted, therefore, i is the number of neighboring vertices and k the total number of vertices inserted until time t, it is guaranteed that the algorithm ends when $i = k + 1$.

Once the edges adjacent to vertex p are loaded, it is necessary to regenerate the triangle formed by these edges. To achieve this, Algorithm 4 (Appendix) is applied.

3.1.3.2. Correctness test for Algorithm 4 (Appendix). Which performs the loading of the temporary Delaunay triangulation, from our internal structure:

1. Initialization: when starting the algorithm, there is only one side, which will consider as infinite or null (NULL).
2. since $p \in T$, then all vertices q adjacent to a p are loaded, this process is shown on line 1, then the initialization of each face is performed, this process is observed on lines 5 through 8. It is like this that before the insertion of each vertex, the Delaunay triangulation corresponding to the inserted vertices, and their respective adjacent vertices, is kept loaded.
3. Completion: let i be the number of nodes inserted until a given iteration and the n total number of vertices inserted until time t, it is guaranteed that the algorithm ends when $i = n + 1$ in the worst case.

3.1.4. The Elimination Operation for the Walk Method. To perform the removal of a vertex in a Delaunay triangulation and as seen in Section 2.4, it is enough to eliminate the protruding edges of the vertex to be inserted along with the vertex. Then perform the triangulation of the resulting polygon, as shown in Figure 3.9 the polygon that should be triangulated is bounded by the purple nodes, with the added edges being within the polygon.

In the case of the persistent data structure, the edges should be updated in the final time as the time in which the vertex removal is performed. And add the new edges with an initial time equal to the current time t and infinite final time (still present in the current triangulation).

There are numerous methods to perform the elimination of a vertex to Delaunay triangulation, in this sense, it is proposed to use a simple and efficient complexity elimination algorithm $(klogk)$ to eliminate the vertex in a Delaunay triangulation, based on Shelling, to this will be used the following lemma [47].

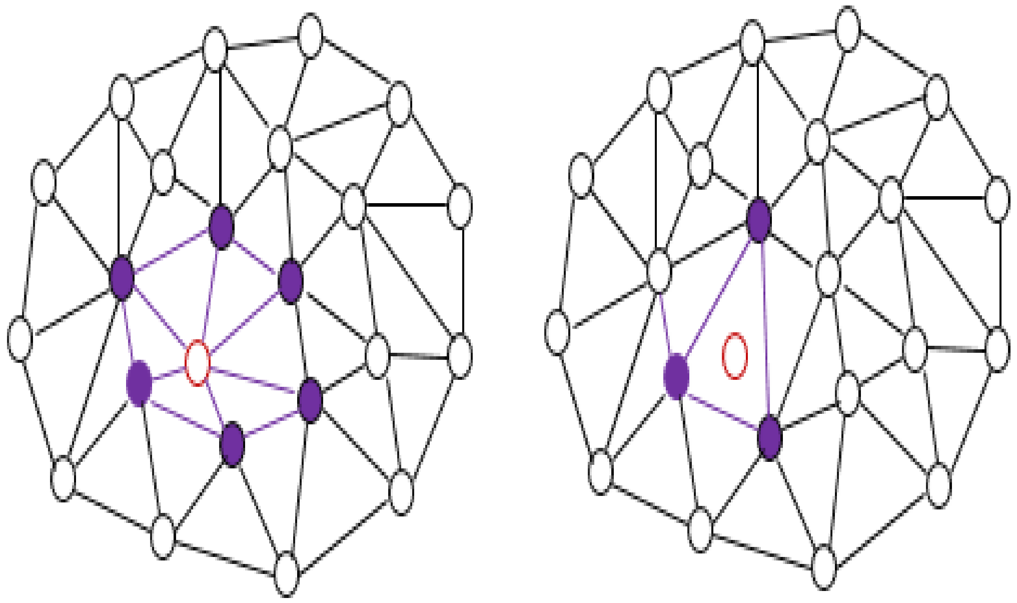

Figure 3.9 The Elimination Operation of the Vertex.

Lemma 3.1: Consider the polygon $H = \{q_0, q_1, \cdots, q_k\}$ and a vertex p such that the edges of $q_i q_{i+1}$ belong to the Delaunay triangulation of $\{q_0, q_1, \cdots, q_k, p\}$. If $\left|(p, circle(q_i, q_{i+1}, q_{i+2}))\right|$ is maximum, then $q_i q_{i+2}$ is an edge of the Delaunay triangulation of $\{q_0, q_1, \cdots, q_{k-1}\}$.

Then the elimination of a vertex can be implemented by maintaining a structure for storing ears and finding the lowest priority. In this implementation, the remaining ears should be part of a doubly-linked list, to better represent the resulting polygon in each iteration of Algorithm 5 (Appendix).

3.2. THE BOWYER-WATSON ALGORITHM

Bowyer [48] and Watson [49] simultaneously presented a random incremental algorithm in their article, which considers Delaunay's rule as the primary criterion, forming triangulations with almost unifying methods.

This algorithm begins with the formation of a sufficiently large artificial triangle that captures all points from a given set P. Artificial points $p_{-1}, p_{-2}\ and\ p_{-3}$ far enough so that they do not affect the structure of Delaunay's triangulation over P, as shown in Figure 3.10. Thus, the Delaunay triangulation was constructed over $P \cup \{p_{-1}, p_{-2}, p_{-3}\}$.

By defining a large artificial triangle, this ensured that all points from P are located inside it. Further, the algorithm randomly selects a point from P and inserts it into a triangulation. For each inserted point, first, find the corresponding triangle or triangle, and then from all existing triangles in triangulation determine that whose circumscribed circle contains the newly inserted point. These triangles are no longer Delaunay's, so delete them from triangulation, and a hole is formed in their place. The algorithm creates connections between the new point and the points that lie on the boundary connections of the hole. In this way, we form new triangles, which are Delaunay's, and fill them with a hole in the triangulation, as shown in Figure 3.9.

In the initial case, the first inserted point will fall inside the artificial triangle and divide it into three smaller triangles, and then the division of the triangles and replacement of the hole by the remaining points from P will be carried out iteratively until the entire set P is processed. In the last step of the triangulation, the algorithm deletes all triangles, of which at least one corner is formed by the point $p_{-1}, p_{-2}, or\ p_{-3}$.

In this step, care must be taken not to destroy the edge of the convex triangulation envelope. The result is a Delaunay triangulation over P.

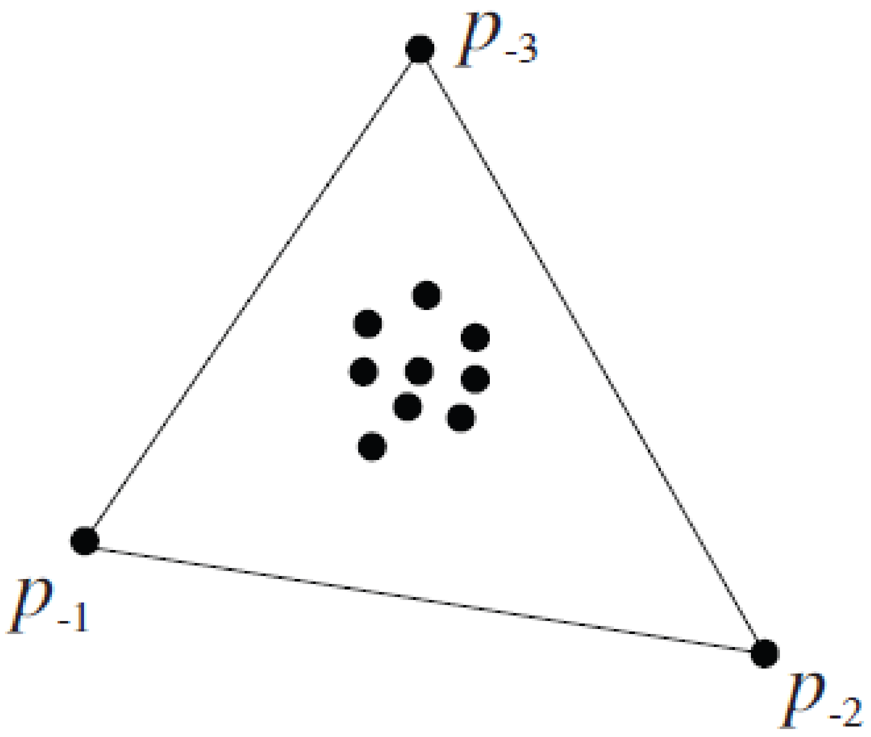

Figure 3.10 Large Triangle (Contains All the Points Inside It).

The algorithm is scalable to several dimensions, and it takes $O\left(n^{1+1/k}\right)$ time to build Delaunay triangulation, where k is the number of dimensions and $O(n)$ time to manage the data structure [48]. For the two-dimensional Delaunay triangulation, the points were to be sorted before being inserted into the algorithm, the algorithm to build the Delaunay triangulation would take $O(n \log n)$ time. In practice, this is usually negligible.

To start measuring the run time of the algorithm, we have chosen a step where makes a random permutation over the read points. Then store the current time of the algorithm.

The method stores the time value in nanoseconds. To finish measuring the run time of the algorithm, determined the step when the algorithm verifies that all points connections on the edge of the convex triangulation hull are present in the DAG and, if not, form the corresponding triangles. In this step, the algorithm returns the Delaunay triangulation and thus completes its purpose, so stopped the measurement after its execution. To conclude the measurement, again use the System-Time method to save the current value, which is greater than the previously stored value exactly for the time taken by the Delaunay triangulation algorithm. Then subtract values from each other to give the algorithm run time in nanoseconds, which is converted to milliseconds and seconds for better transparency.

3.3. THE HYBRID METHOD

A modification to the Walk method is proposed, with this modification the process of finding the triangle tr containing p is accelerated, where p is the vertex to be inserted in the current triangulation T. For this, a structure for the location of vertices D is constructed, which is an acyclic directed graph [50]. The final nodes of D correspond to the triangles of the current triangulation T, the internal nodes of D correspond to the triangles that belonged to the triangulation at some previous time and that are no longer part of the current triangulation.

The structure to find the vertex was taken from De Berg [24] and is constructed as follows. Initially, a triangle is created large enough to contain inside all the vertices to be inserted, this can be seen in Figure 3.11. In this figure, (a) represents a single triangle which we will call Δ_1. In (b) the data structure is shown; which, in this case, only consists of a single node.

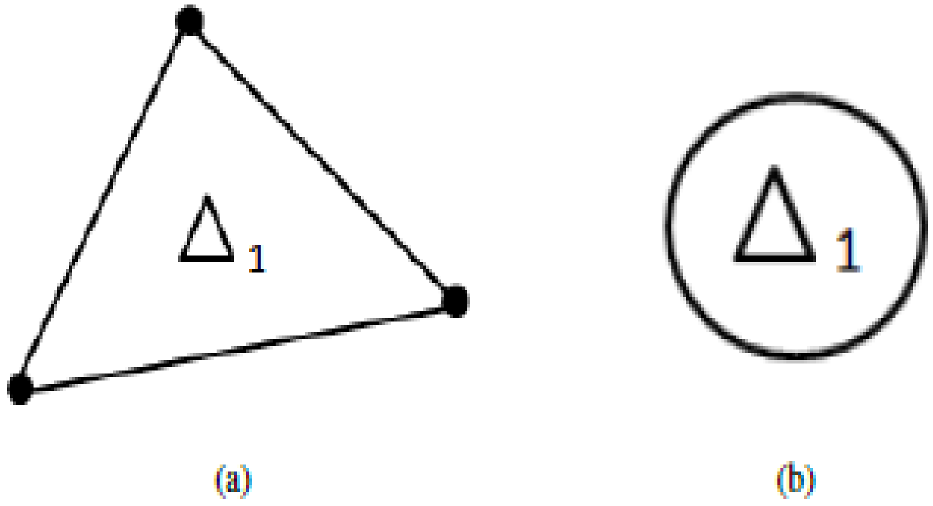

Figure 3.11 The Initial Structure.

