Graph-Theoretic and Computational Geometry Approaches for the Management of Sensor Networks in Process Monitoring

M. Millán[1], M. Ceballos[1,*] and L. Orihuela[2,3]

[1]Dpto. de Ingeniería.
Universidad Loyola Andalucía.
Av. de las Universidades, s/n, 41704 Dos Hermanas, Sevilla (Spain).
mmillangordillo@al.uloyola.es mceballos@uloyola.es
* Corresponding author.
[2]Dpto. de Ingeniería Electrónica, Sistemas Informáticos y Automática.
Universidad de Huelva.
Av. de las Fuerzas Armadas, s/n, 21007, Huelva, (Spain).
[3]Centro de Investigación en Tecnología, Energía y Sostenibilidad, Universidad de Huelva, Carr. Huelva-Palos de la Frontera, s/n, 21819, Palos de la Frontera, Huelva (Spain).
luis.orihuela@diesia.uhu.es

Abstract

This paper integrates graph-theoretic methods and computational geometry to model sensor deployments as weighted Delaunay graphs, capturing spatial proximity and communication costs. Several strategies are proposed to address different and common problems related to the management of sensor networks, including battery management, reconfiguration of network communication topologies, efficient data collection, etc. In particular, six algorithmic methods are presented, combining minimum spanning trees, shortest-path computations, edge contraction, graph labeling, Voronoi diagram segmentation, and Delaunay triangulation. These methods are theoretically supported by some novel results on the hamiltonicity and chromatic properties of Delaunay graphs, ensuring complete traversal routes. The performance and scalability of the framework are shown with a real implementation -monitoring of a sugar cane field-, where the performance and computational requirements of the proposed methods are assessed.

Keywords: Weighted Graph, Voronoi diagram, Delaunay Graph, Sensor network, Algorithm.
2010 Mathematics Subject Classification: 68U05, 05C22, 68M18, 05C85, 68W40.

1 Introduction

Currently, one of the most stimulating and significant areas of research across Science, Engineering, and particularly Mathematics is finding and studying connections between different fields. Novel techniques and methods allow researchers to address unresolved problems, refine existing theories, and achieve groundbreaking results. In this paper, we deal with the link between Graph Theory and sensor networks. Graph Theory provides a powerful framework for modeling and analyzing complex networks, while sensor networks offer a distributed system of interconnected devices that collect and transmit data. The use of a well-established theoretical framework for this application has led to significant advancements in understanding the behavior, optimization and management of sensor networks.

Graph Theory has proven a very useful tool to deal with a wide range of challenges within sensor networks. For example, in [4] the authors study several graph-based optimization approaches such as node placement, routing protocols and data aggregation techniques. They are used to minimize energy consumption, extend network lifetime, and improve overall performance. Graph-based algorithms also seem very useful to deal with network topology construction, data routing, energy-efficient communication and optimize operations in wireless sensor networks [14]. Similar techniques have also been applied to address the load balancing and connectivity problems in wireless sensor networks [11, 18].

Sensor networks are an integral part of the ever-expanding field of Internet of Things (IoT) technology. These networks, composed of spatially distributed sensors, collect, process and transmit data from the physical world. While earlier network configurations emphasized transmitting data to a central server or the cloud for processing, modern sensor network architectures increasingly rely on edge computing. In this paradigm, data is processed locally enabling real-time decision-making and reducing dependency on centralized infrastructure. This shift makes inter-node communication and distributed processing critical for network efficiency and resilience. Wireless sensor networks (WSNs) are especially valuable for real-time monitoring in contexts such as chemical process supervision and environmental surveillance, often operating in hard-to-reach locations. However, their deployment poses significant challenges, particularly related to energy management and network reconfiguration. Most sensor nodes are battery-powered and difficult to replace or recharge, which makes energy efficiency a critical factor

for maintaining network functionality and longevity [2, 24]. The limited power and computational capabilities of individual sensors require careful energy management strategies to maximize their lifespan, avoid data loss, and ensure fault tolerance [13]. To address these challenges, various mathematical battery models have been proposed and implemented in the context of sensor networks [22].

In this work, we address several key challenges in the design and operation of sensor networks, focusing particularly on battery level management and efficient communication. These problems are critical for ensuring both the reliability and the longevity of such systems, where sensor failures or data transmission issues can severely compromise performance.

To tackle these challenges, we rely on a range of tools from Graph Theory and computational geometry. Our approach is rooted in the analysis and exploitation of structural properties of graphs to model network topology, optimize resource allocation, and manage spatial relationships between sensors.

We place particular emphasis on two graph-theoretic properties: Eulerian paths and the chromatic number. Eulerian paths, which traverse each edge exactly once, are especially relevant for minimizing energy consumption in data collection tours such as those carried out by mobile agents in pipeline monitoring or surveillance tasks by reducing redundant movement. On the other hand, the chromatic number informs scheduling and interference avoidance strategies: by assigning distinct channels or activation times to nearby sensors, it helps reduce collisions and idle listening, which are major sources of energy waste in dense deployments like distillation columns or microfluidic systems.

The paper also presents new theoretical results related to these properties. In particular, we analyze conditions under which Eulerian paths exist in Delaunay graphs, and provide bounds and structural insights on their chromatic number. These results enable more efficient design of network protocols and reduce the computational effort needed in practical implementations.

Finally, building on these theoretical foundations, we propose a set of algorithmic strategies that support energy-aware sensor network management. These include procedures for detecting low-battery nodes and adapting the network topology accordingly using minimum spanning trees, shortest paths, and edge contraction techniques. In addition, Voronoi diagrams and Delaunay triangulations are used to model spatial configurations and

guide decision-making in dynamic scenarios.

By integrating these tools, our work offers a comprehensive framework that not only addresses immediate operational challenges but also enhances the theoretical understanding of graph structures in sensor networks.

Throughout this work, we focus on static, planar wireless sensor networks deployed over a two-dimensional region, where sensor nodes are battery-powered and communicate through multi-hop short-range links. Sensors are assumed to be homogeneous in terms of communication capabilities, and the energy cost of communication is modeled as a function of inter-node distance, represented by weighted edges in the associated Delaunay graph. Node mobility, dynamic traffic patterns, and heterogeneous transmission powers are outside the scope of this study. The proposed framework is intended for monitoring-oriented WSNs, such as environmental or agricultural deployments, where periodic data collection and network longevity are primary objectives.

Rather than proposing a detailed physical energy consumption model, this work focuses on graph-based network reconfiguration strategies that explicitly incorporate the battery state of the nodes as a decision variable. The goal is to maintain connectivity, routing feasibility, and structural robustness through topology-aware transformations, rather than to predict absolute energy savings at the hardware level.

The structure of this paper is as follows. In Section 2, some well-known concepts of Graph Theory and Computational Geometry are reviewed. Section 3 explores the connection between Graph Theory and sensor networks, detailing the methodology for associating these fields. Section 4 presents theoretical results concerning the Eulerian character and vertex coloring of Delaunay Graphs associated with sensor networks. Section 5 presents the algorithmic methods for different management-based problems. In Section 6, these algorithms are applied to a real-world example in the field of smart agriculture, demonstrating their performance. After that, Section 7 provides a computational study analyzing the complexity, runtime, and memory usage of each algorithm. At the end of the paper, there is a conclusion section, acknowledgments and references.

2 Preliminaries

In this section, we recall some general concepts on Graph Theory and Computational Geometry bearing in mind that the reader can consult [7] and

[6], respectively for more details.

2.1 Graph Theory

A *graph* is a pair $G = (V, E)$, where V is called the non-empty vertex set (or node set) and E is called the edge set, which is given by unordered pairs of a couple of nodes.

Let us consider the graph $G = (V, E)$ and a vertex $v \in V$. We say that a vertex $u \in V$ is *adjacent* to v if u and v determine an edge in G. This edge is *incident* to the vertices u and v, which are its *endpoints*.

Given the graph $G = (V, E)$, $H = (V', E')$ is a *subgraph* of G if $V' \subseteq V$, $E' \subseteq E$ and every edge of E' is incident to vertices of E'. Given $S \subseteq V$, the subgraph of G *induced* by S is the graph whose vertex set is S and whose edge set consists of all the edges of E whose endpoints belong to S.

The *neighbourhood* of v is the subgraph of G induced by all vertices adjacent to v. The *degree* of a vertex v, $\delta(v)$, is equal to the number of its adjacent vertices.

It is possible to associate a *weight* to each edge and, in such case, we will say that G is a *weighted graph*. The *adjacency or weight matrix* of a weighted graph $G = (V, E)$ is given by $A = (a_{ij})$, where $a_{i,j}$ is the weight of the edge connecting vertex v_i with v_j. In case that both vertices are not adjacent, that weight would be zero. Fon an undirected graph, the matrix A will be symmetric and all the elements in the main diagonal will be zero.

A sequence of consecutive vertices and edges that are mutually adjacent in a graph is known as a *walk or trail*. A graph is *connected* if there is a walk between any pair of vertices. Otherwise, we say that the graph is *disconnected*. A *path* is a walk where all the edges and vertices are different. The *length* of a walk is defined by the number of its edges. A *cycle or circuit* in a graph G is a non-empty walk in which only the first and last vertices are equal. A cycle with length k will be referred to as a *k-cycle*.

Let $G = (V, E)$ be a connected graph. A *cut vertex* or *articulation point* of G is a vertex $v \in V$ such that when v is removed from G, a disconnected graph is obtained (removing a cut vertex from G breaks it in to two or more graphs).

A *Hamiltonian path* in a graph is a walk that visits each vertex exactly once. A *Hamiltonian cycle* (or *Hamiltonian circuit*) is a Hamiltonian path that starts and ends at the same vertex.

A *tree* is a connected graph without cycles. The *spanning tree* of a

graph $G = (V, E)$ is a tree whose set of vertices is V. When G is a weighted graph, the *minimum spanning tree* is defined as the spanning tree having the minimum possible sum of the weights of its edges. Figure 1 shows a weighted graph, followed by a non-minimum spanning tree whose total weight is 11 and a minimum spanning tree with weight 7.

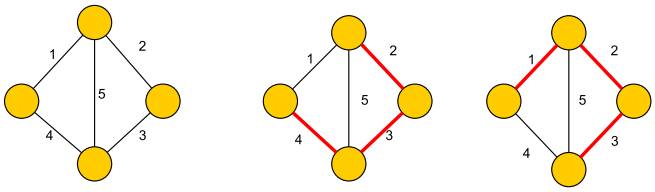

Figure 1: Example of non-minimum and minimum spanning trees in a weighted graph.

A *cycle graph*, C_n, or simply n-cycle [12] is a graph with n vertices containing a single cycle through all of them. A *complete graph* is a graph in which each pair of vertices is connected by an edge. The complete graph with n vertices is usually denoted by K_n. It is a regular graph with degree $n-1$. A *wheel graph* of order n, W_n, or simply n-wheel [10] is a graph that contains a cycle of order $n-1$ and for which every vertex of the cycle is connected to another vertex known as the hub. Therefore $W_n = C_{n-1} + K_1$. A *complete n-partite graph*, $K_{\alpha_1,\alpha_2,\ldots,\alpha_n}$, is a graph where the vertex set can be decomposed into n disjoint sets (no two vertices within the same set are adjacent) such that every pair of vertices in different sets is adjacent. When $n = 2$, we call it a bipartite Graph (see Figures 2 and 3).

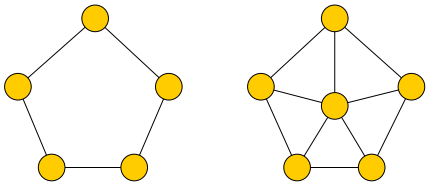

Figure 2: Example of cycle and wheel graphs.

Let G be a graph where e is an edge of G having as incident vertices u and v. The *edge contraction* operation on e is done by removing e while simultaneously merging the two vertices u and v in a new vertex w (see

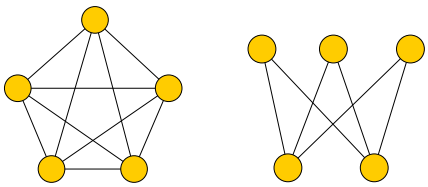

Figure 3: Example of complete and bipartite graphs.

Figure 4).

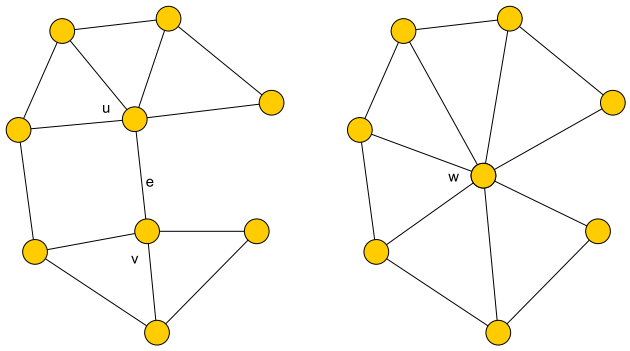

Figure 4: Example of edge contraction.

The resulting graph after applying this operation or the deletion of edges and vertices is called a *minor* of G. The theory of graph minors began with Wagner's theorem [20] which characterizes the planar graphs as being the graphs that do not contain the complete graph K_5 or the complete bipartite graph $K_{3,3}$ as minors. This operation plays an important role in the works of Robertson and Seymour [15, 16, 17]. Their Graph Minor Theorem [8], which explores the properties of graphs that remain invariant under edge contractions and other operations, has broad implications in graph structure theory and computational complexity. Thus, edge contraction is used to study the preservation of connectivity properties. This is significant for designing efficient networks and algorithms. One may ask whether this operation also preserves other properties such as the Eulerian character. This is not true as shown in the following example.

Example 1. *Figure 5 shows a non-Eulerian graph that after applying the contraction of the edge e, we obtain a graph whose vertices have even degree*

and, therefore, is an Eulerian graph.

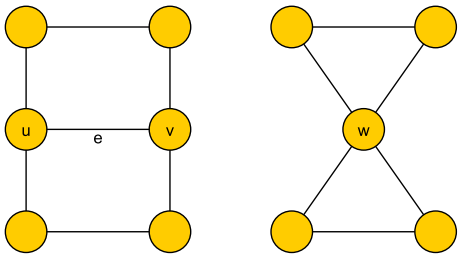

Figure 5: Eulerian character is not preserved under edge contraction.

A *vertex coloring* is an assignment of labels or colors to each vertex of a graph such that there are no adjacent vertices with the same color. A proper k-coloring is a vertex coloring of a graph using k different colors. The minimum number of colors which with the vertices of a graph G may be colored is called the *chromatic number*, denoted by $\chi(G)$. The minimum number required to color the vertices of a graph is called its chromatic number and it is denoted by $\chi(G)$.

