**Applied and Computational Mathematics**
2025; 14(1): 64-77
http://www.sciencepublishinggroup.com/j/acm
doi: 10.11648/j.acm.20251401.15
ISSN: 2328-5605 (Print); ISSN: 2328-5613 (Online)

# Graph-based Algorithms for Battery Management in Sensor Networks: Theoretical Insights and Practical Applications

**María Millán[1], Manuel Ceballos[1, *], Luis Orihuela[2]**

[1]Department of Engineering, Faculty of Engineering, Universidad Loyola Andalucía, Sevilla, Spain
[2]Department of Electronic Engineering, Computer Systems, and Automation, Faculty of Engineering, Universidad de Huelva, Huelva, Spain

## Email address:

mmillangordillo@al.uloyola.es (María Millán), mceballos@uloyola.es (Manuel Ceballos), luis.orihuela@diesia.uhu.es (Luis Orihuela)
[*]Corresponding author

## To cite this article:

María Millán, Manuel Ceballos, Luis Orihuela. (2025). Graph-based Algorithms for Battery Management in Sensor Networks: Theoretical Insights and Practical Applications. *Applied and Computational Mathematics, 14*(1), 64-77. https://doi.org/10.11648/j.acm.20251401.15

**Received:** 26 January 2025; **Accepted:** 10 February 2025; **Published:** 17 February 2025

**Abstract:** Battery level management is crucial for the reliable operation of sensor networks. In this paper, we present a novel framework that leverages graph theory and computational geometry to enhance battery management, optimize communication, and maintain connectivity. Our approach builds weighted Delaunay Graphs as combinatorial models for sensor networks and investigates theoretical aspects of their hamiltonicity. We introduce six algorithmic methods addressing two main objectives. The first set focuses on battery management by identifying low-battery sensors, computing minimum spanning trees and shortest paths and performing edge contraction to simplify network topology while preserving connectivity. The second set examines the hamiltonicity of Delaunay Graphs using graph labeling and combinatorial techniques to analyze the existence of Hamiltonian paths. Our methodology integrates key graph procedures such as minimum spanning trees, shortest path algorithms, edge contraction, and graph labeling with computational geometry tools like Voronoi diagrams and Delaunay triangulation. This combination allows for an accurate modeling of spatial relationships within the network and dynamic adaptation to battery constraints. A case study on monitoring a sugar cane field demonstrates the practical application of our methods. The computational study reveals the efficiency and robustness of the proposed algorithms in optimizing battery usage and improving network performance in real-world scenarios. In conclusion, our work not only addresses critical battery management challenges in sensor networks but also deepens the understanding of the interplay between graph theory and network optimization. This integrated approach paves the way for further research in both theoretical and applied domains, offering promising solutions for enhancing the sustainability and reliability of sensor networks

**Keywords:** Weighted Graph, Voronoi Diagram, Delaunay Graph, Sensor Network, Algorithm

## 1. Introduction

Finding connections between different fields is one of the most stimulating and significant areas of research in Science, Engineering, and particularly Mathematics. Novel techniques allow researchers to address unresolved problems, refine existing theories, and achieve groundbreaking results. This paper explores the link between Graph Theory and sensor networks, leveraging graph-based tools to enhance network optimization and management.

Graph Theory provides a powerful framework for modeling and analyzing complex networks, making it highly useful for studying sensor networks. For instance, [3] investigates graph-based optimization techniques for node placement, routing, and data aggregation, aiming to minimize energy consumption and extend network lifetime. Additionally, graph-based algorithms contribute to network topology construction, data routing, and energy-efficient communication in wireless sensor networks [11], while also addressing load balancing and connectivity issues [8].

Sensor networks play a crucial role in the expanding Internet of Things (IoT), enabling data collection and transmission for diverse applications such as environmental monitoring, industrial automation, healthcare, and smart cities [2]. However, they face challenges related to power constraints

and computational limitations, requiring efficient energy management strategies to ensure system reliability [10]. Various mathematical battery models have been developed to address these issues [17].

This paper focuses on improving battery level management in sensor networks through graph-based techniques, ensuring network reliability and longevity. Our algorithmic approach identifies sensors with low battery and adapts network topology to prevent communication breakdowns. We employ minimum spanning trees and shortest path algorithms for connectivity, edge contraction for topology simplification, and graph labeling for resource optimization. Additionally, Voronoi diagrams and Delaunay triangulation help model spatial relationships within the network.

Another key objective is the structural analysis of sensor networks by associating them with Delaunay Graphs. We investigate Hamiltonian paths addressing gaps in the literature regarding the open problem of whether Delaunay triangulations always contain a Hamiltonian path. This question has been affirmatively answered under the square distance ($L^\infty$-metric) [1]. Section 4 presents theoretical results on this, complemented by an algorithmic procedure to analyze Hamiltonicity of Delaunay graph. A case study on sugar cane field monitoring demonstrates the practical application of this method.