Otherwise, in Figure 3.12, in (a), the insertion of a vertex inside Δ_1 is observed. This generates three new triangles, which we will call Δ_2, Δ_3 *and* Δ_4. In (b), the data structure (acyclic directed graph) of the triangulation in (a) is appreciated. As can be seen, the root node remains Δ_1, that is, the triangulation in the previous time. To this node, nodes Δ_2, Δ_3 *and* Δ_4 are added that correspond to the new triangles generated by inserting the new vertex; In this way we maintain both the current version of the triangulation and the previous ones.

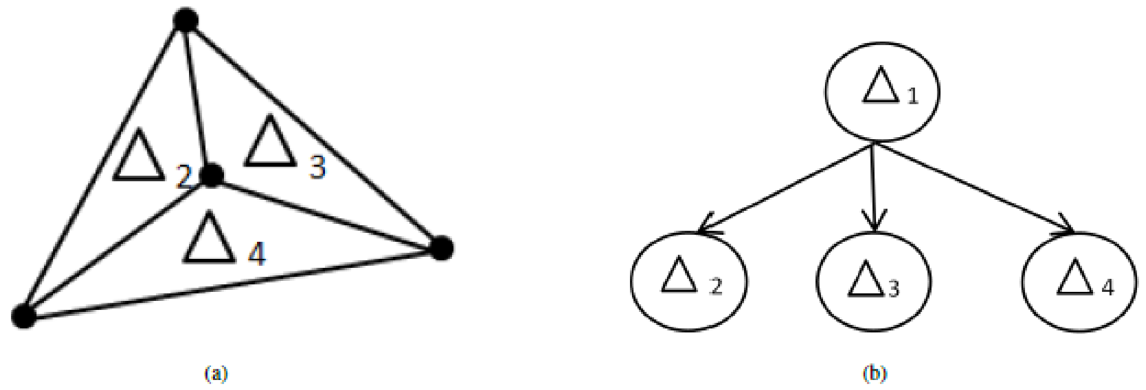

Figure 3.12 The Initial Structure After Inserting a Vertex.

In addition, to perform a flip operation, in this data structure, you must find the two triangles that share the edge in which the operation will be carried out. For example, this behavior is shown in Figure 3.13. In (a1) it can be seen that Δ_7 and Δ_8 have an illegal edge and therefore a flip operation must be performed so that the triangulation changes, eliminating Δ_7 and Δ_8 and generating triangles Δ_{11} and Δ_{12}, as shown in (a2).

On the other hand, this variation in triangulation must generate a change in the data structure that represents it. This can be noticed in (b1) and (b2). Where (b1) and (b2) are the directed acyclic graphs of (a1) and (a2) respectively. In (b2) triangles Δ_{11} and Δ_{12} have triangles Δ_7 and Δ_8 as parents, therefore Δ_7 and Δ_8 were replaced by Δ_{11} and Δ_{12}. In this way all leaf nodes represent the current triangulation, while the other nodes in the graph form triangles in times past. Then you can summarize the data structure in Algorithms 6 and 7 (Appendix).

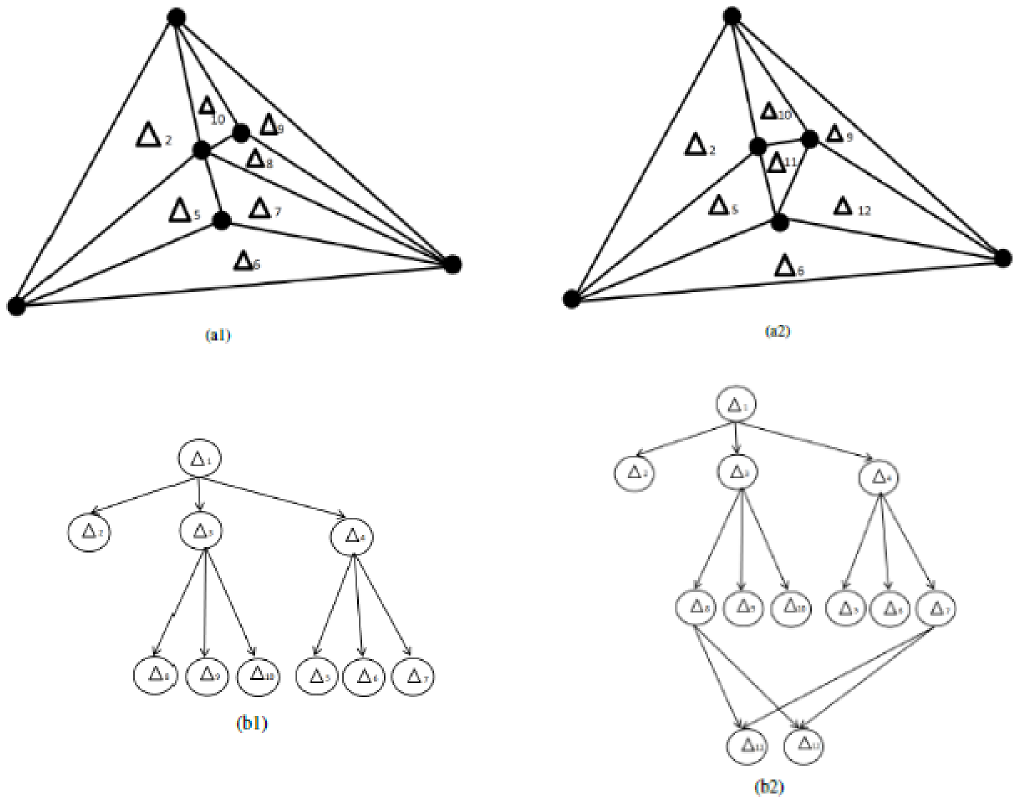

Figure 3.13 The Initial Structure After Insert-Flip.

3.4. THE GRAPH METHOD

The data structure seen above maintains persistence, as it contains all the triangles that were once part of the triangulation. In addition to the triangles of the current triangulation as final nodes of the graph. In addition, if a time identifier was added in each node of the structure and would store in an array sequentially sorted in a non-descending manner by the associated time. Then a triangulation could be retrieved in a given time t making use of binary search on this arrangement and thus be able to retrieve the final nodes of the structure in that certain time.

3.5. THE FULL PERSISTENCE

This section presents an additional contribution. In addition to partial persistence, the data structure supports full persistence, as in Figure 3.14 [33].

To achieve this type of persistence, an integer value is added to each node of the balanced binary search tree. This value indicates the version to which an edge belongs, and, in this way, all operations remain equal to partial persistence.

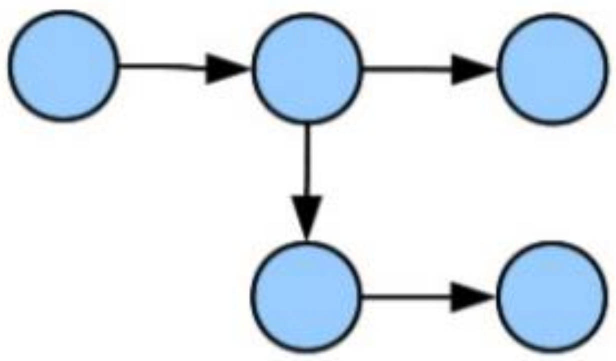

Figure 3.14 The Full Persistence.

All the algorithms applied in the partial persistence method work similarly in the full persistence, with the difference that it is now necessary to consider the value of the version. One of the difficulties, in the method of complete persistence, is to find the vertex in the version tree from where to start the path to find the triangle that contains the vertex to be inserted. This procedure is complicated in contrast to its partial version.

Considering the version tree of Figure 3.14, the problem is reduced to finding an ancestral random node of the version to be modified. So, it is proposed to use dynamic programming to find the ith ancestor of the node, that is to say the $i - th$ node on its way to the root, with this it is guaranteed to find said vertex with a computational cost of $O\ (logn)$ of insertion by node, and a computational cost of consultation $O\ (logn)$ [51].

The focus is on pre-processing, using dynamic programming, sub-matrices of length 2^k. Maintaining a matrix $M\ [N][logN]$ where $M\ [i]\ [j]$ is the 2^j ancestor of i. In Algorithm 8 (Appendix), all 2^i ancestors of p are calculated.

3.5.1. The Correctness Test for the Full Persistence.

1. Initialization: At the beginning of the algorithm, the ancestor of vertex p is the father of p.
2. At the end of each iteration, the 2^i ancestor of p is calculated, from the following recurrence function, where $T(x)$ represents the father of x:

$$Ancestor\ (i;\ x) = \begin{cases} T(x), & if\ i = 0 \\ Ancestor\ \big(i-1;\ Ancestor\ (i-1;\ x)\big) & otherwise. \end{cases} \tag{12}$$

3. Completion: the completion of the algorithm is guaranteed, since the height of the version tree has $a\ value\ \leq\ n$ where n is the number of versions. Therefore, let m be the number of ancestors of p, it is guaranteed that the algorithm ends in the worst case when $i > log2\ (n\ +\ 1)$.

In Algorithm 9 (Appendix), a random node ancestor of p is calculated. We check if in the binary representation in nanometer the bit that corresponds to the position i is on, if it is, id Random Vertices is updated as its $i-th$ ancestor.

3.5.2. The Correctness Test of Algorithm 9 (Appendix).

1. Initialization: Initially the 0 ancestor of $p\ is\ p$.
2. Given the binary representation of ancestor, random vertex is updated to be the next 2^i ancestor of p.

3. Completion: Considering that the height of the tree has $a\ value \leq\ n$, where n is the number of versions, and let m be the number of ancestors of p. Then, it is guaranteed that the algorithm ends in the worst case when $i >\ log2\ (n\ +\ 1)$.

3.6. POINTS RATE IN DELAUNAY TRIANGULATIONS

Many algorithms for constructing Delaunay triangulation depend in one way or another on the number of connections to which each point belongs, that is, on the degrees of the point in the triangulation. There is little mention in the literature on the notion of the highest point rate in Delaunay triangulation. Bern, and Yao [52] proved that the expected maximum point rate in a Delaunay triangulation planar, with points plotted by a homogeneous Poisson process at the interval $\left[0, \sqrt{n}\right]^2$, is known as $\frac{\log n}{\log \log n}$.

$$n - e + m = 1 \tag{13}$$

In practice, it turns out that the boundary is not sufficiently precise, since it is only an approximation limited to the distribution of points by the Poisson process, nor does it imply that in practice there are points that only they radiate at the edge of the convex hull triangulation, many times higher than expected. Broutin, Devillers, and Hemsley [53] have proved that for a random distribution, this ck in the planar in the Delaunay triangulation is more accurate at the expected maximum radiance of this $\log^{2+\varepsilon} n$, for each $\varepsilon > 0$. when that number of points n goes to infinity.

From Euler's formula (4) [12], the Delaunay triangulation DT over a set of power points has n less than $3n$ connections and $2n$ triangles. Since each connection contains two points, the sum of point rates for all nodes in DT is less than 6n. If we divide this sum by the number of all points in DT, we get an average point rate in DT less than 6.