2.2 Computational Geometry

Given a set $S = \{p_1, \ldots, p_n\} \subset \mathbb{R}^2$ of non-aligned points (no three different points are located on the same line), the *Voronoi region or diagram* of p_i is given by

$$\mathcal{V}or(p_i) = \{q \in \mathbb{R}^2 \,|\, d(q, p_i) \leq d(q, p_j), \forall j \neq i\}.$$

It holds that $\mathcal{V}or(p_i) = \bigcap_{i \neq j} h(p_i, p_j)$, where $h(p_i, p_j)$ is the region which contains p_i and is bounded by the perpendicular bisector of the segment connecting p_i and p_j. The Voronoi diagram of S is the tessellation of the euclidean plane given by

$$\mathcal{V}or(S) = \{Vor(p_i)\}_{i=1,\ldots,n}.$$

Figure 6 illustrates an example of such diagrams.

The *dual graph* of $\mathcal{V}or(S)$ is defined as the graph having S as set of vertices and two vertices are adjacent if and only if their corresponding Voronoi regions share an edge. This graph is usually called the *proximity graph or Delaunay Graph* of S. From here on, $\mathcal{DT}(V)$ will denote the Delaunay Graph associated to the set of nodes V. If the points are not collinear and no four

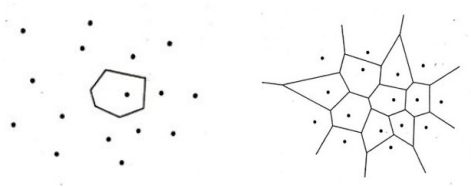

Figure 6: (Left) Voronoi region of one node. (Right) Voronoi diagram.

of them lie on the same circle (i.e., there are no quadruples of cocircular points), then the Delaunay graph forms a triangulation. An example of this is shown in Figure 7.

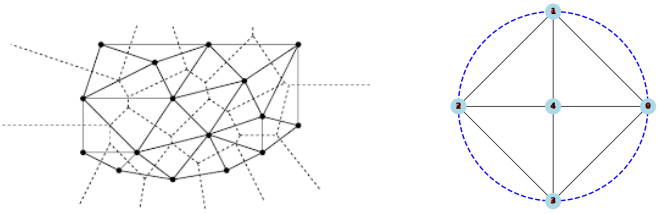

Figure 7: (Left) Delaunay Graph. (Right) Four cocircular vertices.

The *convex hull* of a set of points V is the smallest convex set containing V. For a finite planar set V, the convex hull is a convex polygon. From here on, the bordering cycle of the convex hull of a set of points in the plane V is denoted by $\mathcal{CH}(V)$. Finally, an *internal triangle* is a 3-cycle entirely contained within the convex hull of the vertex set. Equivalently, a triangle is considered internal if at least two of its vertices lie strictly inside $\mathcal{CH}(V)$, or, alternatively, if none of its three edges lies on $\mathcal{CH}(V)$. The set of internal triangles of a given graph G is denoted by $\mathcal{IT}(G)$.

Remark 1. *The notion of internal triangle used in this work is intentionally defined in terms of the convex hull rather than the standard face-based classification of planar embeddings. While internal faces are usually defined with respect to a fixed planar embedding, the hull-based definition adopted here captures whether a triangular configuration is fully enclosed by the network boundary and therefore represents a structurally internal interaction pattern. This distinction is relevant for the proposed graph-based management strategies, which rely on the global geometry of the deployment rather than on a specific planar drawing.*

Figures 8 and 9 illustrate schematic examples of the internal triangle concept.

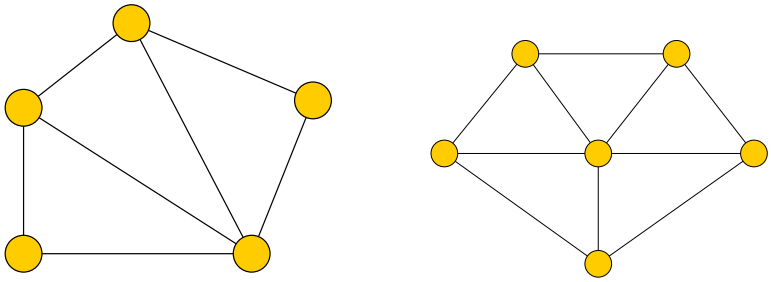

Figure 8: Examples of Delaunay triangulations with no internal triangles. In the first case (left), all vertices lie on the convex hull, whereas this is not the case in the second one (right).

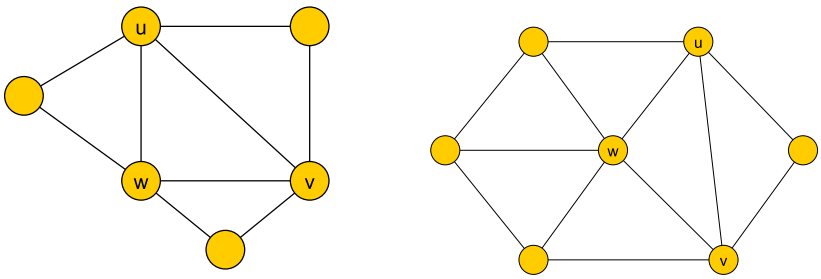

Figure 9: Examples of Delaunay triangulations with one internal triangle determined by vertices u, v and w. In the first case (left), all vertices lie on the convex hull, whereas this is not the case in the second one (right).

3 Sensor networks and Delaunay Graphs

In this section, the connection between Graph Theory and sensor networks is established by modeling the network as a weighted Delaunay graph, an approach used effectively in prior works [9, 19, 21]. Each sensor is represented by a node, and edges are formed based on geometric proximity, reflecting potential communication links. Edge weights are assigned based on Euclidean distances, representing communication costs.

Consider a set of sensors deployed in a planar region. A Delaunay graph is constructed by connecting nodes according to the Delaunay triangulation rules, ensuring good geometric and topological properties, such as avoiding narrow triangles and promoting efficient connectivity. The resulting weighted graph models both spatial layout and communication costs.

Let us focus on two key graph-theoretic properties relevant to sensor networks: Eulerian paths and the chromatic number. Eulerian paths, which traverse each edge exactly once, are useful for designing efficient data collection routes that minimize redundancy and ensure full edge coverage. This is especially beneficial in systems where sensors are arranged along physical infrastructures (such as pipelines, flow lines, or tubing networks) as it helps optimize the movement of mobile data collectors like autonomous robots or drones. The chromatic number is useful for scheduling, interference avoidance, and channel assignment. In particular, it supports the efficient allocation of limited resources (such as assigning distinct communication frequencies or sampling times to nearby sensors) to prevent interference in dense environments like distillation columns, fermenters, or microfluidic systems. Vertex coloring also enables activation scheduling to avoid crosstalk or thermal overlap during synchronized measurements.

From an operational perspective, edge contraction is not a physical merging of sensor nodes, but an abstraction of common reconfiguration mechanisms in wireless sensor networks, such as traffic aggregation or energy-aware rerouting toward a neighbor with higher residual battery. The inherited battery attribute and the 33% threshold represent a simple local policy for delegating communication tasks while preserving connectivity and protecting low-energy nodes.

Analyzing these properties enhances our understanding of network structure and supports designing of robust, energy-efficient, and fault-tolerant methods for real-time monitoring.

The graph-theoretic operations employed in this paper admit a direct operational interpretation in practical WSNs. Edge contraction represents the logical delegation of communication and data forwarding responsibilities from a low-energy node to a neighboring node with higher residual energy, effectively reducing routing overhead without physical node removal. Eulerian and semi-Eulerian paths correspond to efficient data collection or inspection routes, minimizing redundant transmissions or movements of mobile collectors. Vertex coloring models channel, time-slot, or activation scheduling, enabling interference avoidance and energy-aware duty cycling. In this way,

abstract graph transformations are mapped to implementable network actions without requiring changes at the hardware level. Battery thresholds used in the contraction rules (such as the 33% criterion introduced later) model simple local policies for triggering delegation or role reassignment, a common mechanism in energy-aware routing and clustering protocols.

In this work, Eulerian and semi-Eulerian trails are not interpreted as physical trajectories of a mobile agent traversing the network. Instead, edges represent wireless communication links, and an Eulerian trail encodes a logical transmission or scheduling pattern in which each link is activated exactly once per cycle. References to mobile sinks, robots, or drones are provided solely as motivating examples of scenarios where similar graph abstractions arise, but the core framework focuses on multi-hop wireless communication networks.

It is worth noting that the use of weighted edges as an abstraction of communication cost is consistent with power-control mechanisms already deployed in real wireless technologies. In practical WSN and short-range wireless systems, transmission power is dynamically adapted based on link-quality indicators such as RSSI (Received Signal Strength Indicator) or SNR (Signal-to-Noise Ratio). For instance, IEEE 802.11h defines Transmit Power Control (TPC) mechanisms that adjust the transmission power according to measured signal strength, Bluetooth devices employ RSSI-based power control requests between transmitter and receiver, and LTE systems enforce uplink power control so that user devices reach the base station with comparable received power levels regardless of distance. From this perspective, the graph weights used in our framework should be interpreted as an abstract representation of link-quality dependent transmission cost, rather than as literal geometric distances, bridging the gap between graph-theoretic modeling and practical network operation.

From a networking perspective, the proposed battery-aware reconfiguration mechanisms can be interpreted as a combination of dynamic routing, failover, and role migration processes, which are well established in practical network stacks. In particular, the contraction of an edge incident to a low-energy node corresponds to rerouting traffic toward a healthier neighbor and migrating forwarding or aggregation responsibilities, rather than to any physical node removal. Similar mechanisms appear in IP-based mesh networks through link-weight updates and failover in protocols such as OSPF (Open Shortest Path First) or IS-IS (Intermediate System to Intermediate System), as well as in wireless sensor network protocols based on cluster-

ing and role reassignment, including Zigbee Mesh and low-energy adaptive clustering hierarchies (LEACH). In this sense, the proposed graph operations provide a high-level, protocol-agnostic abstraction of these behaviors, focusing on their structural and energy-aware consequences rather than on specific implementation details.

4 Theoretical results

This section derives several theoretical results regarding Eulerian paths and vertex coloring of Delaunay graphs.

First, it is possible to find an upper bound of the number of triangles for a Delaunay Graph according to its planarity and Euler's Theorem as shown in the following

Lemma 1. *If $\mathcal{DT}(V)$ is a Delaunay triangulation of a planar point set V, then the number of triangles satisfies $|T| \leq 2 \cdot |V| - 5$.*

Proof. $\mathcal{DT}(V)$ is planar, therefore, if $\mathcal{DT}(V)$ is a graph with n vertices and m edges satisfying $m \leq 3 \cdot n - 6$. Moreover, according to Euler's Theorem, $|C| + n - m = 2$, where $|C|$ is the number of faces of $\mathcal{DT}(V)$. Since all internal faces are triangles and there is one external face, $|T| = |C| + 1$. Therefore, $|T| = 2 + m - n - 1 = m - n + 1 \leq 3 \cdot n - 6 - n + 1 = 2 \cdot n - 5$. □

The number of triangles, and more concretely the internal ones, is related to the existence of Eulerian trails as we show in

Proposition 1. *If $\mathcal{IT}(\mathcal{DT}(V)) = \emptyset$, then $\mathcal{DT}(V)$ is not Eulerian. Furthermore, either $\mathcal{DT}(V) \cong W_n$ or $V \subseteq \mathcal{CH}(V)$ $(V = V(\mathcal{CH}(V)))$.*

Proof. Since $n > 3$, $\mathcal{CH}(V) \neq \mathcal{DT}(V)$, that is, $\mathcal{DT}(V)$ is not a 3-cycle. Now, we distinguish several cases.

Case 1: $\exists u \in V / u \notin \mathcal{CH}(V)$

Then, $\forall v \in V$ with $v \neq u$, it holds that $v \in \mathcal{CH}(V)$. Indeed, if $v \in V$ with $v \neq u$ verifies that $v \notin \mathcal{CH}(V)$, then there exists $w \in V$ such that $uvwu$ is a 3-cycle. Furthermore, $uvwu$ is an intern triangle since $u \notin \mathcal{CH}(V)$ and $\mathcal{IT}(\mathcal{DT}(V)) = \emptyset$, but this is a contradiction. Therefore, there exists a unique $u \in V - \mathcal{CH}(V)$, $\mathcal{CH}(V)$ has $n - 1$ vertices and $\mathcal{DT}(V) \simeq W_n$.

Case 2: $V \subseteq \mathcal{CH}(V)$

Then, $\mathcal{DT}(V)$ is the cycle C_n where $n = |V|$ adding some edges until creating the triangulation. This set of edges forms a connected graph and

acyclic, that is, a tree T (otherwise we would have vertices which are not in $\mathcal{C}H(V)$). Therefore, $\mathcal{D}T(V) = C_n + T$. Since a tree has at least 2 leaves, $\exists i, j \in \{1, ..., n\}$ such that v_i and v_j are leaves of T. Therefore, v_i, v_j are vertices with degree 3 in $\mathcal{D}T(V)$ and G is not Eulerian.

□

The following lemma shows the possible values for the chromatic number of a Delaunay graph. Please remember that the chromatic number is key for aspects such as interference avoidance and channel assignment in sensor networks.

Lemma 2. *Let G be a Delaunay graph. Then $\chi(G)$ equals 3 or 4.*

Proof. First, note that G is composed of triangles. Graphs formed entirely by triangles have a chromatic number of at least 3, so $\chi(G) \geq 3$. Moreover, G is a planar graph and, according to the well-known Four color Theorem (see [5]), it is satisfied that $\chi(G) \leq 4$. Consequently, $3 \leq \chi(G) \leq 4$. □

Next, several results are given concerning sufficient conditions under which the chromatic number is three.

Proposition 2. *Let G be a Delaunay graph. If G is Eulerian, then $\chi(G) = 3$.*

Proof. Let G be a Delaunay Eulerian graph. According to Euler's Theorem, all the vertices of G have even degree. Let us denote by V the set of vertices of G. Then V can be decomposed as $V = V' \cup V''$, where V' is the set of vertices of degree two and V'' is the set of vertices with even degree greater or equal to four. Now, we show how to develop the vertex coloring of all the vertices in V''. Notice that the vertices in this set are common vertices of at least three triangles. In this way, we start by choosing one of these vertices and go through all its adjacent triangles coloring the vertices alternatively while ensuring that no adjacent vertices have the same color. This is possible due to the fact that all the vertices have even degree. If there is a vertex with odd degree, we will not succeed in this procedure with only three colors. After this, we continue coloring the vertices from the set V'. Let us note that there are no pairs of vertices with degree two that are mutually adjacent, since in that case G would contain an internal face which is not a triangle. Due to this, every vertex in $x \in V'$ is adjacent to an internal triangle and can be colored according to the two colors already chosen for the other two vertices adjacent to x.