The paper is structured as follows: Section 2 reviews fundamental concepts in Graph Theory and Computational Geometry. Section 3 explores their connection to sensor networks. Section 4 presents theoretical results on hamiltonicity of Delaunay Graphs. Section 5 describes four algorithms: three for battery management and one for hamiltonicity. Section 6 applies these methods to a real-world agricultural case study. Finally, we show some conclusions 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 [5] and [4], respectively.

### 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 edge-set, which is given by unordered pairs of a couple of nodes. It is possible to associate a *weight* to each edge and, in such case, we will say that $G$ is a *weighted graph*.

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 on 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, we can define the *minimum spanning tree* as the spanning tree having the minimum possible sum of the weight of its edges. In Figure 1, we show a weighted graph, then a non-minimum spanning tree whose total weight is 11 and a minimum spanning tree with weight 7.

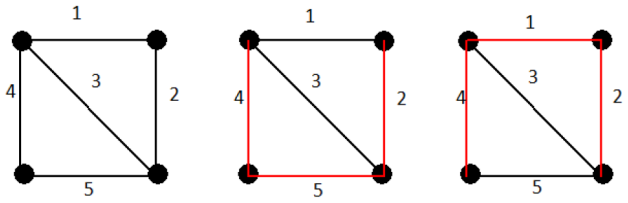


***Figure 1.*** *Example of non-minimum and minimum spanning trees in a weighted graph.*

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 $w$. 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 [15] which characterizes the planar graphs as being the graphs that do not have the complete graph $K_5$ or the complete bipartite graph $K_{3,3}$ as minors. This operation plays a very important role in the works of Robertson and Seymour [12, 13]. Their Graph Minor Theorem [6], 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. In this way, edge contraction is used to study the preservation of connectivity properties. This is significant in designing efficient networks and algorithms.

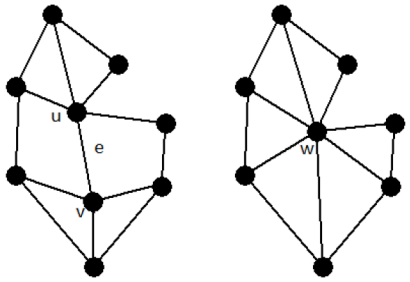


***Figure 2.*** *Example of edge contraction.*

### 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 $Vor(p_i) = \{q \in \mathbb{R}^2 \mid d(q, p_i) \leq d(q, p_j), \forall j \neq i\}$. It is satisfied that $Vor(p_i) = \bigcap_{i \neq j} h(p_i, p_j)$, where $h(p_i, p_j)$ is the region which contains $p_i$ and has as border the bisector of the segment joining $p_i$ and $p_j$. The Voronoi diagram of $S$ is the tessellation of the euclidean plane given by $Vor(S) = \{Vor(p_i)\}_{i=1,\ldots,n}$. Figure 3 shows an example of this kind of diagrams.

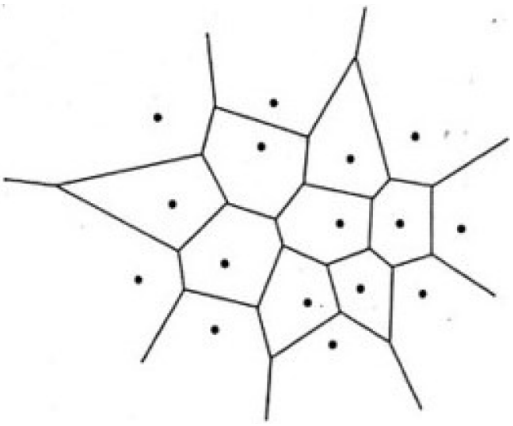

***Figure 3.*** *Voronoi diagram.*

The *dual graph* of $Vor(S)$ is defined as the graph having $S$ as set of vertices and two vertices of $S$ are adjacent in case that they share a border in the Voronoi diagram. This graph is usually called the *proximity graph or Delaunay Graph* of $S$. If the points are not aligned and there is no group of 4 cocircular vertices, the Delaunay Graph is a triangulation. An example of this is shown in Figure 4.

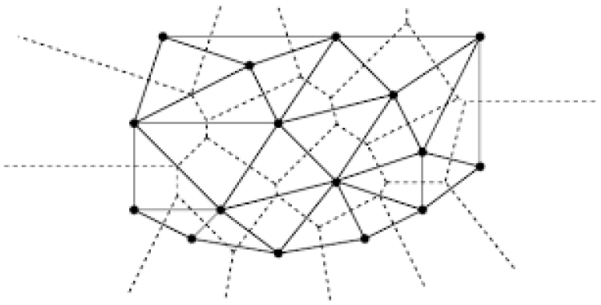

***Figure 4.*** *Delaunay Graph.*

The *convex hull* of a set of points $V$ is the smallest convex set containing $V$. In case that $V$ is a finite set in the plane, the convex hull of $V$ is a convex polygon. From here on, we denote by $CH(V)$ the bordering cycle of the convex hull of a set of points in the plane $V$. Finally, an *internal triangle* in a graph is a triangle (3-cycle) in which at least 2 vertices are not in $CH(V)$ or its 3 edges are not in $CH(V)$. We will denote by $IT(G)$ the set of internal triangles of a given graph $G$.