3.7. DETERMINING THE MAXIMUM AND AVERAGE POINT RATE.

The maximum and the average points rate was determined after the algorithm has already constructed a Delaunay triangulation over a given set of points. A table that will keep the vertices rate for each vertex in the given set was defined. Then, walk across all the triangles that construct the Delaunay triangulation and store its degree for each vertex in the table. This has been done by checking the number of occurrences of vertices in triangles in the DAG. Each time a vertex sits in one of Delaunay triangles, the time counter in the corresponding field in the table will be increased until all vertices and triangles are processed.

Specific examples are vertices that lie on the edge of the convex hull, which has a degree higher than the number of the triangles. Thus, a particular criterion is added for these vertices to check that an individual vertex is part of the edge of the convex hull of the Delaunay triangulation. If so, the rate of the corresponding vertices is increased by one more, and the highest vertices rate is obtained by finding the highest vertices in the vertices rate table. An iterative values comparison was held and always store the maximum until all the vertices are processed. The average vertices rate is obtained by summing all the vertices degrees with each other and dividing that value by the number of vertices.

4. TESTS AND RESULTS

This section explains the tests and results performed during the implementation. The tests were divided into three groups.

The first part of the test is focusing on constructing a planar Delaunay triangulation that is modeled after the Bowyer-Watson algorithm. The data structure that was used for this test is DAG because it allows us to easily navigate between the contained nodes. It is one of the classic implementations with the HashMap class to store the triangles for which created the new class.

The second group of the test is focusing on analyzing the temporal and spatial complexity of the three proposed methods: Walk, Hybrid and Graph, in contrast to the CGAL library. It is important to emphasize here that the three methods are persistent, while CGAL does not. Therefore, in this experiment, it is intended to demonstrate that the computational complexity of the methods with persistence is like the CGAL itself, who does not store temporary information.

Where in the third group of experiments, the temporal and spatial behavior of persistent structures was analyzed, with a set of random data. Finally, randomized data tests were performed to analyze the behavior of the fully persistent data structure. In the last case, tests were carried out with the Walk method because it is the only method that has full persistence.

4.1. THE BOWYER-WATSON ALGORITHM RESULT

For this development of the experimental group the data set that used is the DAG dataset. In this experiment, the procedure is to walk across all triangles in the DAG dataset and remove triangles whose at least one endpoint is equal to the endpoint of the initial triangle. This eliminated the artificial points and consequently. The edge of the convex hull triangulation could be destroyed in this step, so it is necessary to verify that all Delaunay triangulation connections lying on the edge of the convex envelope are indeed located in the DAG.

4.1.1. The Measurement for the Run Time. The running time of the Bowyer-Waston algorithm was examined. Table 4.1 presents the results for the test samples. Figure 4.1 shows the running time of the algorithm as a function of the vertices number.

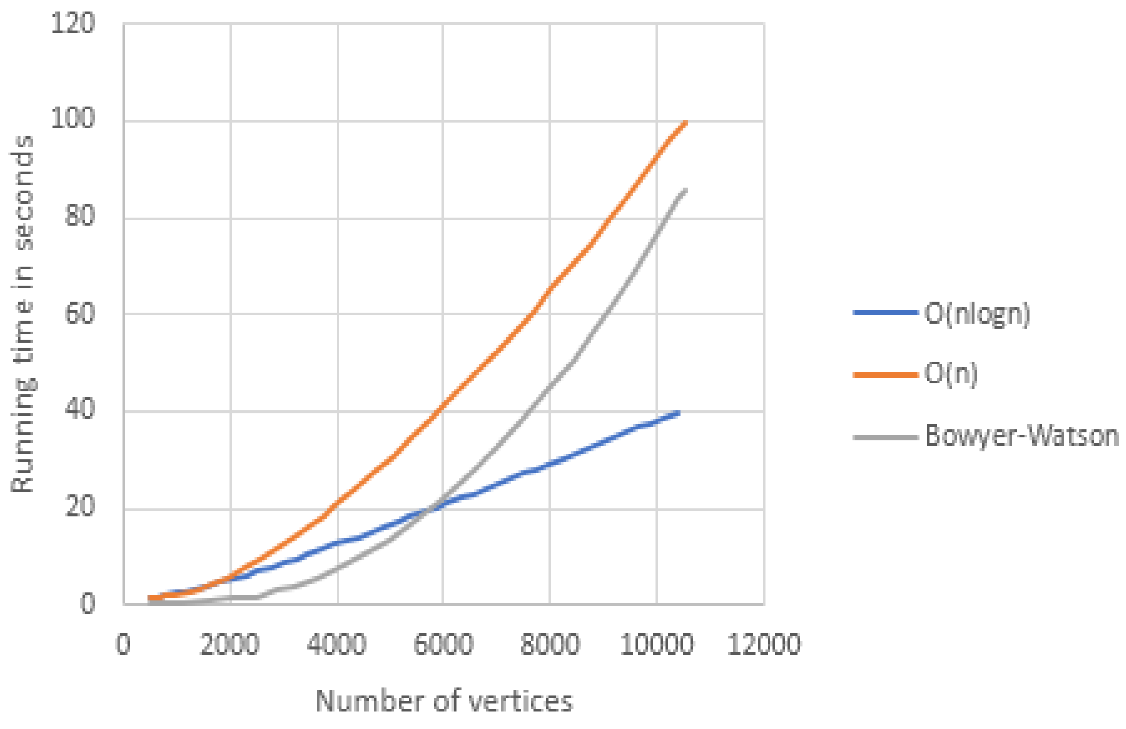

Figure 4.1 The Run Time for the Bowyer-Waston Algorithms.

Table 4.1 The Average Run Time for the Given Number of Vertices.

Number of vertices	Average run-time (in seconds)
2000	1.054
4000	3.851
6000	9.008
8000	16.552
10000	25.593
12000	37.522
14000	50.801
16000	66.588
18000	84.716
20000	100.503
22000	105.864
24000	110.413
26000	115.587
28000	120.841

It is observed that the average execution time of the algorithm increases with the number of vertices but remains within the limit $O(n)$.

4.1.2. Experiments of the Maximum Point Rate for a Given Vertices. The highest points rate in the Delaunay triangulation over the given sample is calculated. Table 4.2 presents the results for the average value of the highest point rate. Table 4.3 lists the maximum and minimum point rate of all samples for the given number of vertices. The results show that when increasing the number of vertices, a regular growth in the value of the highest point rate occurs. It observed that samples up to and including 12000 vertices far exceed the theoretically expected maximum level. Also, samples from 16000 vertices fall below the expected limit on average. It was observed that among 6000 samples for each number of vertices, there was such a model where the maximum rate exceeded the expected limit.

The average point rate in the Delaunay triangulation over the given sample was tested. The results are presented in Table 4.4. As the number of vertices increases, it is clear there is an increase in the average point rate. As the number of vertices increases, it is clear there is an increase in the average vertex rate that approaches asymptotically the limit 6.

4.2. THE EXPERIMENTAL GROUPS TEST IN PARTIAL PERSISTENCE DATA STRUCTURES

For the development of the experiments, two databases were considered, one consisting of a set of images and the other by random vertices. Each of them is described below.

Table 4.2 The Maximum Point Rate for Given Vertices.

Number of vertices	The highest rate for vertices	The expected value of the highest rate for vertices
2000	10.15	8.56
4000	11.355	10.45
6000	11.925	11.56
8000	12.555	12.431
10000	13.085	13.311
12000	13.335	14.021
14000	13.515	14.611
16000	13.805	15.121
18000	14.125	15.571
20000	14.655	15.981
22000	14.775	16.391
24000	14.895	16.751
26000	15.185	17.031
28000	15.475	17.465

Table 4.3 The Maximum and Minimum Value of the Highest Number.

Number of vertices	The maximum value of the highest vertices rate	The minimum value of the highest vertices rate
2000	12	8
4000	12	8
6000	14	8
8000	14	9
10000	15	9
12000	16	10
14000	16	11
16000	18	12
18000	18	12
20000	18	12
22000	19	13
24000	20	13
26000	21	14
28000	22	14

Table 4.4 The Average Value for a Given Number of Vertices.

Number of vertices	The average value of the vertices rate
2000	4.514
4000	5.171
6000	5.289
8000	5.289
10000	5.413
12000	5.433
14000	5.441
16000	5.445
18000	5.446
20000	5.448
22000	5.45
24000	5.45
26000	5.47
28000	5.52

The first experiment was developed using a database of 15 images of different sizes. Harris's algorithm [54], it was applied to each of the images to obtain the most representative set of pixels in the image, these pixels are called key points. Subsequently, the key points were stored in files and from them, the Delaunay triangulation was generated with the 3 proposed methods (Walk, Hybrid and Graph).

In the second experiment, several tests were carried out with different sets of random vertices. 15000, 25000, 35000, 45.000, 55.000, 65.000,75.000 and 95.000 vertices were taken. In both experiments, comparisons were made regarding temporal cost and space cost. The results obtained for each of the test groups are described below.

4.2.1. The Comparing of the Image Experiments Results. Figure 4.2 shows Images in three columns, the first shows the real images, the second column shows the result of applying the Harris algorithm [54] on the images of the first column to obtain the most representative set of pixels of the images (key points). Finally, in the third column the Delaunay triangulation is shown carried out on the vertices of the second column.

4.2.1.1. The cost results for the CPU. Table 4.5 shows the results obtained (in seconds) when applying the proposed methods (Walk, Hybrid and Graph) in comparison with the results of CGAL. In this table, the first column represents the number of vertices (key points) obtained in each image. Finally, columns 2, 3, 4, and 5 are the results, in seconds, of the computational cost of the Hybrid, Graph, Walk, and CGAL algorithms, respectively. In the results, it can be observed that, as the number of vertices is greater, the Graph method has a better performance than CGAL.

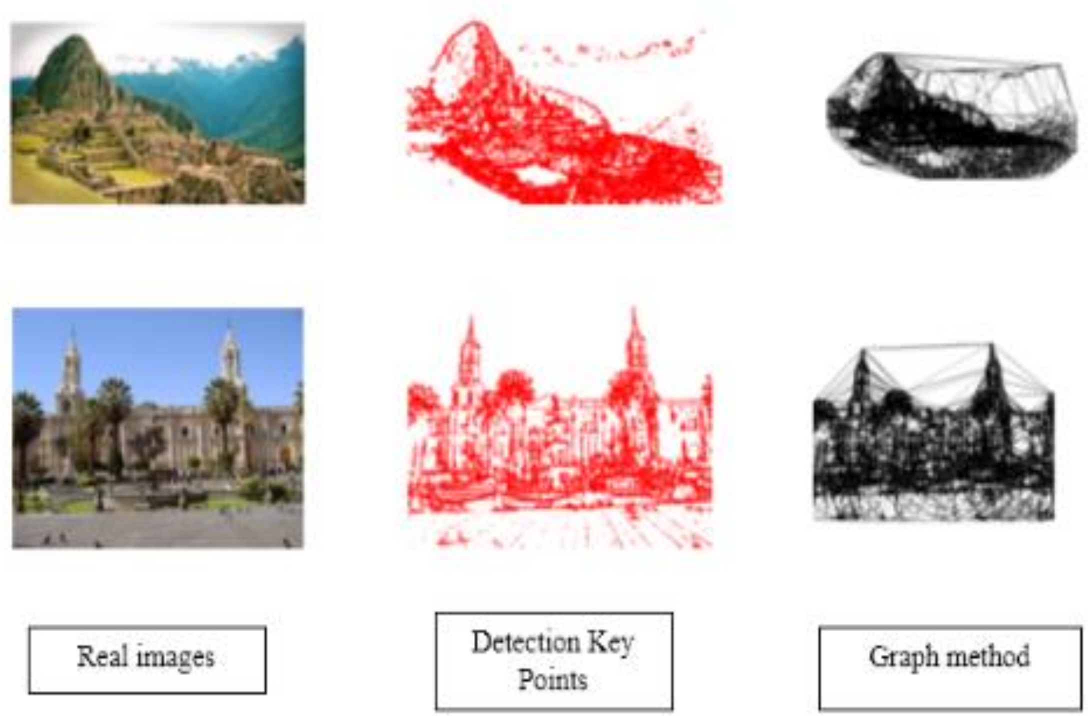

Figure 4.2 Results of Applying Triangulation Algorithms.