□

Proposition 3. *Let G be a planar Delaunay graph whose faces are all triangles. If G does not contain an even wheel W_n as a subgraph, then G satisfies a sufficient structural condition for the application of the 3-coloring procedure described in Proposition 2.*

Proof. The absence of even wheel subgraphs eliminates a well-known family of local configurations that obstruct simple 3-coloring strategies in planar triangulated graphs. Under this structural restriction, the coloring procedure of Proposition 2 can be applied without encountering such obstructions, yielding a proper 3-coloring in practice. This condition is used here as a sufficient criterion within the algorithmic framework rather than as a full characterization of 3-colorable planar triangulations. □

Corollary 1. *If all the vertices of a Delaunay graph G belong to $\mathcal{CH}(G)$, then G satisfies the sufficient structural conditions for the application of the proposed 3-coloring procedure.*

Proof. Let $G = (V, E)$ be a Delaunay graph such that $V \subset \mathcal{CH}(G)$. Then G does not contain any wheel graph W_n as a subgraph for $n \geq 4$. Therefore, G satisfies the hypothesis of Proposition 3, and the 3-coloring procedure described therein can be applied. □

Corollary 2. *If a Delaunay graph G satisfies $\mathcal{IT}(G) = \emptyset$, then G satisfies the sufficient conditions for the application of the proposed 3-coloring procedure.*

Proof. If G has no internal triangles, then all vertices of G belong to $\mathcal{CH}(G)$. The result follows from Corollary 1. □

Remark 2. *The converse of the previous corollary is not true. Moreover, the absence of internal triangles is not equivalent to the condition that all vertices of G belong to its convex hull. For instance, the Delaunay graph in Figure 10 contains one internal triangle and still admits a proper 3-vertex coloring, as shown in the figure.*

Finally, one may wonder if edge contraction operation preserves the chromatic number. This is not true as we can see in the following result.

Proposition 4. *Let G be a graph and let G' be the graph obtained by contracting an edge $e = uv$ in G. Then the chromatic number of G' satisfies*

$$\chi(G') \leq \chi(G).$$

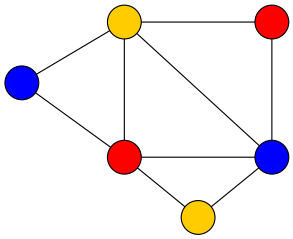

Figure 10: Delaunay graph with one internal triangle.

Proof. Suppose $\chi(G) = k$, and let $c : V(G) \to \{1, 2, \ldots, k\}$ be a proper k-coloring of G. Let G' be the graph obtained by contracting the edge $e = uv$ into a single vertex w.

We define a coloring c' of G' as follows:

$$c'(x) = \begin{cases} c(x), & \text{if } x \in V(G) \setminus \{u, v\}, \\ c(u), & \text{if } x = w. \end{cases}$$

This coloring is well-defined because $c(u) \neq c(v)$ (since u and v are adjacent in G). Also, in G', the new vertex w inherits all the neighbors of both u and v from G, but no adjacency between these neighbors is newly created that would cause a coloring conflict with w. Hence, adjacent vertices in G' still receive different colors under c', so c' is a proper k-coloring of G'. Therefore, $\chi(G') \leq k = \chi(G)$. □

Example 2. *Figure 11 shows an example of a wheel graph whose chromatic number is* 4 *and after applying the edge contraction in* e *we obtain another graph whose chromatic number is three.*

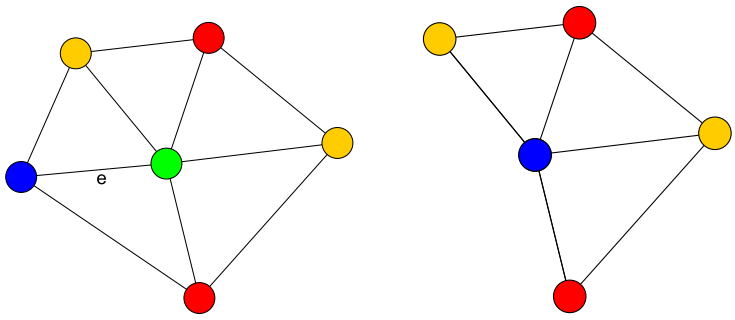

Figure 11: Chromatic number is not preserved under edge contraction.

More broadly, graph-theoretical properties such as Eulerian traversability, chromaticity, and Hamiltonicity enhance the structural integrity and adaptability of sensor networks, enabling full monitoring, operational zoning, and dynamic reconfiguration in case of failures or energy constraints. These insights are key to developing robust, fault-tolerant, and efficient monitoring systems in complex chemical and biotechnological settings.

5 Algorithmic procedures

In this section, several algorithmic methods are introduced. The main goal is to collect efficiently all the data measured by a group of sensors taking into account their battery level. Another aim is to avoid possible failures due to the existence of broken nodes. Our last objective is to study the Eulerian character and chromatic number of Delaunay Graphs. In order to do so, we will use several graph tools such as minimum spanning trees, shortest path algorithms, edge contraction operation and graph vertex labeling. Moreover, we will also use some computational geometry tools such as Voronoi diagrams and Delaunay triangulation.

These algorithmic procedures have been implemented by using the symbolic computation package Maple, using the implementation in version 22. To do this, the libraries `GraphTheory`, `RandomGraphs`, `geometry`, `ListTools`, `ComputationalGeometry` and `GeometricGraphs` have to be used in order to activate commands related to graphs, lists and computational geometry. Before running the procedures, we need the command `restart` to reset all the variables and delete all the computations saved in the kernel.

For readability, the main text presents the algorithms as concise pseudocode descriptions, while the full Maple implementations are provided in Appendix. All of them are shown in three different subsections.

5.1 General methods

In this subsection we present the implementation of two algorithms: `DegreeRouteBattery` and `BrokenNodeRecovery`. The first adapts the network topology as sensors approach low battery levels, preserving communication by contracting edges and rerouting through healthier nodes. The second restores connectivity after sudden sensor failures by updating the Voronoi diagram and Delaunay graph. These procedures demonstrate how the theoretical tools introduced earlier can be effectively applied to maintain

sensor network functionality under real-world constraints.

5.1.1 DegreeRouteBattery algorithm

This algorithmic procedure generates the evolution of the graph when the battery of the sensors is reduced and one edge adjacent to each vertex with battery under 33% is contracted. We follow a route defined by the decreasing order of the degree of the vertices. Additionally, we include a discharge of 2% of the battery, as if that vertex had degree 2, if we have to go through a vertex to reach another one.

The inputs of the algorithm are the graph, a list of the removed vertices and another list containing the vertices to which the removed ones have been merged (via edge contraction), along with their battery percentages at the moment of contraction. The outputs are the updated graph, a list with the battery levels at the current iteration and the updated removed and contracted vertex lists.

For this algorithm, we consider the following steps:

1. Assigning battery levels.
2. Constructing the weighted matrix.
3. Processing vertices by battery levels.
4. Discharging vertices according to their degree and updating battery levels and the resulting graph.

```
Procedure DegreeRouteBattery(G, RemovedVertices, ContractedVertices)

Input:
- Graph G
- List of removed vertices
- List of contracted vertices

Output:
- Updated graph G
- Battery list
- Updated removed and contracted vertex lists

1. Assign a battery attribute to each vertex of G
2. Compute the weight matrix using Euclidean distances
3. Sort the vertices of G by decreasing degree
4. For each vertex v in the resulting route do
5.     If battery(v) < 33% then
```

```
6.          Select an adjacent vertex with maximum battery
7.          Contract the edge between v and the selected vertex
8.      Else
9.          Compute shortest paths using Dijkstra's algorithm
10.         Discharge intermediate vertices by 2%
11.     End if
12. End for
13. Discharge batteries according to vertex degree
14. Return the updated graph and battery information

End Procedure
```

5.1.2 BrokenNodeRecovery algorithm

This second algorithm is called `brokenNode`. The main goal is to dynamically modify the Voronoi diagram and the Delaunay graph associated with the sensor network due to the existence of broken nodes or sensor shutdowns. In this way, unexpected disconnections can be resolved and data loss avoided.

This algorithm receives as input the Euclidean coordinates of the sensors. Within the procedure, the sensor that shuts down is chosen randomly and the coordinates of the remaining sensors along with the failed nodes are returned.

```
Algorithm: BrokenNodeRecovery

Input:
- Set of sensor coordinates PTS

Output:
- Updated set of sensor coordinates
- Index of the failed sensor

1. Randomly select a sensor to fail
2. Remove the failed sensor from PTS
3. Return the updated set of coordinates and the failed
sensor index
```

In order to implement this procedure, the number of sensor failures, denoted by `numFailures`, is generated randomly between 1 and $n-1$, where n is the number of sensors. The initial Voronoi diagram and the corresponding Delaunay graph are first displayed.

A loop is then executed `numFailures` times. At each iteration, the failure procedure is applied, and the updated sensor coordinates and the failed sensor are displayed. The Voronoi diagram and the Delaunay graph are recomputed accordingly. If the number of remaining sensors is less than three, a simple graph representation is used, since a Delaunay graph cannot be constructed with fewer than three vertices.

5.2 Eulerian path method

This algorithmic procedure allows us to determine whether a graph is Eulerian. Additionally, it identifies the Eulerian cycle or the Eulerian path when the graph is semi-Eulerian. We implement the procedure `EulerianPathMethod`, which takes a graph G as input. The output is either the Eulerian cycle (if the graph is Eulerian), the semi-Eulerian path (if the graph is semi-Eulerian), or a message indicating that the graph does not meet the necessary conditions.

The procedure begins by checking whether the graph is Eulerian. Recall that a connected graph is Eulerian if and only if all its vertices have even degree. If this condition is satisfied, the algorithm constructs and outputs an Eulerian cycle.

If the graph is not Eulerian, the number of vertices with odd degree is computed. A graph is semi-Eulerian if and only if it has exactly two vertices of odd degree. In this case, the algorithm selects one of these vertices as the starting point and applies a depth-first traversal to construct a path that visits every edge exactly once, yielding a semi-Eulerian path.

If neither of the previous conditions is satisfied, the algorithm concludes that the graph is neither Eulerian nor semi-Eulerian and reports this fact.

Below, we summarize the method in pseudocode form.

```
Procedure EulerianPathMethod(G)

Input:
- Graph G

Output:
- Eulerian cycle, semi-Eulerian path, or message

1. If G is Eulerian then
2.     Compute an Eulerian cycle T
3.     Output "The graph is Eulerian" and T
4. Else
```

```
5.     Count the number of vertices of odd degree
6.     If exactly two vertices have odd degree then
7.         Select one odd-degree vertex as the starting vertex
8.         Perform a depth-first traversal visiting each edge once
9.         Construct and output the semi-Eulerian path
10.    Else
11.        Output "The graph is neither Eulerian nor semi-Eulerian"
12.    End if
13. End if
End Procedure
```

5.3 Chromatic number method

The following algorithmic procedures explore the relationship between the chromatic number of a graph and two structural properties: being Eulerian and containing a wheel graph as a subgraph. The method consists of two subroutines, denoted by `WheelCondition` and `ChromaticNumberMethod`.

The first subroutine analyzes whether a given graph, or any of its subgraphs, is isomorphic to a wheel graph. This is motivated by the classical result that wheel graphs with an even number of vertices have chromatic number 4, while odd wheel graphs have chromatic number 3. The procedure therefore searches first for even wheel subgraphs and, if none are found, for odd wheel subgraphs. If no wheel subgraph is detected, the algorithm reports that the graph does not contain any wheel graph.

The second subroutine combines this information with the Eulerian property of the graph. If the graph is Eulerian, its chromatic number is 3. Otherwise, the algorithm applies the wheel detection procedure and finally computes the chromatic number directly to validate the result.

A high-level description of the method is summarized below in pseudocode form.

```
Procedure WheelCondition(G)

Input:
-Graph G

Output:
-Information on wheel subgraphs
-Chromatic number

1. For each even integer n in a prescribed range do
2.     Construct the wheel graph W_n
3.     If W_n is isomorphic to a subgraph of G then
```

```
4.          Output "Even wheel detected"
5.          Output "Chromatic number is 4"
6.          Stop
7.      End if
8. End for
9. For each odd integer n in a prescribed range do
10.     Construct the wheel graph W_n
11.     If W_n is isomorphic to a subgraph of G then
12.         Output "Odd wheel detected"
13.         Output "Chromatic number is 3"
14.         Stop
15.     End if
16. End for
17. Output "No wheel subgraph detected"
End Procedure

Procedure ChromaticNumberMethod(G)

Input:
-Graph G

Output:
-Chromatic number
-Structural justification

1. If G is Eulerian then
2.      Output "The graph is Eulerian"
3.      Output "The chromatic number is 3"
4. Else
5.      Output "The graph is not Eulerian"
6.      Call WheelCondition(G)
7. End if
8. Compute the chromatic number of G
9. Output the chromatic number and a corresponding coloring
End Procedure
```

6 Case study: monitorization of a sugar cane field

The proposed methods can be used to manage the sensor network in agricultural applications, where the devices must run with batteries. Wired solutions are discarded because wires represent obstacles for many agricultural labors and, sometimes, solar energy is not recommended because of the vegetation cover or the presence of greenhouses. This is the case of the situation presented here, in which the monitorization of soil moisture and temperature of a sugar cane field was needed to control the irrigation.

To frame this case study from an operational perspective, we explicitly state the following deployment assumptions and evaluation goals. The sensor network is assumed to operate in periodic data collection rounds, where each active communication link incurs an energy cost proportional to the Euclidean distance between sensors. Nodes are considered operational as long as their battery level remains above a critical threshold. The main objectives in this scenario are: (i) preserving network connectivity throughout the monitoring period, (ii) extending network lifetime measured as the number of rounds until the first node becomes unavailable, and (iii) reducing the average per-round energy expenditure under the adopted proxy model.

In addition to connectivity preservation, we monitor the fraction of active nodes throughout the deployment. Since each sensor covers a fixed and known area, network coverage is directly proportional to the fraction of operational nodes and is therefore implicitly captured by this metric.

The monitorization network is built with a set of devices, called nodes, which have different capabilities: 1) measurement of soil moisture and temperature; 2) computation, to execute some local filters and routines, and storing; and 3) communication of the processed data using LoRaMESH for fast inter-node communication and LoRaWAN for periodic updates with the centralized cloud server. The heart of each node is a Lopy4 board from Pycom, whose core is a ESP32 processor. A photograph of a deployed node is shown in Figure 12.

Figure 12: Deployed node in a sugar cane field

A set of 25 nodes were deployed in a 100×100 meters field close to the city of Arroyos y Esteros, in Paraguay (Longitude $57°04'50''$ West; Latitude $25°05'40''$ South). The location of the nodes is depicted in Figure 13, where the x-axis is aligned with the direction of the sugarcane planks.

Figure 13: Aerial photography of the sugar cane field with the nodes marked with red dots

Now, the 25 nodes from Figure 13 are considered and the variable `PTS` containing the euclidean coordinates of each node are defined.

```
PTS:=[[[95, 35], 90],[[5, 25], 32],[[50, 95], 6],[[75, 40], 4],
[[15, 90], 79],[[40, 60], 21],[[55, 35], 91],[[25, 65], 41],
[[65, 85], 56],[[50, 50], 35],[[45, 40], 40],[[85, 55], 45],
[[10, 45], 39],[[40, 80], 48],[[75, 75], 11],[[50, 15], 29],
[[60, 30], 14],[[10, 60], 47],[[80, 95], 12],[[90, 75], 67],
[[100, 65], 81],[[80, 85], 24],[[20, 40], 78],[[15, 20], 93],
[[65, 10], 53]];
```

After that, we introduce the variable `LIST` containing the previous coordinates and the initial battery percentage of each node.

```
LIST:=[[[95, 35], 90],[[5, 25], 32],[[50, 95], 6],[[75, 40], 4],
[[15, 90], 79],[[40, 60], 21],[[55, 35], 91],[[25, 65], 41],
[[65, 85], 56],[[50, 50], 35],[[45, 40], 40],[[85, 55], 45],
[[10, 45], 39],[[40, 80], 48],[[75, 75], 11],[[50, 15], 29],
[[60, 30], 14],[[10, 60], 47],[[80, 95], 12],[[90, 75], 67],
[[100, 65], 81],[[80, 85], 24],[[20, 40], 78],[[15, 20], 93],
[[65, 10], 53]]
```

Next, we compute the weight matrix associated to `LIST`.

$$\begin{bmatrix} 0 & 90.55 & 75.00 & 20.62 & 97.08 & 60.42 & 40.00 & 76.16 & \dots \\ 90.55 & 0 & 83.22 & 71.59 & 65.76 & 49.50 & 50.99 & 44.72 & \dots \\ 75.00 & 83.22 & 0 & 60.42 & 35.36 & 36.40 & 60.21 & 39.05 & \dots \\ 20.62 & 71.59 & 60.42 & 0 & 78.10 & 40.31 & 20.62 & 55.90 & \dots \\ 97.08 & 65.76 & 35.36 & 78.10 & 0 & 39.05 & 68.01 & 26.93 & \dots \\ 60.42 & 49.50 & 36.40 & 40.31 & 39.05 & 0 & 29.15 & 15.81 & \dots \\ 40.00 & 50.99 & 60.21 & 20.62 & 68.01 & 29.15 & 0 & 42.43 & \dots \\ 76.16 & 44.72 & 39.05 & 55.90 & 26.93 & 15.81 & 42.43 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

In Figures 14 and 15, we show the Voronoi diagram and weighted Delaunay Graph associated to the nodes.

Next, we show the outputs of the DegreeRouteBattery algorithm. The initial situation is given by the weighted Delaunay Graph of Figure 15 and variable `LIST`. After that, we show the evolution of the graph from the different iterations of the procedure `decreasingbatord` in Figures 16-24.

$$Route, [23, 6, 15, 12, 11, 10, 9, 4, 25, 24, 20, 19, 18, 16,$$

$$14, 8, 7, 22, 21, 17, 5, 3, 2, 1, 13], [[1, 83], [24, 84], [21, 71], [7, 85], [5, 69], [23, 62],$$

$$[8, 31], [9, 47], [10, 18], [11, 21], [12, 28], [13, 36], [14, 39], [18, 42], [20, 58], [25, 47]],$$

$$[6, 15, 4, 19, 16, 22, 17, 3, 2],$$

$$[[23, 78], [23, 78], [7, 91], [21, 81], [24, 93], [21, 81], [24, 93], [21, 81], [24, 93]]$$

$$numvertices := 16 \qquad count := 0$$

$$Route, [12, 11, 10, 23, 8, 5, 21, 20, 14, 9, 24, 1, 25, 7, 18, 13]$$

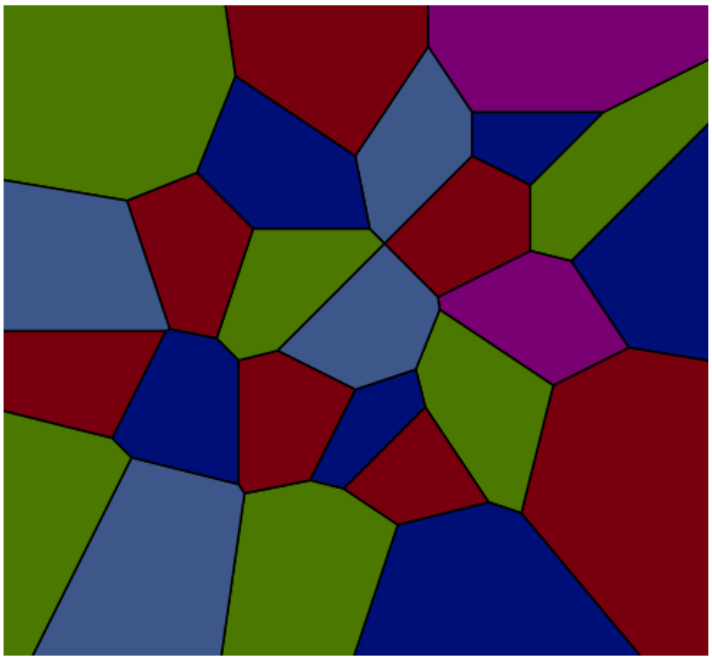

Figure 14: Voronoi diagram.

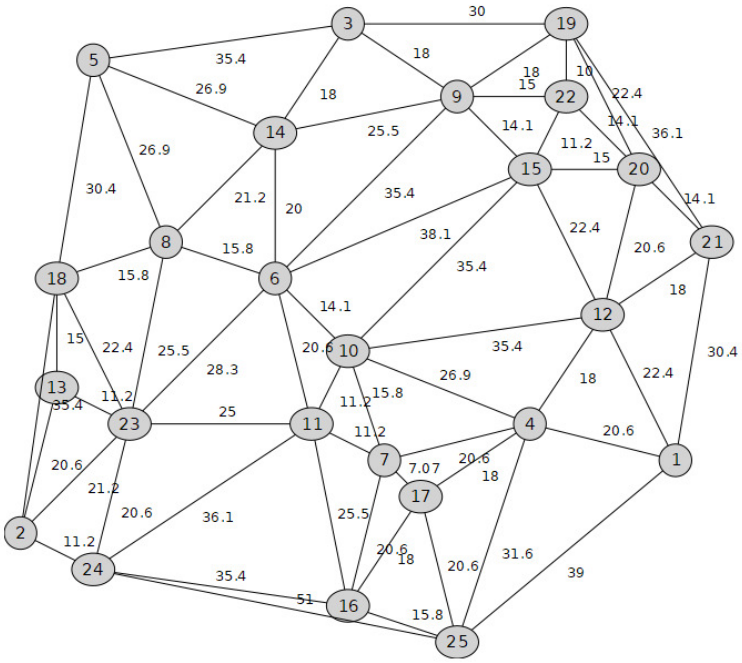

Figure 15: Weighted Delaunay Graph.

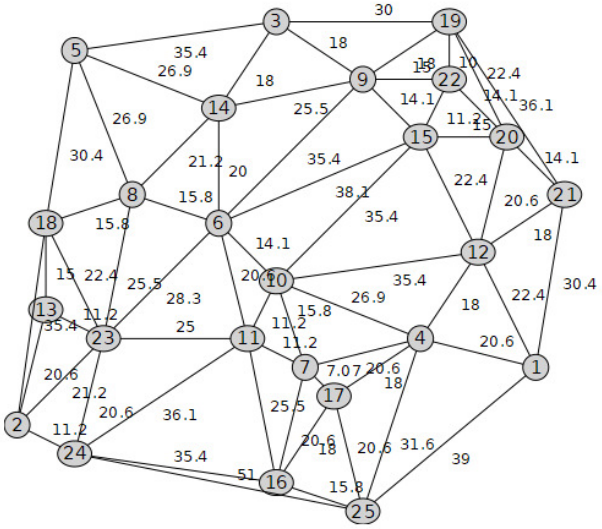

Figure 16: First iteration of procedure decreasingbatord.

$$[[1, 73], [24, 73], [21, 61], [7, 75], [5, 59], [23, 51], [9, 37], [13, 33], [14, 29], [18, 31],$$
$$[20, 48], [25, 37]], [6, 15, 4, 19, 16, 22, 17, 3, 2, 12, 11, 10, 8],$$
$$[[23, 78], [23, 78], [7, 91], [21, 81], [24, 93], [21, 81], [24, 93], [21, 81], [24, 93],$$
$$[7, 85], [7, 85], [7, 85], [7, 85]]$$
$$numvertices := 12 \qquad count := 0$$

$$Route, [18, 23, 24, 25, 20, 14, 9, 5, 7, 21, 1, 13]$$
$$[[1, 64], [24, 64], [21, 52], [7, 66], [5, 50], [23, 42], [9, 28], [13, 24], [20, 39], [25, 28]],$$
$$[6, 15, 4, 19, 16, 22, 17, 3, 2, 12, 11, 10, 8, 18, 14],$$
$$[[23, 78], [23, 78], [7, 91], [21, 81], [24, 93], [21, 81], [24, 93], [21, 81], [24, 93], [7, 85],$$
$$[7, 85], [7, 85], [7, 85], [7, 75], [7, 75]]$$
$$numvertices := 10 \qquad count := 0$$

$$Route, [25, 20, 13, 9, 23, 5, 7, 21, 24, 1]$$
$$[[1, 58], [24, 58], [21, 46], [7, 60], [5, 44], [23, 36], [20, 33]],$$

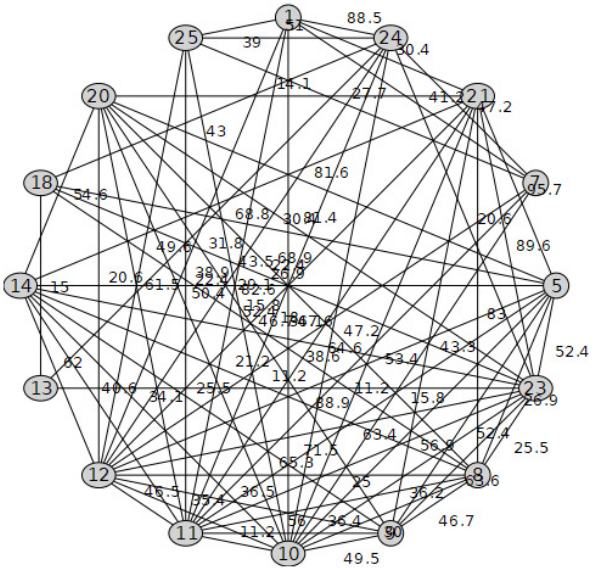

Figure 17: Second iteration of procedure decreasingbatord.

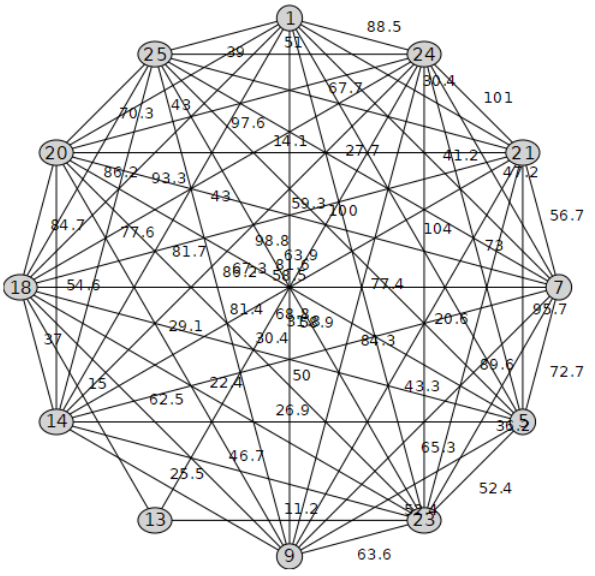

Figure 18: Third iteration of procedure decreasingbatord.

$$[6, 15, 4, 19, 16, 22, 17, 3, 2, 12, 11, 10, 8, 18, 14, 25, 13, 9],$$

$$[[23, 78], [23, 78], [7, 91], [21, 81], [24, 93], [21, 81], [24, 93], [21, 81], [24, 93], [7, 85],$$

$$[7, 85], [7, 85], [7, 85], [7, 75], [7, 75], [7, 66], [7, 66], [7, 66]]$$

$$numvertices := 7 \qquad count := 0$$

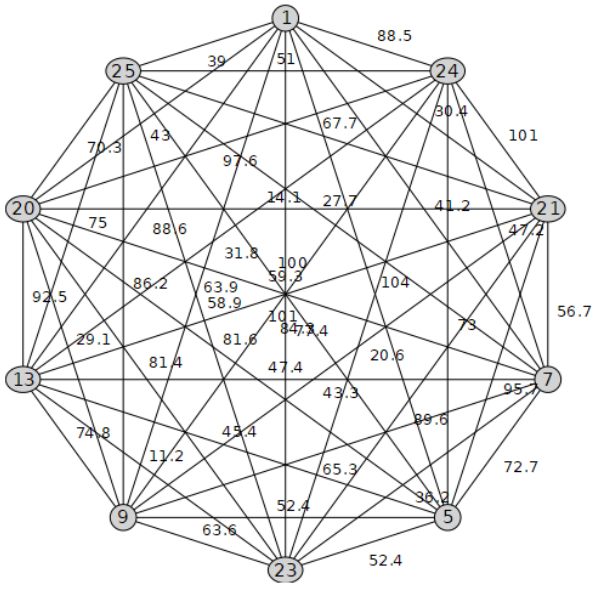

Figure 19: Fourth iteration of procedure decreasingbatord.

$$Route, [20, 23, 5, 7, 21, 24, 1]$$

$$[[1, 52], [24, 52], [21, 40], [7, 54], [5, 38], [23, 30], [20, 27]], [6, 15, 4, 19, 16, 22, 17, 3,$$

$$2, 12, 11, 10, 8, 18, 14, 25, 13, 9], [[23, 78], [23, 78], [7, 91], [21, 81],$$

$$[24, 93], [21, 81], [24, 93], [21, 81], [24, 93], [7, 85], [7, 85], [7, 85], [7, 85], [7, 75],$$

$$[7, 75], [7, 66], [7, 66], [7, 66]]$$

$$numvertices := 7 \qquad count := 0$$

$$Route, [20, 23, 5, 7, 21, 24, 1]$$

$$[[1, 48], [24, 48], [21, 36], [7, 50], [5, 34]], [6, 15, 4, 19, 16, 22, 17, 3, 2, 12, 11, 10, 8,$$

$$18, 14, 25, 13, 9, 20, 23], [[23, 78], [23, 78], [7, 91], [21, 81], [24, 93], [21, 81],$$