On the other hand, the Walk and Hybrid method have a lower performance, because the first one uses the modified search algorithm which increases the computational cost by a factor of $O(\sqrt{n})$. While in the second, it combines the internal data structure with the data structure of the Graph method, so that it is necessary to load the triangles of the internal structure each time an insert is made.

4.2.1.2. The cost results for the RAM. In the same way as in the experiment to evaluate the temporary cost or cost in CPU, Table 4.5 shows the results of RAM consumption of the three proposed methods Walk, Hybrid and Graph compared with CGAL.

While Table 4.6 shows that, of the three methods proposed to build a persistent data structure, the lowest RAM consumption was obtained by the Graph method. It is important to note, at this time, that the CGAL library requires less RAM. However, CGAL is not persistent, which implies that it does not maintain information from past versions and in that sense, it is expected that its spatial complexity or consumption of RAM memory is much lower than the 3 proposed methods.

4.2.2. The Experimental Results on the Random Dataset. This section will contain the cost result for the CPU and RAM.

4.2.2.1. The cost results for the CPU with the random dataset. In the same way as previous experiments, in Table 4.7 the temporal behavior of the proposed algorithms (Walk, Hybrid and Graph) can be seen in comparison with CGAL for the construction of the Delaunay triangulation for a certain number of vertices. Although CGAL does not admit persistence in any of its types, the objective was to analyze whether any of the proposed algorithms, in addition to being persistent, it could execute Delaunay triangulation competitively with CGAL.

Figure 4.3 shows that the Walk method, for the random dataset, has a computational cost greater than the Hybrid and Graph methods and obviously CGAL. In addition, we can see that the Graph method has a computational cost lower than the two proposed methods (Walk and Hybrid) and is even competitive with CGAL. In summary, the proposed Graph method proved to be the most efficient of the proposed methods.

Table 4.5 Analysis of the Temporal Cost (in seconds) of the Walk, Hybrid, Graph and CGAL Methods.

No. of vertices	Hybrid method	Graph method	Walk method	CGAL
1239	0.432	0.254	0.541	0.066
4362	1.123	0.623	1.289	0.242
4624	1.276	0.665	1.560	0.278
4954	1.3.14	0.679	1.658	0.308
5231	1.357	0.682	2.170	0.364
7436	2.851	1.103	2.946	0.648
9753	3.182	1.558	3.688	0.874
13245	6.001	2.099	5.480	1.296
21669	6.487	3.776	11.605	2.603
21788	10.406	3.799	11.714	2.621
30689	13.754	6.091	16.768	5.147
33579	36.845	6.831	21.386	5.417
125889	37.583	25.853	103.931	34.907
128588	43.167	27.204	104.992	38.415
131478	49.475	27.259	106.477	39.075

Table 4.6 Analysis of the Spatial Cost (in Kbytes) of the Walk, Hybrid Graph and CGAL Methods.

No. of vertices	Hybrid method	Graph method	Walk method	CGAL
1239	48.376	1798	46.580	448
4362	53.164	6110	49.696	740
4624	54.112	6914	50.588	736
4954	54.152	7298	50.597	740
5231	54.468	7358	51.689	740
7436	59.608	12.078	55.221	1032
9753	64.904	16.990	58.717	1440
13245	71.468	22.786	62.849	1728
21669	90.300	40.050	75.777	2808
21788	90.312	40.354	75.879	2808
30689	109.008	56.714	87.515	3832
33579	113.948	62.974	90.955	4008
125889	284.204	213.186	204.863	13.672
128588	284.542	213.576	205.654	13.672
131478	301.910	228.960	217.446	14.628

Table 4.7 Analysis of the Cost (in Kbytes) for the Walk, Hybrid Graph and CGAL Methods.

Random insertion	Hybrid method	Graph method	Walk method	CGAL
5000	2.154	1.112	3.254	1.019
15000	3.774	2.096	6.929	1.886
25000	5.447	3.105	8.757	2.831
35000	7.333	4.103	12.179	3.502
45000	9.388	5.165	16.027	4.575
55000	10.896	6.204	19.454	5.655
65000	13.439	7.687	23.054	6.900
75000	13.954	8.390	28.559	9.094
85000	17.616	11.081	33.656	9.494
95000	18.844	11.412	36.424	11.732

This behavior is due to the fact that in the Graph method, the algorithm to find the triangle which contains the vertex that is intended to be inserted, has a $logn$ computational cost, in relation to the Walk method whose search algorithm has a cost of $\sqrt{n}$. On the other hand, the hybrid method adds, to the Walk data structure, an additional structure to allow the triangle search to be done in $log\ (n)$ time, however, by requiring loading all the triangles affected by the process of insertion, its computational cost is increased and, for this reason, its performance is lower than the Graph method.

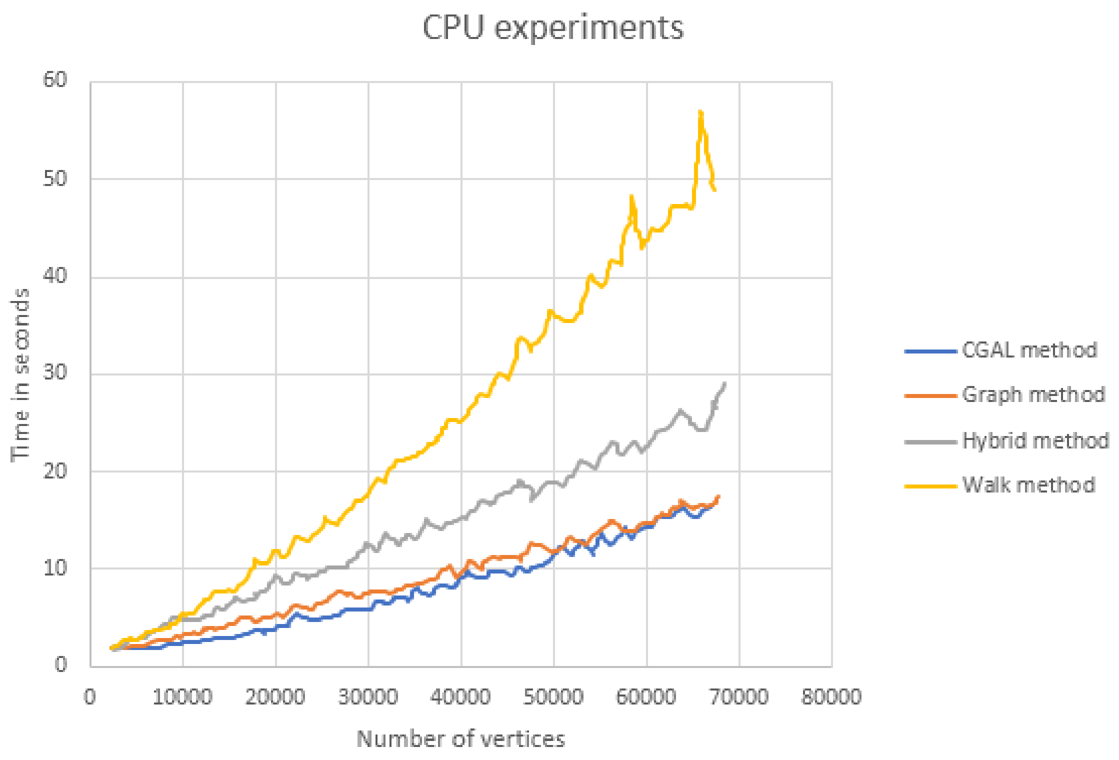

Figure 4.3 Comparison of CPU Performance Between Proposed Methods and CGAL (Line Diagram) at Random Points.

In conclusion, we can say that the Graph method is the method with the best results in terms of computational cost and, it meets the objective of developing a persistent data structure that minimizes computational time when performing operations on the structure. In this specific case, of the operations from insertion.

4.2.2.2. The cost results for the RAM with the random dataset. Table 4.8 shows the results in the consumption of RAM (Kbytes) of the proposed algorithms (Walk, Hybrid and Graph) compared to CGAL. Although CGAL does not admit persistence in any of its types, the objective was to analyze whether any of the proposed algorithms, in addition to being persistent, could execute Delaunay triangulation competitively with CGAL.

Figure 4.4 diagram line data shown in Table 4.8, in this figure the Graph method requires less memory than the Hybrid Walk and methods. It can be concluded that, without including the implementation in CGAL, the method that consumes less RAM is the Graph method, this is because in random distributions the depth of the graph is $O\ (logn)$.

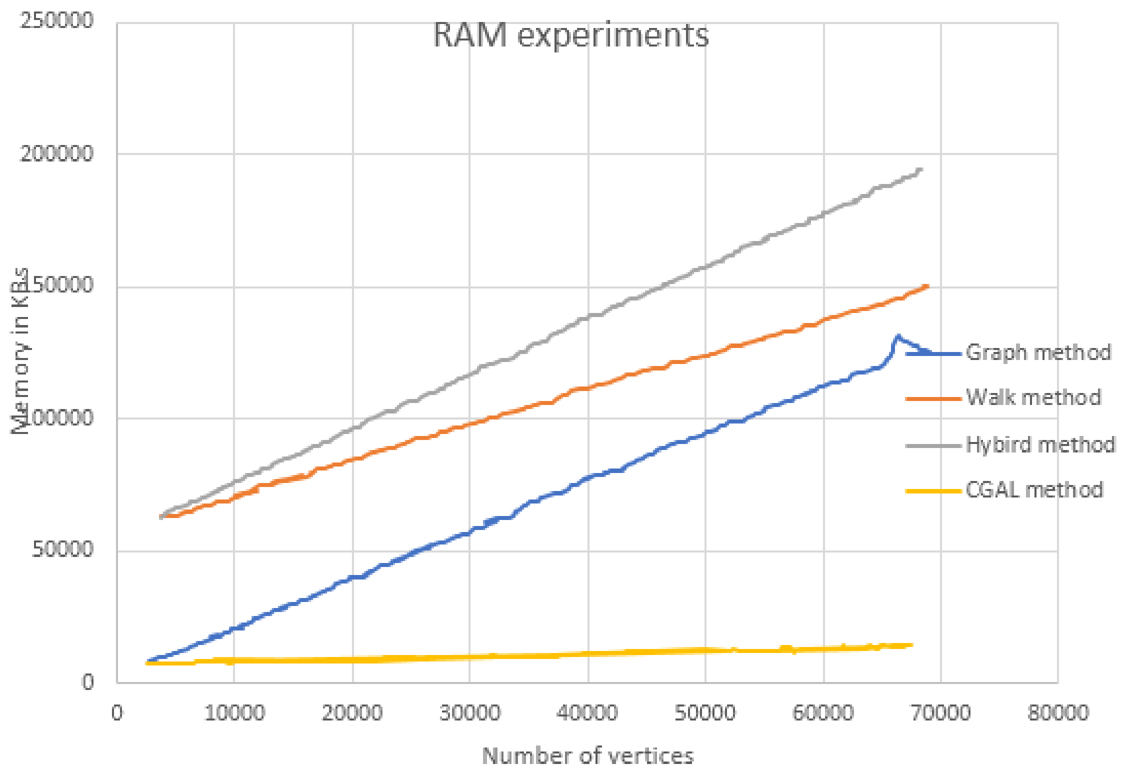

Figure 4.4 Comparison of CPU Performance Between Proposed Methods and CGAL (Line Diagram) at Random Points.