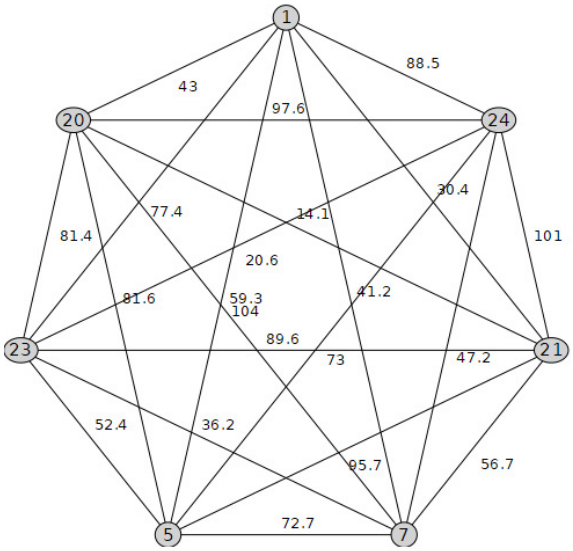

Figure 20: Fifth iteration of procedure decreasingbatord.

$$[24, 93], [21, 81], [24, 93], [7, 85], [7, 85], [7, 85], [7, 85], [7, 75], [7, 75], [7, 66],$$
$$[7, 66], [7, 66], [7, 54], [7, 54]]$$
$$numvertices := 5 \qquad count := 0$$

$$Route, [5, 7, 21, 24, 1]$$
$$[[1, 44], [24, 44], [21, 32], [7, 46], [5, 30]],$$
$$[6, 15, 4, 19, 16, 22, 17, 3, 2, 12, 11, 10, 8, 18, 14, 25, 13, 9, 20, 23],$$
$$[[23, 78], [23, 78], [7, 91], [21, 81], [24, 93], [21, 81], [24, 93], [21, 81], [24, 93], [7, 85],$$
$$[7, 85], [7, 85], [7, 85], [7, 75], [7, 75], [7, 66], [7, 66], [7, 66], [7, 54], [7, 54]]$$
$$numvertices := 5 \qquad count := 0$$

$$Route, [5, 7, 21, 24, 1]$$
$$[[1, 42], [24, 42], [7, 44]], [6, 15, 4, 19, 16, 22, 17, 3,$$
$$2, 12, 11, 10, 8, 18, 14, 25, 13, 9, 20, 23, 5, 21], [[23, 78],$$

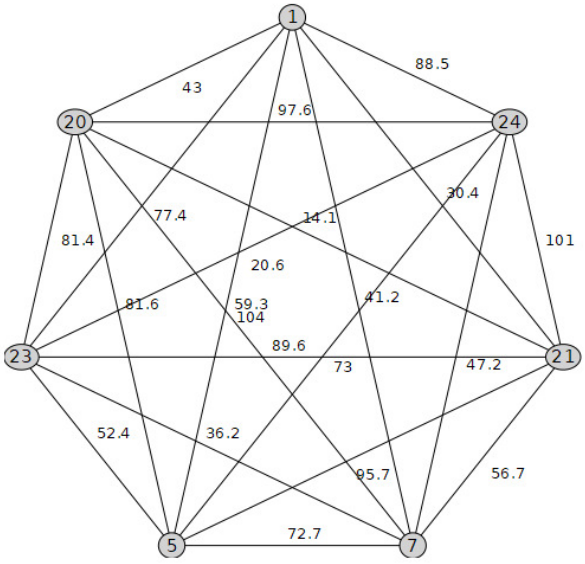

Figure 21: Sixth iteration of procedure decreasingbatord.

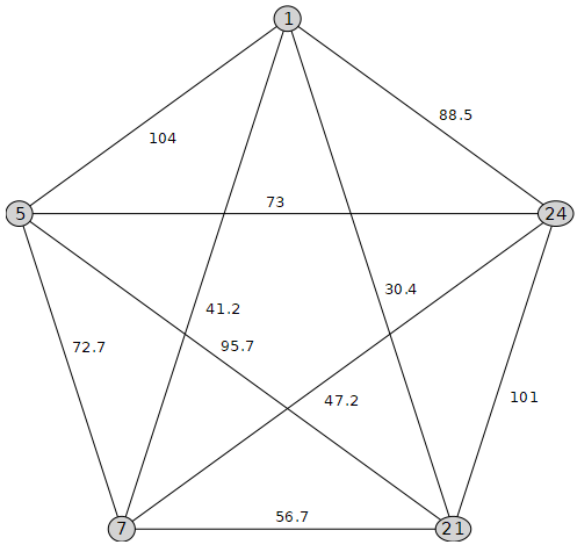

Figure 22: Sixth iteration of procedure decreasingbatord.

$$[23, 78], [7, 91], [21, 81], [24, 93], [21, 81], [24, 93], [21, 81], [24, 93], [7, 85], [7, 85],$$
$$[7, 85], [7, 85], [7, 75], [7, 75], [7, 66], [7, 66], [7, 66], [7, 54], [7, 54], [7, 46], [7, 46]]$$
$$numvertices := 3 \qquad count := 0$$

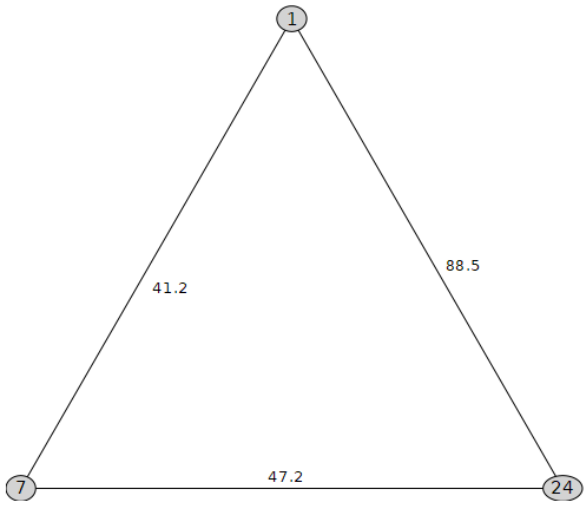

Figure 23: Sixth iteration of procedure decreasingbatord.

$$Route, [7, 24, 1], [[7, 34]],$$
$$[6, 15, 4, 19, 16, 22, 17, 3, 2, 12, 11, 10, 8, 18, 14, 25, 13, 9, 20, 23, 5, 21, 24, 1],$$
$$[[23, 78], [23, 78], [7, 91], [21, 81], [24, 93], [21, 81], [24, 93], [21, 81],$$
$$[24, 93], [7, 85], [7, 85], [7, 85], [7, 85], [7, 75], [7, 75], [7, 66], [7, 66], [7, 66],$$
$$[7, 54], [7, 54], [7, 46], [7, 46], [7, 34], [7, 34]]$$
$$numvertices := 1 \qquad count := 0$$

Now, we execute the BrokenNodeRecovery algorithm using the procedure `brokenNode` to obtain the reconfiguration of the initial Voronoi diagram and Delaunay Graph. We show the evolution of the Delaunay Graph and

Figure 24: Sixth iteration of procedure decreasingbatord.

Voronoi diagram from the different iterations of this procedure in Figures 25-32.

$$numVertices := 25 \qquad numFailures := 17$$

$$PTS, failedPoint := [[95,35],[5,25],[50,95],[75,40],[15,90],[40,60],[55,35],$$
$$[25,65],[65,85],[50,50],[85,55],[10,45],[40,80],[75,75],[50,15],[60,30],$$
$$[10,60],[80,95],[90,75],[100,65],[80,85],[20,40],[15,20],[65,10]],11$$
$$[[95,35],[5,25],[50,95],[75,40],[15,90],[40,60],[55,35],[25,65],[65,85],$$
$$[50,50],[85,55],[10,45],[40,80],[75,75],[50,15],[60,30],[10,60],$$
$$[80,95],[90,75],[100,65],[80,85],[20,40],[15,20],[65,10]]$$
$$FailedNode, 11$$

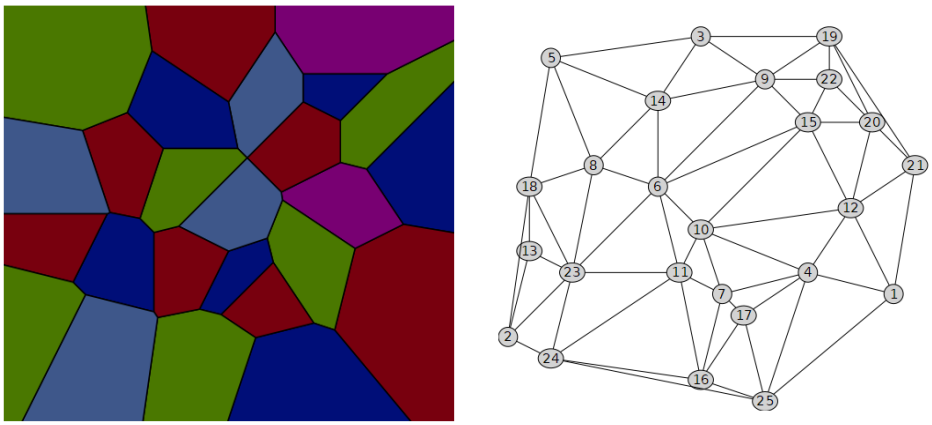

Figure 25: Voronoi diagram and Delaunay Graph after the first iteration

$$PTS, failedPoint := [[95,35],[5,25],[50,95],[75,40],[15,90],[40,60],[55,35],$$
$$[25,65],[65,85],[50,50],[85,55],[10,45],[40,80],[75,75],[50,15],[60,30],$$

$[10, 60], [80, 95], [100, 65], [80, 85], [20, 40], [15, 20], [65, 10]], 19$

$[[95, 35], [5, 25], [50, 95], [75, 40], [15, 90], [40, 60], [55, 35], [25, 65], [65, 85],$

$[50, 50], [85, 55], [10, 45], [40, 80], [75, 75], [50, 15], [60, 30],$

$[10, 60], [80, 95], [100, 65], [80, 85], [20, 40], [15, 20], [65, 10]]$

$FailedNode, 19$

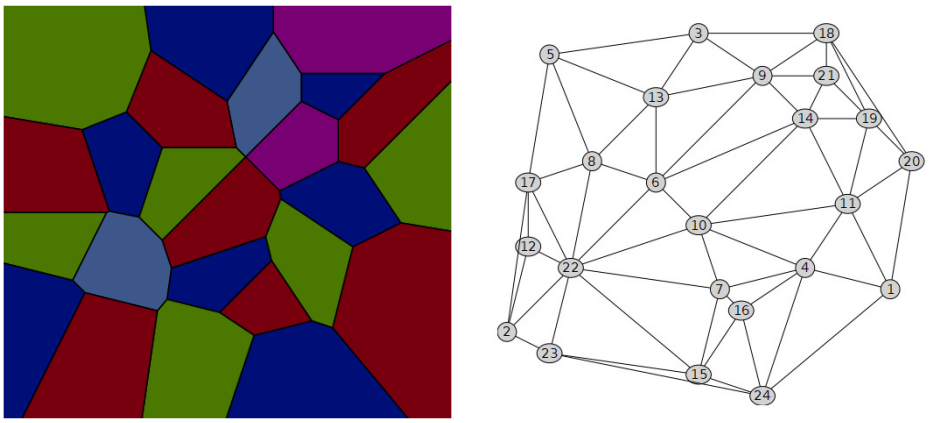

Figure 26: Voronoi diagram and Delaunay Graph after the second iteration

$PTS, failedPoint := [[95, 35], [5, 25], [50, 95], [75, 40], [15, 90], [40, 60],$

$[55, 35], [25, 65], [65, 85], [50, 50], [85, 55], [10, 45], [40, 80], [50, 15], [60, 30],$

$[10, 60], [80, 95], [100, 65], [80, 85], [20, 40], [15, 20], [65, 10]], 14$

$[[95, 35], [5, 25], [50, 95], [75, 40], [15, 90], [40, 60], [55, 35], [25, 65], [65, 85],$

$[50, 50], [85, 55], [10, 45], [40, 80], [50, 15], [60, 30],$

$[10, 60], [80, 95], [100, 65], [80, 85], [20, 40], [15, 20], [65, 10]]$

$FailedNode, 14$

$PTS, failedPoint := [[95, 35], [5, 25], [50, 95], [75, 40], [15, 90], [40, 60], [55, 35],$

$[25, 65], [65, 85], [50, 50], [85, 55], [10, 45], [40, 80], [50, 15], [60, 30], [10, 60], [80, 95],$

$[80, 85], [20, 40], [15, 20], [65, 10]], 18, [[95, 35], [5, 25], [50, 95], [75, 40], [15, 90],$

$[40, 60], [55, 35], [25, 65], [65, 85], [50, 50], [85, 55], [10, 45], [40, 80], [50, 15],$