Table 4.8 Analysis of the Spatial Cost (in Kbytes) of the Walk, Hybrid, Graph and CGAL Methods at Random Points.

Random Insertion	Hybrid method	Graph method	Walk method	CGAL
5000	69.548	18.836	59.748	1440
15000	80.240	29.232	66.744	2048
25000	91.716	38.832	74.440	2660
35000	102.408	48.204	81.296	3228
45000	113.232	59.624	88.300	3832
55000	125.512	69.392	96.572	4524
65000	136.204	78.926	103.568	5060
75000	147.028	88.440	110.564	5584
85000	157.720	97.812	117.548	6188
95000	168.412	107.184	124.556	6792

4.2.3. Query Experiments for the Incremental Random Datasets. In the same way as previous experiments, in Table 4.9 and in Figure 4.5 you can see the temporal cost of the proposed algorithms (Walk, Hybrid and Graph) on a triangulation of 95000 vertices, where queries are made to be able to recover a certain triangulation in a previous time t, from a total of t vertices, for the algorithms (Walk, Hybrid and Graph).

Where Table 4.10 and Figure 4.6 show the computational cost of fully loading a triangulation at an earlier time t for the Walk, Hybrid and Graph methods.

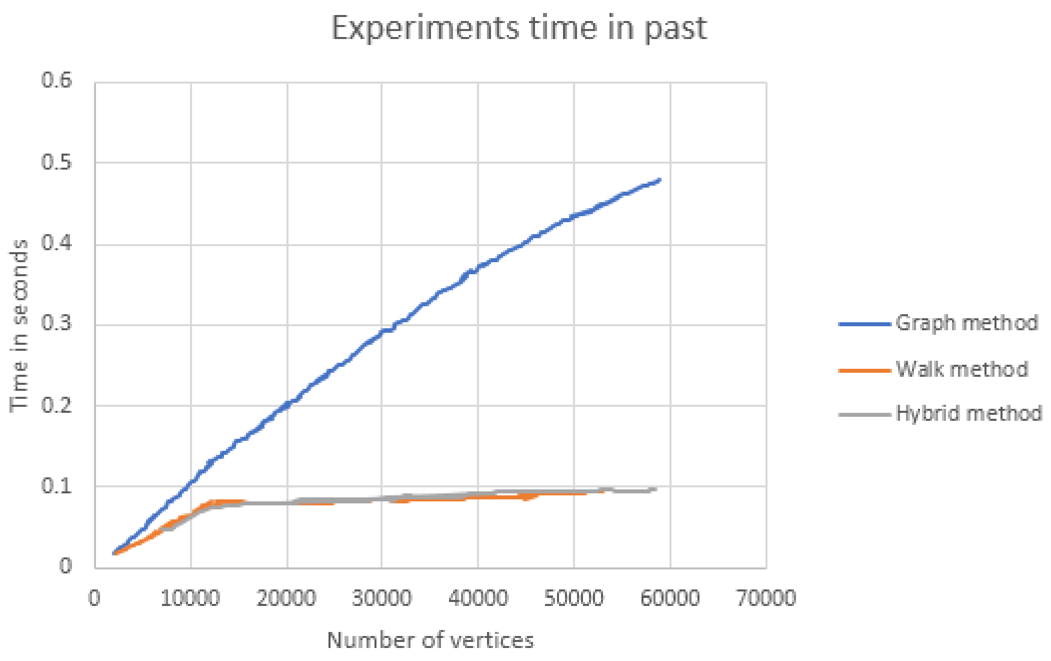

Figure 4.5 Consultations in Previous Times for the Delaunay Triangulation.

Table 4.9 Analysis of the Temporal Cost (in seconds) When Loading a Triangulation in the Walk, Hybrid and Graph Methods at Random Points.

No. of Vertices	Hybrid method	Graph method	Walk method
5000	0.034	0.068	0.038
15000	0.035	0.107	0.040
25000	0.037	0.145	0.042
35000	0.039	0.182	0.044
45000	0.043	0.217	0.047
55000	0.045	0.253	0.049
65000	0.051	0.282	0.051
75000	0.053	0.311	0.054
85000	0.055	0.333	0.057
95000	0.059	0.354	0.060

Table 4.10 Analysis of the Temporary Cost (in seconds) When Consulting the Walk, Hybrid and Graph Methods at Random Points.

No. of Vertices	Hybrid method	Graph method	Walk method
5000	0.236	0.187	0.247
15000	0.330	0.348	0.334
25000	0.429	0.499	0.437
35000	0.531	0.654	0.562
45000	0.635	0.792	0.644
55000	0.745	0.936	0.746
65000	0.856	1.075	0.858
75000	0.967	1.192	0.963
85000	1.072	1.338	1.074
95000	1.195	1.477	1.187

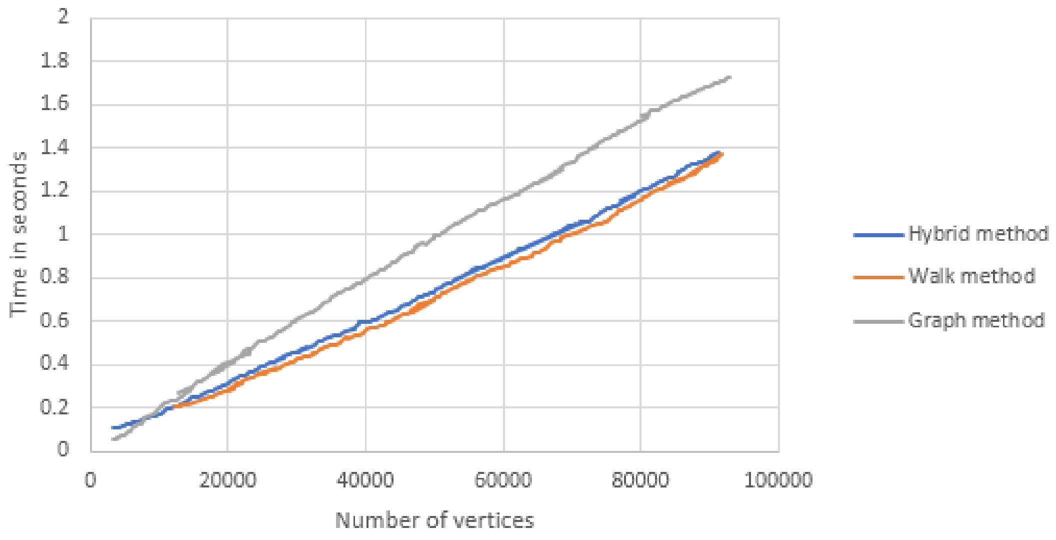

Figure 4.6 Delaunay Triangulation Load in Previous Times.

4.3. THE EXPERIMENTAL GROUPS TEST IN FULL PERSISTENCE DATA STRUCTURES

Unlike the previous partial persistence, in this experiment, Delaunay triangulation was constructed considering that each insertion operation is carried out in a version of the structure so that each modification will produce a new version.

In this section, experiments were carried out to determine the computational cost of the search method for the triangle containing the inserted vertex. Comparisons were made between the proposed algorithm and a brute force technique. On the other hand, comparisons were made between the fully persistent structure and CGAL modified to simulate total persistence.

4.3.1. Search the Algorithm Evaluation. Table 4.11 shows the times to find the triangle that contains the vertex to be inserted, as well as the loading operations of the triangles that contain this vertex and the flips necessary to construct the Delaunay Triangulation. Where Figure 4.7 shows the differences in performance when using the two types of algorithms to search for the triangle that contains the vertex to be inserted (brute force and random vertex).

To analyze the performance of the queries in the proposed data structure, a fully persistent Delaunay triangulation with 15,000 vertices was created. It should be noted that the queries allow finding the number of neighbors of a vertex in a certain time t, and of a certain version. As in the previous experiment, the queries maintain a similar performance and independent the size of the triangulation for the time of the query and the version in which it is located. In Figure 4.8, it can be seen that since the nodes are directly indexed, the query time maintains an almost constant value per search time.

Table 4.11 Performance Results in Insertion Operation (in seconds) in Delaunay Triangulation with Complete Persistence.

No. of Vertices	Walk method	Brute Force Method
500	0.002	0.007
1000	0.058	0.054
3500	0.836	0.927
10000	6.729	8.043
15000	18.109	21.043

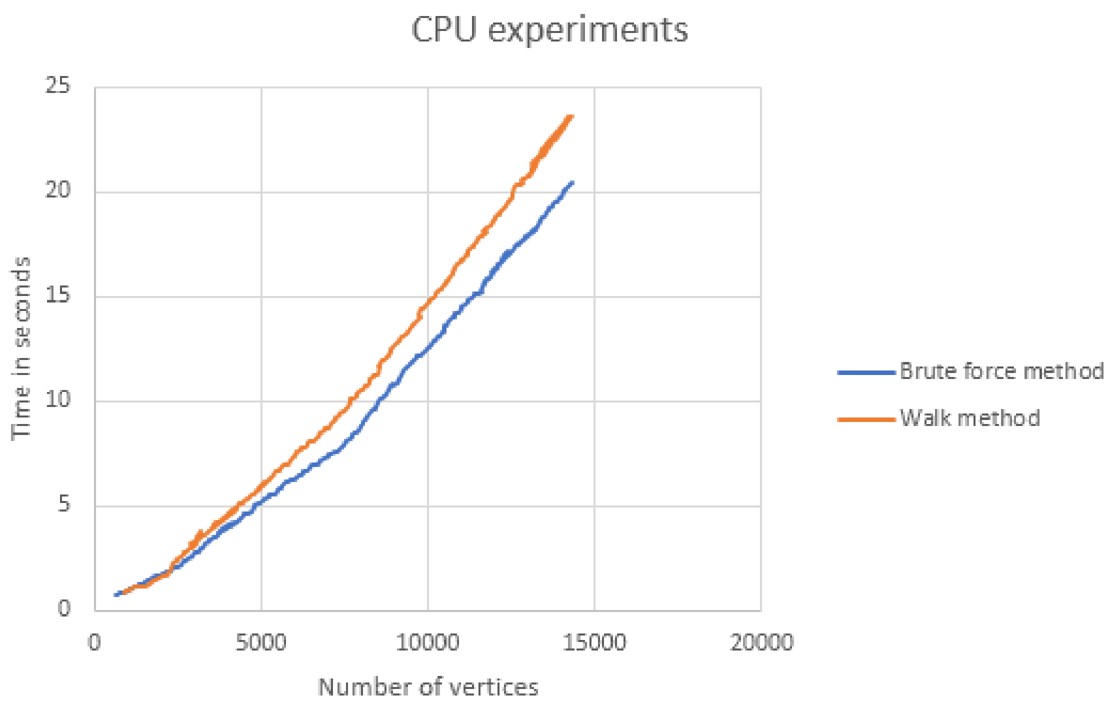

Figure 4.7 Comparison of Execution Times (in seconds) of a Brute Force-Based Method Against the Walk Method for Full Persistence.

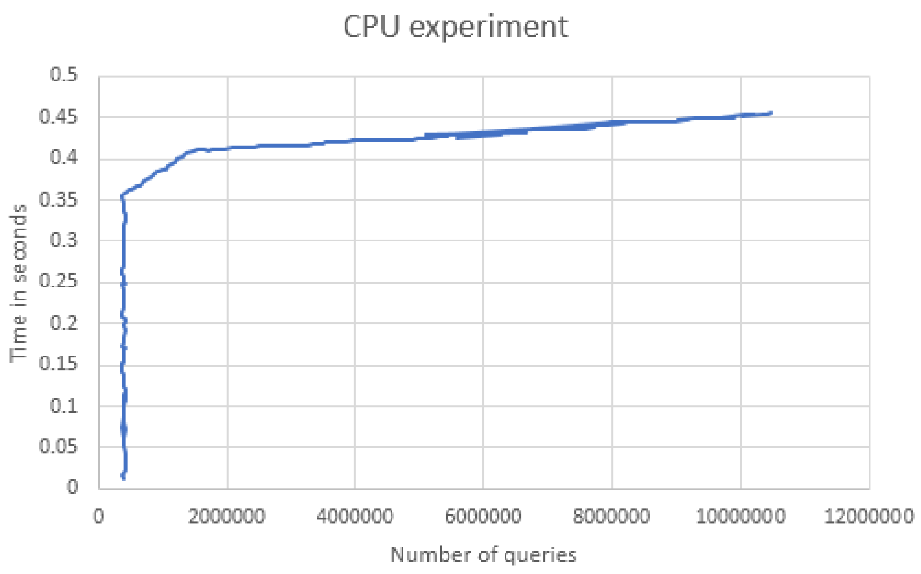

Figure 4.8 Execution Time (in seconds) of Query Operations for Full Persistence.

4.3.2. Evaluation of the Proposal Method and the CGAL Method. The last experiment was developed in order to compare the performance, in terms of RAM memory and efficiency, between the proposal and the CGAL (Computational Geometry Algorithms Library), which is a library of geometric algorithms widely used in various C ++ applications.

The experiment was developed considering that CGAL does not have persistence, in this sense, and to be fair, vertices were inserted incrementally both in CGAL and in the proposal. On the other hand, after each insertion, a copy was made of the entire structure in CGAL to allow persistence and contrast results with our proposal. In this way, demonstrate that, indeed, our persistent structure drastically reduces the amount of RAM memory, and is also more efficient than maintaining persistence in CGAL with the naive method of keeping copies.

Table 4.12 shows the amount of RAM memory, in Kbytes, required by CGAL and by the proposal. Also, the results are observed, in seconds, of the efficiency of our proposal in contrast to CGAL. In the results, it can be seen that as the number of vertices to be inserted increases, CGAL consumes significantly a greater amount of RAM memory in contrast to the proposal. On the other hand, you can also notice the computational efficiency of our proposal.

In Figures 4.10 and 4.11, you can see the delay times as well as the use of RAM memory, for the given proposal and its naive implementation in CGAL. In both cases, it is confirmed in practice, the asymptotic differences between both implementations.

Table 4.12 Comparison of Efficiency (in seconds) and RAM Memory Consumption (in Kbytes) Between CGAL and the Proposal for Complete Persistence.

	Vertices	500	1000	3500	10000	15000
Proposed Method	RAM	24	1875	6573	11.764	14.652
	CPU	0.103	0.431	3.752	4.375	6.869
CGAL Method	RAM	28	61.212	98.431	6.186.564	8.256.432
	CPU	0.365	0.984	13.457	102.587	150.768

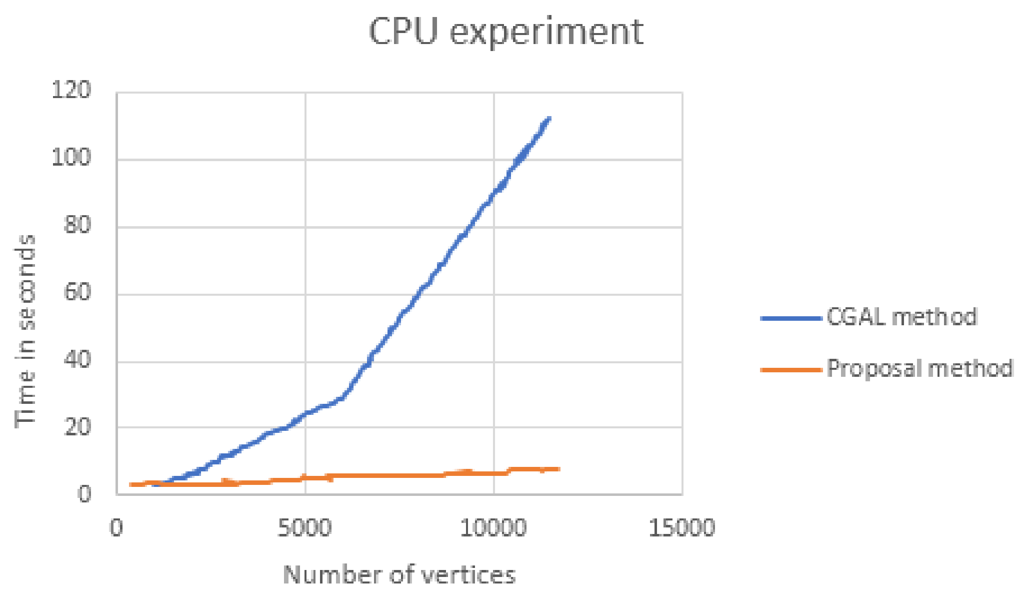

Figure 4.9 Comparison of Execution Time (in seconds) Between Proposal Method and CGAL for Full Persistence.

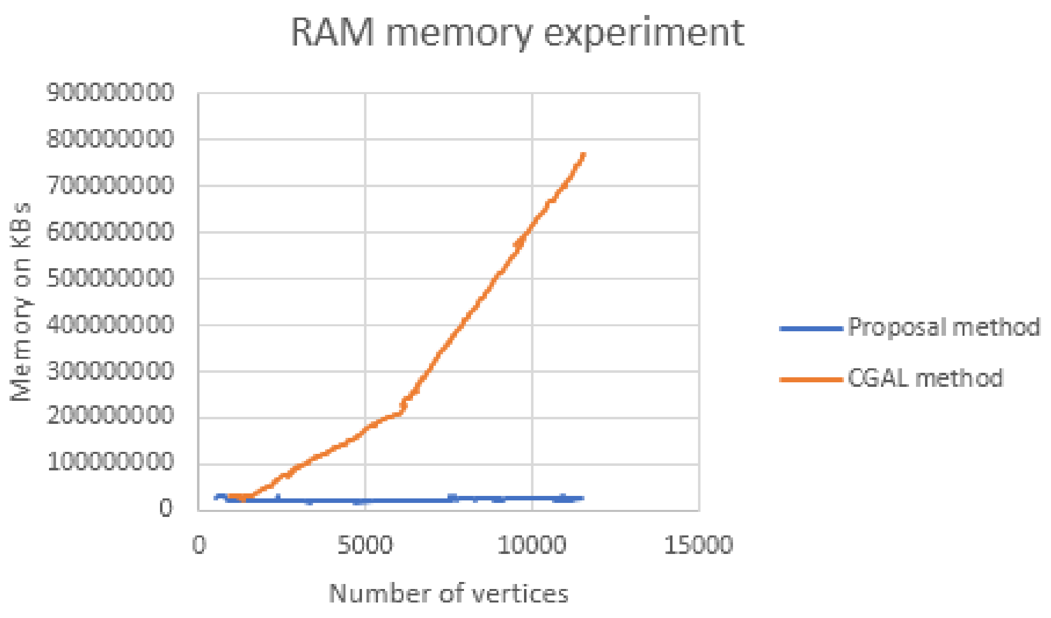

Figure 4.10 RAM Memory Comparison (in Kbytes) Between Our Proposal and CGAL for Complete persistence.

4.4. THE ALGORITHM COMPLEXITY ANALYSIS

This section analyzes the computational complexity of the proposed data structures. The main difference between the Walk, Hybrid and Graph methods is given by the search procedure for the triangle that contains the inserted vertex in the Delaunay triangulation. The Walk method chooses a random vertex on which it makes a journey until finding a vertex of the triangle containing the vertex to be inserted. While the Hybrid and Graph methods use the structure described in subsections 3.2 and 3.3, the difference between these two methods is due to the way persistence is maintained, since, in the Graph method, only one use is made. Data structure to store all the triangles generated during the triangulation, as well as the corresponding time, in which they were created.

In the three proposed methods, the only one capable of supporting the removal operation is the Walk method. This is because the other two proposed methods Graph, and Hybrid still lack an optimal way to perform this operation. Each routine is then evaluated separately.

1. Label assignment method: For this method, the algorithm described in subsection 3.1.1 is used, which uses a hash table, whose complexity is O (1). This routine is used in the Walk and Hybrid method.
2. Triangles around a vertex loading method: The complexity of this routine is $O\ (klogk)$ where k is the number of neighbors for a vertex. This routine is used in the Walk method since the number of adjacent triangles is usually amortized constant, we can consider that the cost of this procedure is a constant C.

3. Search method for a triangle in a triangulation: This algorithm varies depending on the proposed method. In the case of walk, it has a computational complexity of $\sqrt{n}$, however, for the hybrid and Graph methods, its computational time is $\theta(\log(n))$.

The triangle load depends on the number of neighboring vertices of a vertex in a Delaunay triangulation, it is usually constant. Therefore, the Walk method has an asymptotic complexity greater than $O(C)$ when loading the triangles affected by the vertex to be inserted. In the case of the Hybrid and Graph, this cost is 0, since all the triangles are already loaded in the structure.

Then the final complexity of the persistent data structures proposed when inserting a vertex is $O\left(\sqrt{n}\right)$ for the Walk method, $O(log\ (n))$ for the Hybrid method and $\theta(\log(n))$ for the Graph method.

5. CONCLUSION AND DISCUSSION

5.1. CONCLUSION

This dissertation deals with the field of Delaunay triangulations. Different types of data structures were proposed and developed for Delaunay triangulations.

The first implementation was used the DAG data structures with incremental insertion algorithm. The time measurements of the execution of the improved incremental algorithm has shown that the actual execution time is within the theoretical. However, choosing a DAG for a data structure is not the most optimal choice for incremental construction algorithms. Guibas and Stolfi [55] have presented a quad edge data structure designed to triangulate vertex in two- and three-dimensional spaces. O. Devillers [47] presented a data structure of the Delaunay hierarchy that operates on the location of a vertex that is relative to its closest neighbors. The structure takes up a much smaller percentage of memory compared to DAG. Upgrading our algorithm with an improved data structure would optimize memory consumption, thus increasing the complexity of the algorithm. Broutin, Devillers, and Hemsley derived the theoretical limit for the asymptotic behavior of the maximum vertex rate in Delaunay triangulation as the number of vertices increases [53]. Analysis of the results of our measurements showed that, in practice, the highest vertex rate will remain below the expected limit for a large number of vertices, which is estimated at 4000 vertices in the selected sample. The theoretically expected value of the average vertex rate should not exceed 6. The results of this dissertation's measurements confirm this. By increasing the number of vertices, in practice, the value of the average vertex rate asymptotically approaches the expected value of 6 and does not exceed it. With increasing the number of vertices, both the highest and the average rate vertex gradually increases. Since the point patterns are arranged at the same interval, it can be concluded that the maximum and average point rates depend on the density of the vertices distribution.