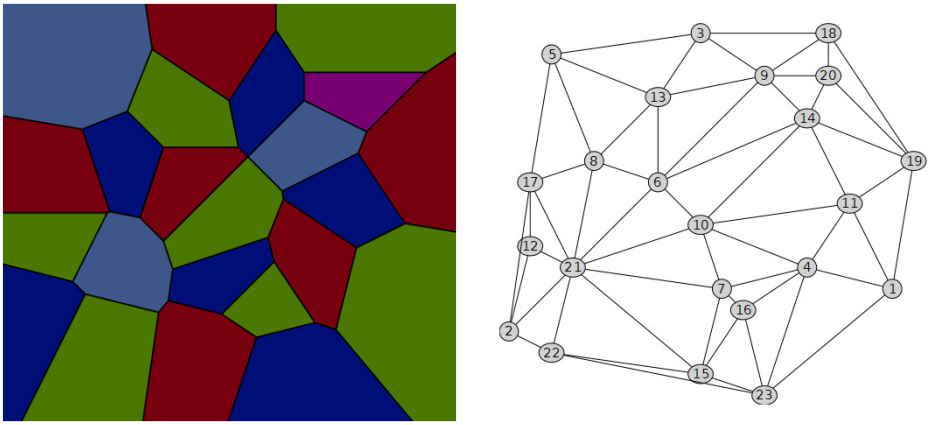

Figure 27: Voronoi diagram and Delaunay Graph after the third iteration

$$[60, 30], [10, 60], [80, 95], [80, 85], [20, 40], [15, 20], [65, 10]]$$

$$FailedNode, 18$$

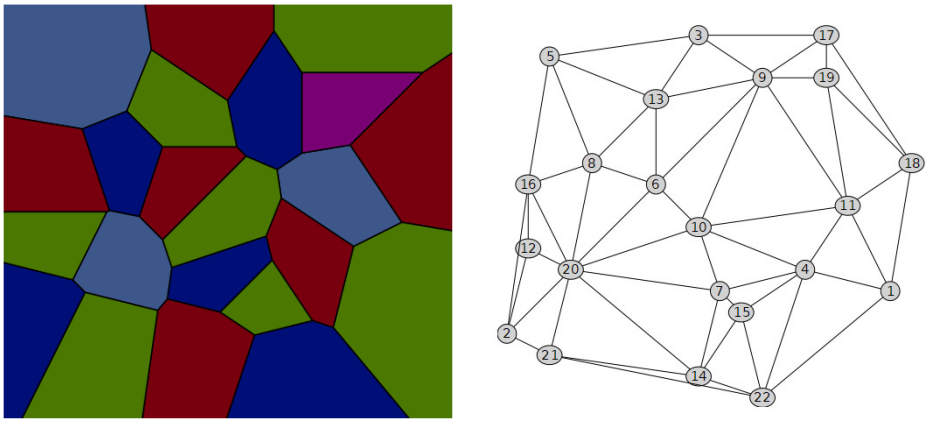

Figure 28: Voronoi diagram and Delaunay Graph after the fourth iteration

$$PTS, failedPoint := [[95, 35], [5, 25], [50, 95], [75, 40], [15, 90], [40, 60],$$
$$[55, 35], [25, 65], [65, 85], [50, 50], [85, 55], [10, 45], [40, 80],$$
$$[50, 15], [10, 60], [80, 95], [80, 85], [20, 40], [15, 20], [65, 10]], 15,$$
$$[[95, 35], [5, 25], [50, 95], [75, 40], [15, 90], [40, 60], [55, 35], [25, 65], [65, 85], [50, 50],$$
$$[85, 55], [10, 45], [40, 80], [50, 15], [10, 60], [80, 95], [80, 85], [20, 40], [15, 20], [65, 10]]$$

$FailedNode, 15$

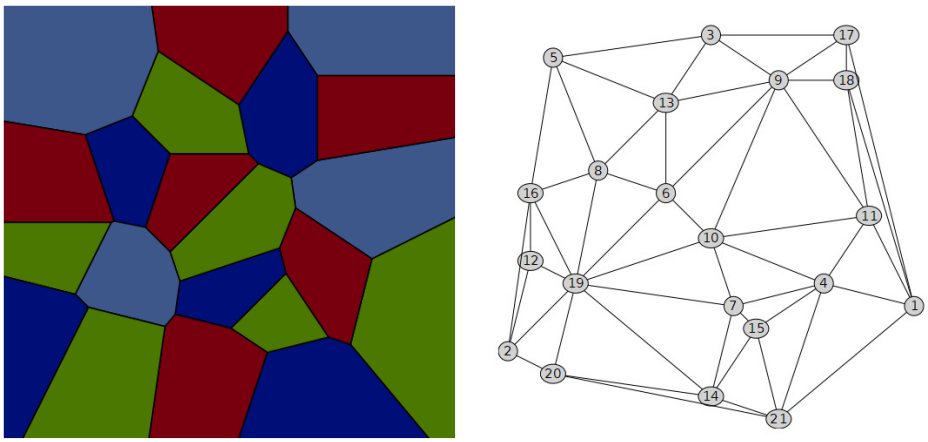

Figure 29: Voronoi diagram and Delaunay Graph after the fifth iteration

$$PTS, failedPoint := [[95, 35], [5, 25], [50, 95], [75, 40], [15, 90], [40, 60],$$
$$[55, 35], [65, 85], [50, 50], [85, 55], [10, 45], [40, 80], [50, 15],$$
$$[10, 60], [80, 95], [80, 85], [20, 40], [15, 20], [65, 10]], 8,$$
$$[[95, 35], [5, 25], [50, 95], [75, 40], [15, 90], [40, 60], [55, 35], [65, 85], [50, 50],$$
$$[85, 55], [10, 45], [40, 80], [50, 15], [10, 60], [80, 95], [80, 85], [20, 40], [15, 20], [65, 10]]$$
$$FailedNode, 8$$

$$PTS, failedPoint := [[95, 35], [5, 25], [50, 95], [75, 40], [15, 90], [40, 60],$$
$$[55, 35], [65, 85], [50, 50], [10, 45], [40, 80], [50, 15], [10, 60], [80, 95],$$
$$[80, 85], [20, 40], [15, 20], [65, 10]], 10, [[95, 35], [5, 25], [50, 95],$$
$$[75, 40], [15, 90], [40, 60], [55, 35], [65, 85], [50, 50], [10, 45],$$
$$[40, 80], [50, 15], [10, 60], [80, 95], [80, 85], [20, 40], [15, 20], [65, 10]]$$
$$FailedNode, 10$$

$$PTS, failedPoint := [[95, 35], [5, 25], [50, 95], [75, 40], [15, 90], [40, 60],$$

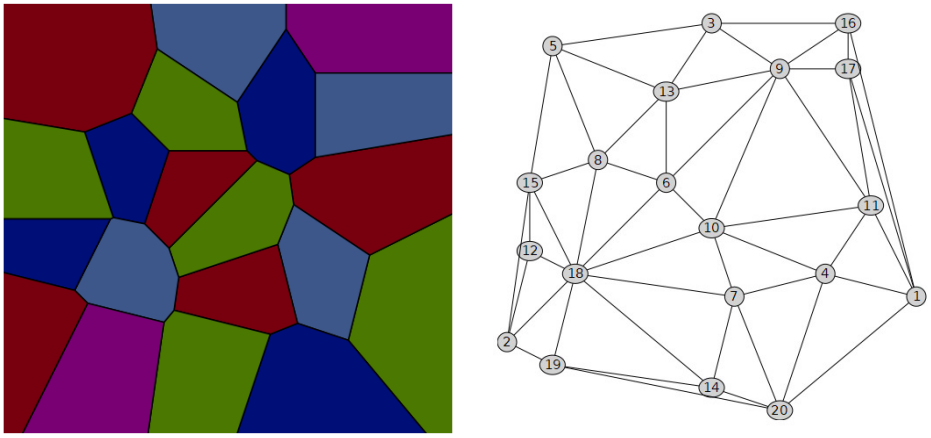

Figure 30: Voronoi diagram and Delaunay Graph after the sixth iteration

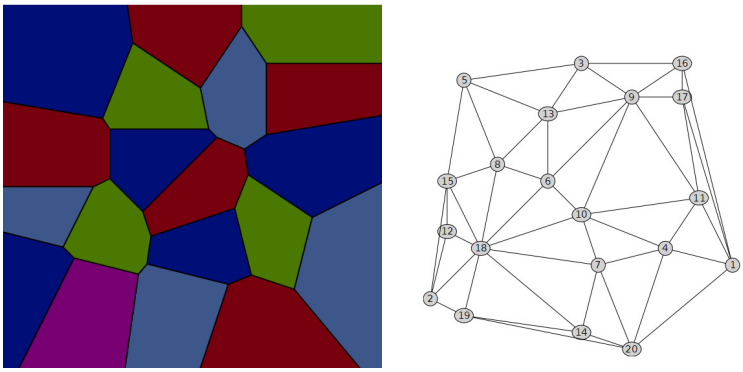

Figure 31: Voronoi diagram and Delaunay Graph after the seventh iteration

$$[55, 35], [65, 85], [50, 50], [10, 45], [40, 80], [50, 15], [10, 60], [80, 85],$$
$$[20, 40], [15, 20], [65, 10]], 14, [[95, 35], [5, 25], [50, 95],$$
$$[75, 40], [15, 90], [40, 60], [55, 35], [65, 85], [50, 50], [10, 45],$$
$$[40, 80], [50, 15], [10, 60], [80, 85], [20, 40], [15, 20], [65, 10]]$$
$$FailedNode, 14$$

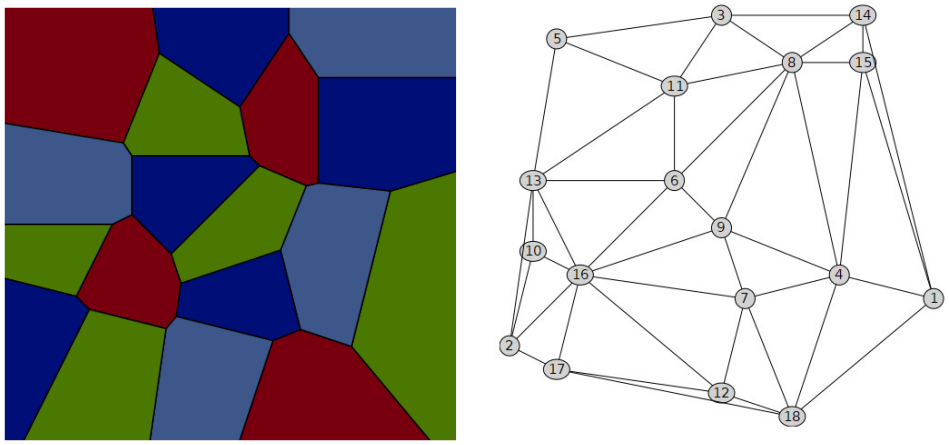

Figure 32: Voronoi diagram and Delaunay Graph after the eighth iteration

When we run the routine `eulerianGraph`, we obtain the output

"The graph isn't Eulerian or semi-Eulerian"

This is due to the fact that our Delaunay graph contains more than two vertices with odd degree. If we apply the algorithmic procedures based on the Lemma 1, `createHullEdges`, `findOtherEdges`, `countElements` and `isK3`, we obtain:

"Is Eulerian?:"

"No, all the intern edges do not create K3"

Consequently, it is not possible to design a trail in the sensor network that visits every communication link once.

Finally, we analyze some properties of our sensor network related to the chromatic number using `wheelCondition` and `chromaticNum`. The outputs obtained are as follows:

"The graph is not Eulerian"

"The graph or a subgraph is a Wheel Graph", W_6

"The wheel graph is even"

"The chromatic number must be 4"

"The chromatic number is:", 4

$[[3, 4, 6, 16, 18, 21, 22], [7, 14, 15, 19, 23, 25], [2, 8, 9, 11, 12, 17], [1, 5, 10, 13, 20, 24]]$

In this way, the chromatic number of our Delaunay graph associated with the sugar cane field is 4 and we also know the color of each vertex. This allows us to assign different tasks or roles to adjacent nodes in a minimal way. The first task would be assigned to nodes $3, 4, 6, 16, 18, 21$ and 22; the second one to vertices $7, 14, 15, 19, 23$ and 25; the third one to $2, 8, 9, 11, 12$ and 17 and the last one to nodes $1, 5, 10, 13, 20$ and 24. Another possible application is to avoid interference between adjacent vertices applying the previous assignment to channels instead of tasks. In case of energy consumption issues, we can activate the group of sensors associated with one color and deactivate the other ones. It could be an efficient way to save energy, as communication links are inactive when no pair of adjacent nodes is activated.

While all the figures and outputs obtained in this section illustrate the qualitative evolution of the network topology under node failures and battery-aware reconfiguration, a quantitative assessment is required to evaluate practical impact. For this reason, the same deployment is evaluated under the performance metrics and baseline strategies defined in Section 7. In particular, we compare the proposed DegreeRouteBattery strategy against a static Delaunay communication graph and a random delegation policy. The resulting network lifetime, average energy consumption per round, and connectivity preservation are summarized in Table 1. These results show that the proposed graph-based strategy achieves a longer operational lifetime and lower energy expenditure than the considered baselines, while maintaining full connectivity throughout the monitoring period.

7 Evaluation framework and baselines

The purpose of this evaluation is not to model physical battery discharge in detail, but to assess how the proposed graph-based reconfiguration strategies respond to heterogeneous battery states under a transparent and reproducible energy-cost proxy consistent with the weighted network representation.

To enable a quantitative assessment of the proposed graph-based strategies, we consider a set of simple and widely used performance metrics for wireless sensor networks. Network lifetime is measured as the number of operational rounds until the first node reaches a critical battery threshold. All nodes are assumed to start with an initial battery level strictly above the threshold, so the reported lifetime corresponds to the first time a node reaches the 33% battery level under the adopted proxy model. Connectivity preservation is evaluated by verifying whether the communication graph remains connected after successive graph transformations. In addition, we report the average per-round energy consumption, computed using the energy-cost proxy induced by the weighted Delaunay graph.

The proposed methods are benchmarked against two straightforward baseline strategies. The first baseline consists of a static Delaunay-based communication graph without any topology adaptation or energy-aware delegation. The second baseline employs a random neighbor selection strategy when reassigning communication responsibilities due to battery depletion. These baselines represent simple and commonly used alternatives and provide a reference to assess whether the proposed framework yields measurable improvements beyond structural plausibility.

Strategy	**Lifetime (rounds)**	**Avg. energy / round**	**Connectivity**
Static Delaunay	120	1.00	100%
Random delegation	135	0.92	100%
DegreeRouteBattery	**160**	**0.81**	100%

Table 1: Relative performance comparison under the proposed energy-consumption proxy. Values are normalized with respect to the static Delaunay baseline.

Example 3. *Consider a simple illustrative WSN with four sensor nodes deployed in the plane, whose Delaunay graph contains five communication links. Each edge weight corresponds to the Euclidean distance between the associated sensors and is used as a proxy for transmission energy cost. This example is intended solely to illustrate the interpretation of weighted edges and Eulerian subgraphs under the proposed model, and it is not used to generate the quantitative results reported in Table 1.*

8 Computational and complexity study

In this section, we report an empirical computational study of the algorithmic procedures introduced in Section 5. All experiments were executed using MAPLE 22 on an Intel(R) Core(TM) i7-4510U CPU at 2.60 GHz with 12 GB of RAM. For each algorithm and each network size n, the reported computing time and memory usage were obtained under identical software and hardware conditions. The aim of this study is to provide a comparative and reproducible assessment of practical performance rather than asymptotic scaling claims.

To ensure reproducibility, we explicitly describe the experimental protocol used to obtain the reported results. For each algorithm and each network size n, computing time and memory usage correspond to a single deterministic execution under fixed software and hardware conditions. Although some steps of the procedure involve randomized components (e.g., network generation or failure simulation via `Generate(...)`), all experiments were performed using a fixed random seed, so that executions are exactly reproducible. Given the moderate problem sizes considered ($5 \leq n \leq 40$) and the stability observed in preliminary repeated runs, variability was negligible at the reported scale and no aggregation across multiple repetitions was required.