Where the second and third implementations presents three data structures (Walk, Hybrid and Graph) that deals with Walk, Hybrid and Graph. The Walk method has a computational cost, for the insertion operation, of $O(\sqrt{n})$ and, a computational cost for the elimination operation of $O\ (klog\ (k))$; where k is the number of adjacent vertices and n is the total number of vertices. For the Hybrid method, the insertion algorithm has a complexity of $O\ (log\ (n))$, however, it does not support deletion. Finally, the Graph method has an insertion cost of $\theta(log\ (n))$ and does not support deletion either.

In addition, correction evaluations of each of the proposed algorithms were carried out and tests with random image and vertex databases were developed, reaching the following conclusions:

The data structure, in its partial version, is competitive with respect to other data structures raised and even better than those that do not have persistence, the latter being the big difference. In addition, it can be used without using persistence.

In the full version, the data structure shows an advance in the state of the art, this is because, in the literature, there is currently no data structure for Delaunay triangulation with complete persistence.

In the partial version, the Graph method demonstrated a better performance in relation to the Walk and Hybrid methods, both in computational cost and in the use of RAM. However, Graph does not present persistent elimination. In comparison, the

Walk method, although it is the least efficient of the other two, has persistent elimination and, since the efficient insertion algorithm is related to the performance of the search algorithm of the triangle that contains the vertex to be inserted, which is $O(\sqrt{n})$, so it is possible to create a more efficient persistent search algorithm to improve the results of this method.

5.2. LIMITATION

In the fully persistent version, the edge loading operation still has problems with its performance. This is due to the need to store information about the version number and start time and end time values at which there are adjacent edges with other vertices.

Performing the search for the triangle that contains the vertex to be inserted has an amortized complexity of $O(\sqrt{n})$. Therefore, the possibility of minimizing this dimension through a more efficient search algorithm is still pending.

APPENDIX

ALGORITHMS

Algorithm (1): PS-Search (Vertex p, Internal Structure T, Integer t).

```
Data: Vertex $p$, Internal Structure $DT$, Integer $t$
Result: Edge E
    1.  v: = Random Vertex $(DT, t)$;
    2.  E: = DT-LoadEdges $(v, DT, t)$;
    3.  E: = Sort $(E, p)$;
    4.  e: = takes an edge$\in E$;
    5.  if Right Of $(X, e)$ then
    6.  e: = Sym.e
    7.  end;
    8.  while true do
    9.  if X = e. Dest or X=e.Org then
    10. return e;
    11. else
    12. p: = 0;
    13. if not Right Of (X, e.O next) then
    14. p: =1;
    15. if not Right Of (X, e.D prev) then
    16. p: =2;
    17. if p is 0 then
    18. return e;
    19. else
    20. if p is 1 then
    21. e: = e.O next;
    22. else
    23. if p is 2 then
    24. e: = e.D prev;
    25. else
    26. if dist (e.O next, X)<dist (e.D Prev, X) then
    27. e: = e.O next;
    28. else
    29. e: = e.D Prev;
return e
```

The Algorithm 1 shows the proposed method to search for a near vertex in the persistent triangulation. For this, the input data is: the vertex p to insert, the Delaunay triangulation persistent data structure and an integer value t, which indicates the time at which the insert operation will be performed.

Inline 1 of the algorithms, the RandomSpeed function is executed, which returns, at random, a vertex of the structure. This procedure is performed in $O(1)$ order because the vertices are indexed in a hash table. Once the vertex v is loaded; then, in line 2, all the edges of this vertex are retrieved, which are stored in a balanced search tree, as long as the creation time of the adjacent edges are within the range of parameter T. This procedure takes a time of $O(k)$, where k is the number of edges adjacent to v. Line 3 orders the edges of v counterclockwise and the ordering is done in a time $klog(k)$. Inline 4, an edge of E is taken and, from line 5 onwards, the algorithm is the same as that proposed by [38], with the additional difference, that each time a visit is required a new vertex, both that vertex and its adjacent ones must be loaded from the internal structure.

Algorithm (2): DT-Insert Vertex (Internal Structure DT, Vertex x, integer t).

Data: Internal Structure DT, Vertex x, integer t
Result: Internal Structure DT with x inserted at time t

```
1.  $id_x$= Get Id $(x, DT)$;
2.  Assert $(id_x \notin DT)$;
3.  E = PS-Search $(DT, x, t)$;
4.  $id_p$= E.org;
5.  $id_q$=E. dest;
6.  $id_r$=get-vertex (E, p);
7.  stack A;
8.  stack V;
9.  temp-tri = create DT ();
10. create-edge (DT, t, $id_x$, $id_p$);
11. create-edge (DT, t, $id_x$, $id_q$);
12. create-edge (DT, t, $id_x$, $id_r$);
13. A. insert $(id_p, id_q)$;
14. V. insert $(id_x)$;
15. A. insert $(id_q, id_r)$;
16. V. insert $(id_x)$;
17. A. insert $(id_r, id_p)$;
18. V. insert $(id_x)$;
19. for E=A.pop (),$id_0$=V.pop () do
20.     $id_{ca}$=E.org;
21.     $id_{cb}$=E. dest;
22.     Face FH=temp-tri. face $(id_{ca}, id_{cb}, id_0)$;
23.     Assert (FH! = NULL);
24.     $id_{op}$=FH. Neighbor (FH. index $(id_0)$);
25.     If temp-tri. Illegal-Edge (E) then
26.         DT-Load Triangles (temp-tri, Dt, o, t);
27.         temp-tri.Flip (E);
28.         DT.update-time (E);
29.         DT.create-Edge $(id_0, id_{op}, t)$;
30.         A. insert $(id_{op}, id_{ca})$;
31.         V. insert $(id_{cb})$;
32.         A. insert $(id_{op}, id_{cb})$;
33.         V. insert $(id_{ca})$;
```

Algorithm 2 presents the modification to the classical incremental algorithm for Delaunay triangulations, to work in the internal data structure, which is part of the proposal for the persistent Delaunay triangulation.

Line 1 of the algorithm, gets the identifier of the balanced binary tree that stores information about the edges of vertex p. It is important to note that the first time this tree is empty. Line 3 returns in E one of the three edges that form the triangle that contains p. Lines 4, 5 and 6 obtain the identifiers of three vertices of the triangle that contains vertex p. In lines 10 to 18, the edges that make up the triangle are created and inserted together with their vertices in arrays A and V respectively. These edges will be used to analyze whether they are legal or illegal edges. The analysis process runs on lines 19 through 33. The most important part of this code segment is found on lines 25 through 33; Here, we directly evaluate whether an edge is illegal and if it is, only the triangles that will be affected by the flip operation and that can subsequently trigger new flip operations are loaded.

Algorithm 3: DT-Load Edges (Vertex p, Internal Structure DT, Integer t).

```
Data: Vertex $p$, Internal Structure $DT$, Integer $t$
Result: Arrangement $V$
  1. Arrangement V;
  2. $id$ = GetId ($p$, $DT$);
  3. for each node $x$ in the tree connected with $id$ do
  4.     Add $x$: $DV$ to $V$;
  5. Sort ($V, p$);
  6. return $V$
```

In Algorithm 3, line 2 returns the identifier of the balanced search binary tree, which is indexed by p and is stored in the internal structure. The time required to recover this vertex is $O(1)$. Subsequently, in lines 3, 4 and 5, all vertices are obtained within the search binary tree connected with p.

Algorithm 4: DT-Load Triangles (Triangulation T, Internal Structure DT, Vertex p, Integer t).

```
Data: Triangulation $T$, Internal Structure $DT$, Vertex $p$, Integer $t$
Result: loaded Triangulation $T$ with triangles adjacent to $p$
    1. Array $V$ = DT-Load Edges $(p, DT, t)$;
    2. Sort $(V, p)$;
    3. for Each Vertex $q$ in $V$ do
    4.       T. Create (triangle ($p$, q Previous, $q$);
    5. for Each Vertex $q$ in $V$ do
    6.       triangle $tr$ = T. get Reference (triangle ($p$, $q$ Previous, $q$));
    7.       neighbor.tr = Next Triangle;
    8.       neighbor.tr (q) = Previous Triangle;
```

Once the edges adjacent to vertex p are loaded, it is necessary to regenerate the triangle formed by these edges. To achieve this, Algorithm 4 is applied. This algorithm, on line 1, obtains all the edges adjacent to vertex p, later on, lines 3 and 4 the triangles are created within the current triangulation t. On lines 5 through 8, the neighborhood information is initialized for each vertex of each triangle. This in order to guarantee the correct operation of the incremental algorithm.

Algorithm 5: DT-Delete (Internal Structure DT, Vertex p, Integer t).

```
Data: Internal Structure $DT$, Vertex $p$, Integer $t$
Result: DT with removed $p$ at time $t$
    1.  Array $P$ = DT-Load Edges $(p, DT, t)$;
    2.  Ears List;
    3.  Min-Heap $PQ$;
    4.  N = P.size ()
    5.  for $i = 1; i < N;\ i++do$
    6.       current ear = Ear $(i, (i\ +\ 1)$ mod N, $(i\ +\ 2)$ mod N);
    7.       if Counterclockwise (current) then
    8.           PQ.insert (Power ($p$, Get-Vertices ($P$, ear)), current);
    9.  while PQ.empty () == false do
    10.      current ear = PQ.min ();
    11.      if isEar (Ears, ear) then
    12.          add $(DT, ear, t)$;
    13.          creation $t$
    14.          next ear = Get-Next (Ears, current);
    15.          previous ear = Get-Previous (Ears, current);
    16.         PQ.insert (Power ($p$, Get-Vertices ($P$, next)), next));
    17.         PQ.insert (Power (p, Get-Vertices (P, previous)), previous));
    18.        Update (Ears, current);
```

Algorithm 6: Insert Vertex (Structure D, Vertex x).

```
Data: Structure $D$, Vertex $x$
Result: Structure $D$ with inserted $x$
    1.  Triangle tr = Find Triangle (D.root);
    2.  Vertex $p$ = $tr \geq p0$;
    3.  Vertex $q$ = $tr \geq p1$;
    4.  Vertex $r$ = $tr \geq p2$;
    5.  Triangle newchild0 = Triangle $(p, q, x)$;
    6.  Triangle newchild1 = Triangle $(q, r, x)$;
    7.  Triangle newchild2 = Triangle $(q, r, x)$;
    8.   $tr \geq insert\ childs.(newchild0)$;
    9.  $tr \geq insert\ childs.(newchild1)$;
    10. $tr \geq insert\ childs.(newchild2)$;
```

Algorithm 7: Find Triangle (Triangle tr, Vertex x).

Data: Triangle tr, Vertex x

Result: Triangle tr containing x

```
1. if $tr \geq is - Sheet()$ then
2.      return $tr$;
3. for Each Triangle ctr in tr do
4.    if $ctr \geq contains\ (x)$ then
5.       return Find Triangle $(ctrl, x)$;
```

In Algorithm 7 inline 1 we arrive at the base case, in which we have arrived at the triangle that contains x, from lines 3 to 5, we search for a triangle recursively (in the child nodes) that contains x.

Algorithm 8: LCA-Calculate (Matrix M, Depth Array, Vertex p, Vertex parent-p).

Data: Matrix M, Depth Array, Vertex p, Vertex parent-p

Result: M with inserted p

```
1. id = GetId $(p, DT)$;
2. $pid$ = GetId $(parent - p, DT)$;
3. $M[id][0] = pid$;
4. depth $[id]$ = depth $[pid]$ + 1;
5. for i=1; $i < \log 2$ (depth $[id]$); $i++$ do
6.      $M[id][i] = M\big[M[id][i-1]\big][i-1]$;
```

In Algorithm 8, all 2^i ancestors of p are calculated. In lines 1 and 2, indexed labels are obtained through the vertices. In line 3 the base case of the recursion is considered, that is, the ancestor of p is its father. In line 5, the depth of node p is calculated, this to obtain the $i-th$ ancestor of p; Subsequently, in lines 5 and 6 the 2^i ancestor of p is calculated, for $i > 0$ which defines 2^i ancestor of p as 2^{i-1} ancestor of 2^{i-1} ancestor of p.

Algorithm 9: Random Ancestor (Matrix M, Depth Fix, Vertex p).

Data: Matrix M, Depth arrangement, Vertex p

Result: M with inserted p

```
1. Vertical Random $id$ = Get Index ($p$);
2. ancestor  = Random (depth [$id$]);
3. for $i = 0; i < \log 2\ (depth\ [id]); i++do$
4.     if ancestor $(1 < j)$ then
5.         randomVerticeid = $M$ [randomVerticeid] [$i$];
6. return RandomVerticeid;
```

In Algorithm 9, an ancestor random node of p is calculated. Inline 1, the label of p is obtained. Inline 2, a random number is obtained, which indicates the $i-th$ ancestor of p. Inline 3 the result is initialized, which must be initialized as p, in lines 4 to 6, the $i-th$ ancestor is decomposed into sums of powers of 2, in line 5, we ask if in the binary representation of ancestor the bit corresponding to the position i is on, if it is, idVerticeRandom is updated as its $i-th$ ancestor.

BIBLIOGRAPHY

1. Mehta, D.P. and S. Sahni, *Handbook of data structures and applications*. 2004: Chapman and Hall/CRC.

2. Fiat, A. and H. Kaplan, *Making data structures confluently persistent.* Journal of Algorithms, 2003. **48**(1): p. 16-58.

3. Demaine, E.D., S. Langerman, and E. Price. *Confluently persistent tries for efficient version control*. in *Scandinavian Workshop on Algorithm Theory*. 2008. Springer.

4. Shamos, M.I. and D. Hoey. *Closest-point problems*. in *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. 1975. IEEE.

5. Razafindrazaka, F.H., *Delaunay triangulation algorithm and application to terrain generation.* postgraduate diploma.–African Institute for Mathematical Sciences May, 2009.

6. Aurenhammer, F., *Voronoi diagrams—a survey of a fundamental geometric data structure.* ACM Computing Surveys (CSUR), 1991. **23**(3): p. 345-405.

7. Boissonnat, J.-D., et al. *Triangulations in CGAL*. in *Proceedings of the sixteenth annual symposium on Computational geometry*. 2000.

8. Mohammed-Ali, W., C. Mendoza, and R.R. Holmes, *Riverbank stability assessment during hydro-peak flow events: The lower Osage River case (Missouri, USA).* International Journal of River Basin Management, 2020: p. 1-9.

9. Hjelle, Ø. and M. Dæhlen, *Triangulations and applications*. 2006: Springer Science & Business Media.

10. Rippa, S., *Minimal roughness property of the Delaunay triangulation.* Computer Aided Geometric Design, 1990. **7**(6): p. 489-497.

11. Faugeras, O.D., E. Le Bras-Mehlman, and J.-D. Boissonnat, *Representing stereo data with the delaunay triangulation.* Artificial Intelligence, 1990. **44**(1-2): p. 41-87.

12. Cheng, S.-W., T.K. Dey, and J. Shewchuk, *Delaunay mesh generation*. 2012: CRC Press.

13. Eddy, W.F., *A new convex hull algorithm for planar sets.* ACM Transactions on Mathematical Software (TOMS), 1977. **3**(4): p. 398-403.

14. McHugh, J.A., *Algorithmic graph theory*. Vol. 68056. 1990: Citeseer.

15. Saramäki, J., et al., *Generalizations of the clustering coefficient to weighted complex networks.* Physical Review E, 2007. **75**(2): p. 027105.

16. Lipton, R.J. and R.E. Tarjan, *A separator theorem for planar graphs.* SIAM Journal on Applied Mathematics, 1979. **36**(2): p. 177-189.

17. Dirichlet, G.L., *Über die Reduction der positiven quadratischen Formen mit drei unbestimmten ganzen Zahlen.* Journal für die reine und angewandte Mathematik, 1850. **1850**(40): p. 209-227.

18. Thiessen, A.H., *Precipitation averages for large areas.* Monthly weather review, 1911. **39**(7): p. 1082-1089.

19. Delaunay, B., *Sur la sphere vide.* Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk, 1934. **7**(793-800): p. 1-2.

20. Snoeyink, J. and M. Van Kreveld. *Linear-time reconstruction of Delaunay triangulations with applications*. in *European Symposium on Algorithms*. 1997. Springer.

21. Pocchiola, M. and G. Vegter. *Pseudo-triangulations: theory and applications*. in *Proceedings of the twelfth annual symposium on Computational geometry*. 1996.

22. Driscoll, J.R., et al. *Making data structures persistent*. in *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 1986.

23. Overmars, M.H., *Searching in the past ii: general transforms*. 1981, Tech. Rep. RUU.

24. De Berg, M., et al., *Visibility graphs*, in *Computational geometry*. 2000, Springer. p. 307-317.

25. Mirzaian, A., *Triangulating simple polygons: Pseudo-triangulations*. 1988, Citeseer.

26. Ramaswami, S., *Convex hulls: Complexity and applications (a survey).* Technical Reports (CIS), 1993: p. 264.

27. Aichholzer, O., W. Mulzer, and A. Pilz, *Flip distance between triangulations of a simple polygon is NP-complete.* Discrete & computational geometry, 2015. **54**(2): p. 368-389.

28. Chew, L.P., *Constrained delaunay triangulations.* Algorithmica, 1989. **4**(1-4): p. 97-108.

29. Fortune, S., *Voronoi diagrams and Delaunay triangulations*, in *Computing in Euclidean geometry*. 1992, World Scientific. p. 193-233.

30. Preparata, F.P. and M.I. Shamos, *Computational geometry: an introduction*. 2012: Springer Science & Business Media.

31. Cignoni, P., C. Montani, and R. Scopigno, *DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed.* Computer-Aided Design, 1998. **30**(5): p. 333-341.

32. Avis, D., D. Bremner, and R. Seidel, *How good are convex hull algorithms?* Computational Geometry, 1997. **7**(5-6): p. 265-301.

33. Demaine, E.D., et al., *Dynamic optimality—almost.* SIAM Journal on Computing, 2007. **37**(1): p. 240-251.

34. Buchsbaum, A.L. and R.E. Tarjan, *Confluently persistent deques via data-structural bootstrapping.* Journal of Algorithms, 1995. **18**(3): p. 513-547.

35. Kaplan, H., C. Okasaki, and R.E. Tarjan, *Simple confluently persistent catenable lists.* SIAM Journal on Computing, 2000. **30**(3): p. 965-977.

36. Gärtner, B. and M. Hoffmann, *Computational Geometry Lecture Notes1 HS 2012.* Dept. of Computer Science, ETH, Zürich, Switzerland, 2013.

37. Roussopoulos, N., S. Kelley, and F. Vincent. *Nearest neighbor queries*. in *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. 1995.

38. Brown, P.J. and C.T. Faigle, *A robust efficient algorithm for point location in triangulations*. 1997, University of Cambridge, Computer Laboratory.

39. Sarnak, N. and R.E. Tarjan, *Planar point location using persistent search trees.* Communications of the ACM, 1986. **29**(7): p. 669-679.

40. Dietz, P.F. *Fully persistent arrays*. in *Workshop on Algorithms and Data Structures*. 1989. Springer.

41. Tarjan, R.E., *Amortized computational complexity.* SIAM Journal on Algebraic Discrete Methods, 1985. **6**(2): p. 306-318.

42. van Emde Boas, P., R. Kaas, and E. Zijlstra, *Design and implementation of an efficient priority queue.* Mathematical systems theory, 1976. **10**(1): p. 99-127.

43. Dietzfelbinger, M., et al., *Dynamic perfect hashing: Upper and lower bounds.* SIAM Journal on Computing, 1994. **23**(4): p. 738-761.

44. Blelloch, G., et al., *Persistent triangulations.* Journal of Functional Programming, 2001. **11**(5): p. 441-466.

45. Seidel, R. *Constructing higher-dimensional convex hulls at logarithmic cost per face*. in *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 1986.

46. Lischinski, D., *Incremental delaunay triangulation.* Graphics gems IV, 1994: p. 47-59.

47. Devillers, O., *The delaunay hierarchy.* International Journal of Foundations of Computer Science, 2002. **13**(02): p. 163-180.

48. Bowyer, A., *Computing dirichlet tessellations.* The computer journal, 1981. **24**(2): p. 162-166.

49. Watson, D.F., *Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes.* The computer journal, 1981. **24**(2): p. 167-172.

50. Kao, T., *Dynamic maintenance of Delaunay triangulations.* work, 1991. **9**: p. 32.

51. Bender, M.A. and M. Farach-Colton. *The LCA problem revisited*. in *Latin American Symposium on Theoretical Informatics*. 2000. Springer.

52. Bern, M., D. Eppstein, and F. Yao, *The expected extremes in a Delaunay triangulation.* International Journal of Computational Geometry & Applications, 1991. **1**(01): p. 79-91.

53. Broutin, N., O. Devillers, and R. Hemsley, *The Maximum Degree of a Random Delaunay Triangulation in a Smooth Convex.* 2014.

54. Yi-bo, L. and L. Jun-Jun, *Harris corner detection algorithm based on improved contourlet transform.* Procedia Engineering, 2011. **15**: p. 2239-2243.

55. Guibas, L.J., D.E. Knuth, and M. Sharir, *Randomized incremental construction of Delaunay and Voronoi diagrams.* Algorithmica, 1992. **7**(1-6): p. 381-413.

VITA

Esraa Habeeb Khaleel Al-Juhaishi was born in Baghdad, Iraq. She received her bachelor’s degree in Mathematics and Computer Applications Science in 2003 from Al-Nahrain University, Bagdad, Iraq. Esraa received her master's degree in Mathematics and Computer Applications Science in 2006 from Al-Nahrain University, Bagdad, Iraq. In 2009 Esraa started working as a faculty member in the college of Mathematics and Computer Science at Tikrit University, Iraq. In 2014, Esraa awarded a Ph.D. scholarship by the Ministry of Higher Education and Scientific Research of Iraq and started a Ph.D. degree at Missouri University of Science and Technology. In 2019 she received her second master's degree in Applied Mathematics from the Missouri University of Science and Technology. Esraa received her third master's degree in Computer Science from Missouri University of Science and Technology in 2020. In August 2020, Esraa received her Ph.D. in Computer Science from Missouri University of Science and Technology.

.