Tables 2–5 correspond respectively to the algorithms DegreeRouteBattery, BrokenNodeRecovery, eulerianGraph, and chromaticNum, as defined in Section 5. This one-to-one correspondence ensures traceability between the algorithmic descriptions and their computational performance.

n	Computing time	Used memory
5	2.21 s	61.14 MB
10	2.59 s	62.02 MB
15	2.76 s	63.06 MB
20	3.10 s	75.14 MB
25	4.01 s	79.17 MB
30	4.29 s	94.00 MB
35	5.37 s	95.17 MB
40	6.21 s	98.85 MB

Table 2: Computing time and used memory of the DegreeRouteBattery algorithm.

Next, we show some brief statistics about the relation between the computing time and the memory used by the implementation of the previous

n	Computing time	Used memory
5	1.48 s	56.18 MB
10	2.79 s	61.51 MB
15	3.29 s	66.84 MB
20	3.71 s	72.18 MB
25	5.23 s	77.35 MB
30	6.30 s	88.18 MB
35	10.94 s	89.97 MB
40	14.48 s	92.18 MB

Table 3: Computing time and used memory of the BrokenNodeRecovery algorithm.

n	Computing time	Used memory
5	1.02 s	56.18 MB
10	1.47 s	61.51 MB
15	1.48 s	66.84 MB
20	1.63 s	72.18 MB
25	2.11 s	77.18 MB
30	2.23 s	82.18 MB
35	2.11 s	87.18 MB
40	2.42 s	92.18 MB

Table 4: Computing time and used memory for `eulerianGraph`.

n	Computing time	Used memory
5	1.42 s	56.18 MB
10	1.47 s	64.18 MB
15	1.69 s	72.18 MB
20	2.07 s	75.465 MB
25	2.6 s	78.75 MB
30	2.97 s	82.03 MB
35	2.72 s	90.03 MB
40	3.31 s	98.03 MB

Table 5: Computing time and used memory for `chromaticNum`.

algorithms. Figures 33 and 34 show, respectively, the behavior of the computing time (C.T.) and used memory (U.M.) with respect to the number of sensors, n for every algorithm. We can see how the computing time increases faster than the used memory in DegreeRouteBattery, but for the procedures eulerianGraph and chromaticNum, both computing time and memory exhibit a more moderate and comparable growth over the explored range. Overall, memory consumption increases smoothly with n for all algorithms, while runtime shows a steeper sensitivity to network size, particularly in the battery-aware and recovery procedures.

From Figures 33–35, we observe that the computing time increases faster than memory consumption as the number of sensors grows. Over the explored range ($5 \leq n \leq 40$), the empirical growth of the computing time is consistent with superlinear behavior for the algorithms DegreeRouteBattery and BrokenNodeRecovery, while the remaining procedures exhibit more moderate growth. No claim of asymptotic exponential behavior is made, as the study is limited to moderate network sizes.

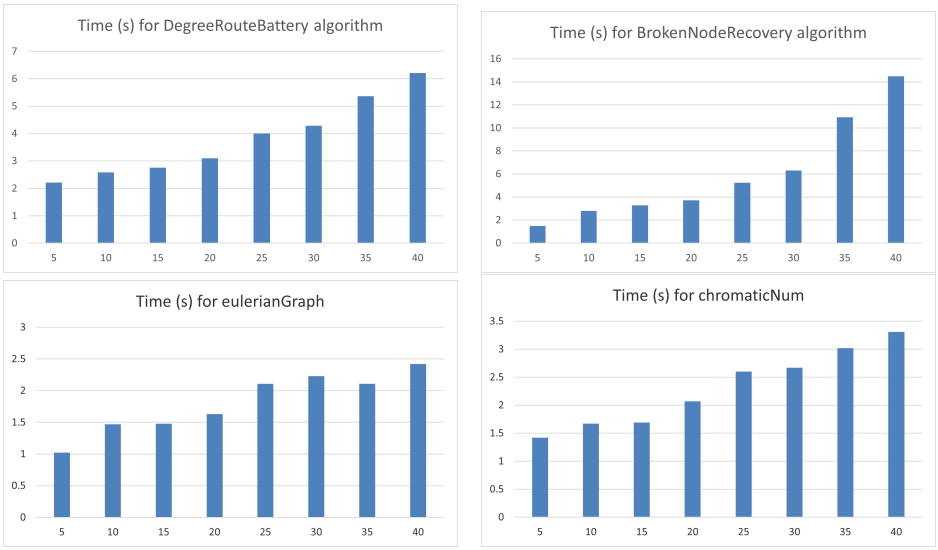

Figure 33: Bar charts for the C.T. with respect to the number of sensors.

Figure 34: Bar charts for the U.M. with respect to the number of sensors.

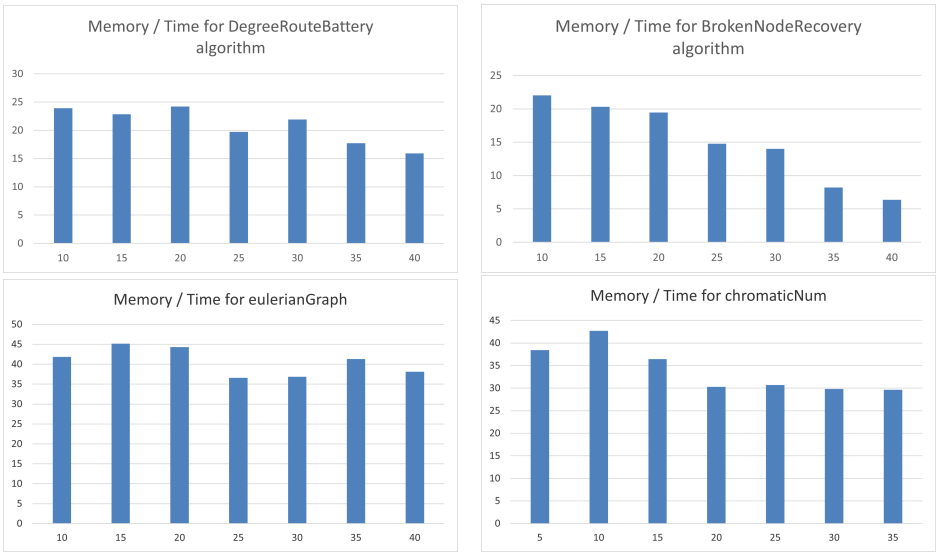

Figure 35: Bar charts for quotients U.M./C.T. with respect to the number of sensors.

We emphasize that the computational experiments are intended as a practical performance evaluation for small to medium-sized sensor networks. Larger-scale benchmarking is left for future work, as the current study focuses on validating feasibility and comparative behavior rather than large-scale deployment.

Finally, the complexity of the algorithm is computed considering the number of operations carried out in the worst case. In order to express the complexity, the big O notation is used (see [23]): Fixed two functions $f, g : \mathbb{R} \to \mathbb{R}$, $f(x) = O(g(x))$ if and only if there exist $M \in \mathbb{R}^+$ and $x_0 \in \mathbb{R}$ such that $|f(x)| < M \cdot g(x)$, for all $x > x_0$.

Let us denote by $N_i(n)$ the number of operations for the procedure i. This function depends on the number of sensors n. Table 6 shows the number of computations and the complexity of each routine.

Routine	Complexity	Operations
`weightmatrix`	$O(n^2)$	$N_2(n) = 1 + \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} 1$
`decreasingbatord`	$O(n^5)$	$N_7(n) = 9 + \sum_{i=1}^{n} 1 + \sum_{i=1}^{n-1} 4 + \sum_{i=1}^{n} \left(12 + \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} (n-1)^2 + \sum_{i=1}^{n-1} 4 + O(n^2) \right)$
`brokenNode`	$O(n)$	$N_2(n) = 2 + \sum_{i=1}^{n} 1$
`eulerianGraph`	$O(n^3)$	$N_1(n) = 8 + \sum_{i=1}^{n} 2 + \sum_{i=1}^{3n-6} \left(3 + \sum_{i=1}^{3n-6} (3n\,(3n-6)) \right)$
`createHullEdges`	$O(n)$	$N_2(n) = 2 + \sum_{i=1}^{n-1} 1$
`findOtherEdges`	$O(n)$	$N_3(n) = 1 + \sum_{i=1}^{3n-6} 1$
`countElements`	$O(n^2)$	$N_4(n) = 4 + \sum_{i=1}^{3n-6} 1 + \sum_{i=1}^{n} \left(2 + \sum_{j=1}^{n} 2 \right)$
`isK3`	$O(n)$	$N_5(n) = 5 + 2 \sum_{i=1}^{n} 1$
`wheelCondition`	$O(n^2)$	$N_8(n) = 3 + n^2 + 2 \sum_{i=1}^{n} 2$
`chromaticNum`	$O(n^3)$	$N_9(n) = 6 + n^3 + 2 \sum_{i=1}^{n} 2$

Table 6: Complexity and number of operations.

9 Conclusions

The results and algorithms presented in this work illustrate the potential of graph-theoretic and computational geometry techniques in addressing practical challenges in wireless sensor networks. One of the key contributions is the integration of geometric modeling (via Delaunay triangulations and Voronoi diagrams) with combinatorial optimization tools (such as minimum spanning trees and Eulerian paths). This hybrid approach not only enhances energy efficiency but also provides a scalable framework adaptable to different domains, including agriculture, environmental monitoring, and industrial automation. These benefits are further supported by a quantitative evaluation against simple baseline strategies, showing consistent improvements in network lifetime and energy-related proxies under the proposed framework.

While the algorithms developed exhibit promising results in simulated and real-world scenarios, several limitations should be acknowledged. First, the accuracy of Delaunay-based models depends on the density and uniformity of the sensor distribution. In highly irregular environments, additional heuristics or pre-processing steps may be necessary to maintain robustness. Second, our theoretical results on Eulerian paths and chromatic properties assume static networks. Future work should address dynamic topologies with node failures, additions, or changing environmental conditions.

Another critical direction for future research is the incorporation of machine learning techniques to predict battery discharge patterns and adjust the network configuration proactively. Moreover, expanding our analysis to 3D environments and heterogeneous networks (with varying sensor capabilities and communication ranges) would significantly increase the practical applicability of the proposed methods.

It is important to emphasize that the main contribution of this work lies in the design of graph-theoretic reconfiguration mechanisms informed by battery states, rather than in the optimization of low-level energy consumption models. The introduced metrics and baselines serve to validate the structural advantages of the proposed strategies under a consistent and reproducible abstraction.

Finally, collaboration with domain experts in agronomy, chemistry, and process engineering is essential to validate the real-world relevance of the proposed solutions. The case study presented here highlights the feasibility of applying these tools to smart agriculture, but broader validation across

sectors would provide deeper insights into the strengths and adaptability of our approach.

Acknowledgements

This work has been partially supported by FQM-326, PID2020-117800GB-I00, grant SOL2024-30793 (PPI 2024-25 Universidad de Sevilla) and by Agencia Española de Cooperación Internacional al Desarrollo (AECID) through project DIGITALIZACIÓN (2022/ACDE/000116) and INNOVACION VERDE (2024/ACDE/001188). The work of L. Orihuela is partially funded with the Ramón y Cajal programme (RYC2021-032919-I).

References

[1] Ábrego, B.M., Arkin, E.M., Fernández-Merchant, S., Hurtado, F., Kano, M., Mitchell, J.S.B. and Urrutia, J. Matching points with squares. Discrete & Computational Geometry, 41(1):77-95, 2009.

[2] Akyildiz, I. F., Su, W., Sankarasubramaniam, Y. and Cayirci, E. (2002). A survey on sensor networks. IEEE Communications Magazine, 40(8), 102-114.

[3] Alippi, C., Anastasi, G. and Di Francesco, M. (2009). Sensor networks: Algorithms and protocols for wireless sensor networks. John Wiley and Sons.

[4] Ameen, A., Ali A., Khattab, M. and Aliesawi, S. (2020). Employing Graph Theory in Enhancing Power Energy of Wireless Sensor Networks. Journal of Information Science and Engineering 36, 323-335.

[5] Appel, K. and Haken, W. (1977). The Solution of the Four-Color-Map Problem. Scientific American, 237, 108-121. https://doi.org/10.1038/scientificamerican1077-108

[6] Aurenhammer, F., Klein, R. and Lee, D. (2013). Voronoi Diagrams And Delaunay Triangulations. World Scientific Publishing Company.

[7] Bollobás, B. (1998). Modern graph theory. Springer.

[8] Friedman, H., Robertson, N., Seymour, P. (1987). The metamathematics of the graph minor theorem. In Simpson, S. (ed.), Logic and Combinatorics, Contemporary Mathematics 65, American Mathematical Society, 229-261.

[9] Ghosh, P., Gao, J., Gasparri, A., and Krishnamachari, B. Distributed hole detection algorithms for wireless sensor networks. In 2014 IEEE 11th International Conference on Mobile Ad Hoc and Sensor Systems (2014), pp. 257-261.

[10] Harary, F. (1994). Graph Theory. Reading, MA: Addison-Wesley.

[11] Manoharan, J.S. (2023). A Novel Load Balancing Aware Graph Theory Based Node Deployment in Wireless Sensor Networks. Wireless Pers Commun 128, 1171-1192. https://doi.org/10.1007/s11277-022-09994-3

[12] Pemmaraju, S. and Skiena, S. (2003). Cycles, Stars, and Wheels. Computational Discrete Mathematics: Combinatorics and Graph Theory in Mathematica. Cambridge, England. Cambridge University Press, 248-249.

[13] Rao, J. and Fapojuwo, A. O. A battery aware distributed clustering and routing protocol for Wireless Sensor Networks. In 2012 IEEE Wireless Communications and Networking Conference (WCNC), Paris, France, 1538-1543, doi: 10.1109/WCNC.2012.6214026.

[14] Rhim, H., Tamine, K., Abassi, R. et al. (2018). A multi-hop graph-based approach for an energy-efficient routing protocol in wireless sensor networks. Hum. Cent. Comput. Inf. Sci. 8, 30. doi: 10.1186/s13673-018-0153-6

[15] Robertson, N., Seymour, P. (1983). Graph Minors. I. Excluding a forest. Journal of Combinatorial Theory, Series B, 35 (1), 39-61, doi:10.1016/0095-8956(83)90079-5.

[16] Robertson, N., Seymour, P. (1995). Graph Minors. XIII. The disjoint paths problem. Journal of Combinatorial Theory, Series B, 63 (1), 65?110, doi:10.1006/jctb.1995.1006.

[17] Robertson, N., Seymour, P. (2004). Graph Minors. XX. Wagner's conjecture. Journal of Combinatorial Theory, Series B, 92 (2), 325-357, doi:10.1016/j.jctb.2004.08.001.

[18] Saeed, N., Celik, A., Al-Naffouri, T.Y. and Alouini, M.S. (2018). Connectivity Analysis of Underwater Optical Wireless Sensor Networks: A Graph Theoretic Approach. In 2018 IEEE International Conference on Communications Workshops (ICC Workshops), Kansas City, MO, USA, pp. 1-6, doi: 10.1109/ICCW.2018.8403740.

[19] Shao, C. Wireless sensor network target localization algorithm based on two- and three-dimensional Delaunay partitions, (2021). Journal of Sensors, 4047684.

[20] Wagner, K. (1937), Uber eine Eigenschaft der ebenen Komplexe. Math. Ann., 114, 570-590, doi:10.1007/BF01594196, S2CID 123534907.

[21] Wang, Y., and Li, X.-Y. (2007). Efficient Delaunay-based localized routing for wireless sensor networks: Research articles. Int. J. Commun. Syst. 20, 767–789.

[22] Watfa, M.K., Mirza, O., Kawtharani, J. (2009). BARC: A Battery Aware Reliable Clustering algorithm for sensor networks. Journal of Network and Computer Applications, 32(6), 1183-1193 https://doi.org/10.1016/j.jnca.2009.05.005.

[23] Wilf, H.S. (1986). Algorithms and Complexity, Prentice Hall, Englewood Cliffs.

[24] Yick, J., Mukherjee, B. and Ghosal, D. Wireless sensor network survey. Computer Networks, 52(12):2292-2330, 2008. https://doi.org/10.1016/j.comnet.2008.04.002

Appendix

DegreeRouteBattery algorithm

In the first step, we consider the variable `LIST`, which contains the list of sensors, each represented by its Euclidean coordinates and battery level percentage. Each element of `LIST` has the format `[[a,b],c]`, where `[a,b]` represents the coordinates and `c` is the battery level. The battery percentage is stored as a vertex attribute.

```
> battery := [];
> for i to nops(LIST) do
> SetVertexAttribute(G, i, "battery" = LIST[i][2]);
> battery := [op(battery), [i, LIST[i][2]]];
> od;
> print(Initial Battery battery);
```

Then, the procedure computes a route according to the decreasing order of the vertex degrees.

In the second step, the distance between every pair of sensors is computed. Let $\{v_i\}_{i=1}^{n}$ denote the vertices corresponding to the sensor locations. A weight matrix is constructed whose elements are the edge weights between each pair of points. These weights are defined as non-negative Euclidean distances and represent communication costs between sensors.

```
> for i from 1 to nops(LIST) do
> point(v||i, LIST[i][1][1], LIST[i][1][2]);
> od;
```

Then we construct the weight matrix whose elements are the euclidean distance between each pair of points. That way, we show the implementation of the subprocedure called `weightmatrix`. It receives as input the coordinates and battery level of each sensor and uses the commands `distance` and `decimal` to express the result rounded to n decimal places. Finally, the output is the weight matrix.

```
> weightmatrix:=proc(L)
> local M; M:=Matrix(1..nops(L),1..nops(L),
shape = symmetric);
> for j from 1 to nops(L)-1 do
> for k from j+1 to nops(L) do
> M[j,k]:=MapleTA:-Builtin:
decimal(2,distance(v||j,v||k));
> od;od;
```

```
> return M;
> end proc:
```

Next, each vertex is checked to determine whether its battery level is below 33%. If so, an edge adjacent to that vertex is contracted, delegating its communication role to a neighboring vertex with higher residual battery. If the battery level is above the threshold, the shortest path to the next vertex in the route is computed using Dijkstra's algorithm. Any intermediate vertices along this path are discharged by 2% due to their activation. Finally, the battery of each vertex is discharged according to its degree and the battery attributes are updated. This procedure is applied to each vertex in the graph, regardless of whether it has been merged through contraction. The updated graph and battery information are then returned.

```
> decreasingbatord := proc(G, Rvert, Cvert)
> local vertices, bdr, v, battery, VertIndex, newbattery, oldbattery, dist,
sp, sum, shortP, Rv, Cv, C, H, N, h, i, j, k, P, l, r, verts, Q, R, route;
> Cv := Cvert;
> Rv := Rvert;
> route := [];
> H := G;
> for v in Vertices(H) do
> route := [op(route), [Degree(G, v), v]];
> od;
> route := sort(route, (a, b) -> b[1] < a[1]);
> route := [route[i][2] $ (i = 1 .. nops(route))];
> print(Route, route);
> for r to nops(route) do
> if GetVertexAttribute(H, route[r], "battery") < 33 then
> shortP := [];
> sp := [];
> dist := [];
> Rv := [op(Rv), route[r]];
> N := Neighbors(H, route[r]);
> for i to nops(N) do
> for j from i + 1 to nops(N) do
> shortP := [op(shortP), [DijkstrasAlgorithm(H, N[i], N[j])[1],
DijkstrasAlgorithm(H, N[i], N[j])[2]]];
> od; od;
> P := [seq(GetVertexAttribute(H, l, "battery"), l in N)];
> Q := FindMaximalElement(P, position)[2];
> R := {N[Q], route[r]};
> C := [N[Q], GetVertexAttribute(H, N[Q], "battery")];
> Cv := [op(Cv), C];
> H := foldl(Contract, H, R);
```

```
> verts := Vertices(H);
> if route[r] in verts then
> for i to nops(verts) do
> if verts[i] = route[r] then
> VertIndex := i;
> verts[VertIndex] := C[1];
> H := RelabelVertices(H, verts);
> SetVertexAttribute(H, verts[VertIndex], "battery" = C[2]);
> fi; od; fi;
> if 0 < nops(shortP) then
> for i to nops(shortP) do
> if evalb({shortP[i][1][-1], shortP[i][1][1]} in Edges(H)) = true then
> SetEdgeWeight(H, {shortP[i][1][-1], shortP[i][1][1]}, shortP[i][2]);
> else
> AddEdge(H, [{shortP[i][1][-1], shortP[i][1][1]}, shortP[i][2]]);
> fi; od; fi;
> else
> for j from r + 1 to nops(route) do
> sp := DijkstrasAlgorithm(H, route[r], route[j])[1];
> if 2 < nops(sp) then
> for k from 2 to nops(sp) - 1 do
> oldbattery := GetVertexAttribute(H, sp[k], "battery");
> SetVertexAttribute(H, sp[k], "battery" = oldbattery - 2);
> od; fi;
> break;
> od; fi; od;
> vertices := Vertices(H);
> newbattery := [];
> bdr := [];
> for v in vertices do
> bdr := Degree(H, v);
> oldbattery := GetVertexAttribute(H, v, "battery");
> SetVertexAttribute(H, v, "battery" = oldbattery - bdr);
> newbattery := [op(newbattery), [v, GetVertexAttribute(H, v, "battery")]];
> od;
> return H, newbattery, Rv, Cv;
> end proc:
```

The previous implementation is applied inside a while loop with the condition of having more than one vertex. That way, when there is just one vertex left, we finish the procedure and all the information of the sensors is in that vertex with a good battery level. In case that we have more than one vertex but all of them have low battery, the procedure is stopped and a message is printed. Moreover, in each iteration the graph is drawn to see the evolution of the graph.

```
> vert := Vertices(G);
> DrawGraph(G);
> steps := nops(vert);
> numvertices := nops(vert);
> RemovedVertices := [];
> ContractedVertices := [];
> while 1 < numvertices do
> G, battery, RemovedVertices, ContractedVertices := decreasingbatord(G,
RemovedVertices, ContractedVertices);
> numvertices := nops(Vertices(G));
> count := 0;
> for b in battery do
> if b[2] < 33 then count := count + 1;
> fi; od;
> if count = numvertices then
> print(ALL SENSORS LEFT HAVE LOW BATTERY battery);
> break;
> fi;
> DrawGraph(G);
> od;
```

BrokenNodeRecovery algorithm

```
> brokenNode := proc(PTS)
> local failedPoint, remainingPTS, i;
> failedPoint := Generate(integer(range = 1 .. nops(PTS)));
> remainingPTS := [];
> for i to nops(PTS) do
> if failedPoint <> i then
> remainingPTS := [op(remainingPTS), PTS[i]];
> fi; od;
> return remainingPTS, failedPoint;
> end proc:
```

In order to complete the implementation of this routine, we generate the number of sensor failures, `numFailures`. This number is generated randomly and its value is between 1 and the number of vertices minus one, $(n-1)$. Then, we display the initial Voronoi diagram and Delaunay Graph. In the implementation, we create a loop that repeats as many times as the value of `numFailures`. Within the loop, we call the function and display the coordinates of the remaining sensors, the failed sensor, and the representations of the Voronoi diagram and Delaunay Graph. If there are 2 or fewer vertices, we show a simple graph, as it is not possible to create a Delaunay Graph with fewer than 3 vertices.

```
> DG := DelaunayGraph(PTS);
> numVertices := nops(Vertices(DG));
> numFailures := Generate(integer(range = 1 .. numVertices - 1));
> DrawGraph(DG);
> VoronoiDiagram(PTS);
> for n from 1 to numFailures do
> PTS, failedPoint := brokenNode(PTS);
> print(PTS);
> print(Failed*Node, failedPoint);
> if nops(PTS) = 2 then
> print("There are not enough nodes to create a Delaunay Graph");
> H := Graph(nops(PTS), {{1, 2}});
> SetVertexPositions(H, PTS);
> elif nops(PTS) = 1 then
> print("There are not enough nodes to create a Delaunay Graph");
> H := Graph(nops(PTS));
> else
> H := DelaunayGraph(PTS);
> fi;
> DrawGraph(H);
> VoronoiDiagram(PTS);
> od;
```

Eulerian path method

```
> eulerianGraph := proc(H)
> local count, i, j, v, w, path, verts, N, vertices, edges, visited;
> if IsEulerian(H, 'T') then
> print("The graph is eulerian");
> print(The eulerian cycle is T);
> else
> count := 0;
> verts := [];
> for v in Vertices(H) do
> if type(Degree(H, v), odd) then
> count := count + 1;
> verts := [op(verts), v];
> fi; od;
> if count = 2 then
> print("The graph is semieulerian");
> vertices := Vertices(H);
> edges := Edges(H);
> visited := {op(vertices)};
> path := [];
> for i to nops(edges) do
> if edges[i][1] = verts[1] and not edges[i] in visited then
> visited := visited union {edges[i]};
```

```
> path := [op(path), verts[1], edges[i][2]];
> w := edges[i][2];
> break;
> elif edges[i][2] = verts[1] and not edges[i] in visited then
> visited := visited union {edges[i]};
> path := [op(path), verts[1], edges[i][1]];
> w := edges[i][1];
> break;
> fi; od;
> while nops(path) <= nops(edges) do
> N := Neighbors(H, w);
> for i to nops(N) do
> for j to nops(edges) do
> if edges[j][1] = w and edges[j][2] = N[i] and not edges[j] in visited then
> visited := visited union {edges[j]};
> path := [op(path), N[i]];
> w := N[i];
> break;
> elif edges[j][2] = w and edges[j][1] = N[i] and not edges[j] in visited then
> visited := visited union {edges[j]};
> path := [op(path), N[i]];
> w := N[i];
> break;
> fi; od; od; od;
> print("Semieulerian Path:", path);
> else
> print("The graph isn't eulerian or semieulerian");
> fi; fi;
> end proc:

> createHullEdges := proc(hull)
> local edges, i;
> edges := [];
> for i to nops(hull) - 1 do
> edges := [op(edges), {hull[i + 1], hull[i]}];
> od;
> edges := [op(edges), {hull[-1], hull[1]}];
> return edges;
> end proc:

> findOtherEdges := proc(allEdges, hullEdges)
> local otherEdges, edge, i;
> otherEdges := [];
> for edge in allEdges do
> if not edge in hullEdges then
> otherEdges := [op(otherEdges), edge];
> fi; od;
```

```
> return otherEdges;
> end proc:

> countElements := proc(otherEdges)
local elements, counts, element, i, vertexSet, pair, Index, elementCounts;
> vertexSet := [];
> for pair in myList do
> vertexSet := [op(vertexSet), op(pair)];
> od;
> elements := [];
> counts := [];
> for element in vertexSet do
> if element in elements then
> Index := NULL;
> for i to nops(elements) do
> if elements[i] = element then
> Index := i;
> break;
> fi; od;
> counts[Index] := counts[Index] + 1;
> else elements := [op(elements), element];
> counts := [op(counts), 1];
> fi; od;
> elementCounts := [seq([elements[i], counts[i]], i = 1 .. nops(elements))];
> return elementCounts;
> end proc:

> isK3 := proc(otherEdges)
> local vertices, vertexSet, i, j, pair, count, counts, index, G;
> counts := countElements(otherEdges);
> count := 0;
> index := [];
> for i to nops(counts) do
> if counts[i][2] = 2 then
> count := count + 1;
> else
> index := [op(index), i];
> fi; od;
> if count = nops(counts) then
> return "Yes, there is 1 intern K3";
> else
> count := 0;
> for i in index do
> if counts[i][2] mod 2 = 0 then
> count := count + 1;
> fi; od;
> if count = nops(index) then
```

```
> return "Yes, all the intern edges create ", 1/3*nops(otherEdges), " K3";
> else
> return "No, all the intern edges do not create K3";
> fi; fi;
> end proc:
```

Chromatic number method

```
> wheelCondition := proc(H)
> local n, Wn, isomorphicResult;
> for n from 3 by 2 to 19 do
> Wn := WheelGraph(n);
> isomorphicResult := IsSubgraphIsomorphic(Wn, H, isomorphism);
> if not (isomorphicResult = false) then
> print("The graph or a subgraph is a Wheel Graph", W_{n + 1});
> print("The wheel graph is even");
> print("The chromatic number must be 4");
> break;
> fi; od;
> if isomorphicResult = false then
> for n from 4 by 2 to 20 do
> Wn := WheelGraph(n);
> isomorphicResult := IsSubgraphIsomorphic(Wn, H, isomorphism);
> if not (isomorphicResult = false) then
> print("The graph or a subgraph is a Wheel Graph", W_{n + 1});
> print("The wheel graph is odd");
> print("The chromatic number must be 3");
> break;
> fi;
> if n = 20 then
> print("The graph or a subgraph is not a Wheel Graph");
> fi; od; fi;
> end proc:

> chromaticNum := proc(H)
> if IsEulerian(H, 'T') then
> print("The graph is Eulerian");
> print("The chromatic number must be 3");
> else
> print("The graph is not Eulerian");
> WheelCondition(H);
> fi;
> print("The chromatic number is:", ChromaticNumber(H, 'col'));
> print(col);
> end proc:
```