# 3. Graph Theory and Sensor Networks

In this section, we establish the connection between Graph Theory and sensor networks, focusing on the association of a sensor network with a weighted Delaunay Graph. This association has been used in previous works such as [14, 16]. The main idea is that each sensor is represented as a vertex (or node) in the graph, and edges are created based on the geometric proximity of sensors in the physical space. This approach captures both the spatial arrangement and the communication links between sensors, providing an intuitive and efficient model of the network. The edges are further weighted using the Euclidean distance between connected vertices, reflecting the actual physical distances between sensors.

After constructing the weighted Delaunay Graph, we address the challenge of optimizing data collection across the sensor network. To achieve this, we explore the use of Hamiltonian paths, which visit every sensor exactly once. These paths are critical for designing efficient data collection routes, minimizing communication delays, and reducing network congestion. Additionally, studying Hamiltonian paths provides insights into the network's connectivity and robustness. By identifying such paths, we can enhance the network's resilience to sensor failures, ensuring continuous data flow and adapting to dynamic changes in the system.

# 4. Theoretical Results

In this section, we show several theoretical results regarding the hamiltonicity of Delaunay Graphs associated with sensor networks. From here on, $DT(V)$ will denote the Delaunay Graph associated to the set of nodes $V$.

First, it is possible to estimate the number of triangles for a Deluany Graph according to its planarity and Euler's Theorem as it is shown in the following

*Lemma* 4.1. The number of triangles in $DT(V)$ is at most $2 \cdot |V| - 5$.

*Proof.* $DT(V)$ is planar, therefore, if $DT(V)$ is a graph with $n$ vertices and $m$ edges verifying $m \leq 3 \cdot n - 6$. Moreover, according to Euler's Theorem, $|C| + n - m = 2$, where $|C|$ is the number of faces of $DT(V)$. That number of faces will be the number of triangles plus the external face. 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 Hamiltonian cycles as we show in the following results.

*Proposition* 4.1. If $IT(DT(V)) = \emptyset$, then $DT(V)$ is Hamiltonian. Furthermore, either $DT(V) \cong W_n$ or $V \subseteq CH(V)$ $(V = V(CH(V)))$.

*Proof.* First of all, we can assume that $|V| = n > 3$ since in case that $|V| = 3$ we would have a 3-cycle which is trivially Hamiltonian. Since $n > 3$, $CH(V) \neq DT(V)$, that is, $DT(V)$ is not a 3-cycle. Now, we distinguish several cases.

*Case 1:* $\exists u \in V / u \notin CH(V)$

Then, $\forall v \in V$ with $v \neq u$, it is satisfied that $v \in CH(V)$. If $v \in V$ with $v \neq u$ verifies that $v \notin CH(V)$, then $\exists w \in V / uvwu$ is a 3-cycle. Furthermore, $uvwu$ is an internal triangle and $IT(DT(V)) = \emptyset$, but this is a contradiction. Therefore, there exists a unique $u \in V - CH(DT(V))$, $CH(V)$ has $n-1$ vertices and $DT(V) \simeq C_{n-1} + K_1 = W_n$. Therefore, $DT(V)$ is Hamiltonian since every wheel graph is.

*Case 2:* $V \subseteq CH(V)$

Then, $DT(V)$ is the cycle $C_n$ adding some edges until creating the triangulation. Consequently, $DT(V)$ is Hamiltonian and the Hamiltonian cycle is given by $CH(V)$.

*Proposition* 4.2. If $|IT(DT(V))| < 3$, then $DT(V)$ is Hamiltonian.

*Proof.* In case that $IT(DT(V)) = \emptyset$ we know that $DT(V)$ is Hamiltonian due to Proposition 4.1.

Let us assume that $DT(V)$ has one unique internal triangle, $T$, whose vertices are $\{u, v, w\}$. Since $T$ is an internal triangle, each one of the edges $uv$, $uw$ and $vw$ belongs to another 3-cycle. Now, we have two possibilities. On one hand, we suppose that there exist another three vertices $\{a, b, c\}$ such that $auv$, $bvw$ and $cuw$ determine the other three triangles containing the edges $uv$, $uw$ and $vw$, respectively. In this case, those vertices $\{a, b, c\}$ have degree two, $|V| = 6$ and $V = V(CH(V))$. Consequently, $CH(V)$ is the Hamiltonian cycle $C_6$ and we conclude that $DT(V)$ is Hamiltonian. On the other hand, if the previous situation is not satisfied, then $V = \{u, v, w, a, b_1, \ldots, b_r\}$, where $a$ is the unique vertex with degree two. Moreover, $a$ is adjacent to two vertices among $\{u, v, w\}$. Without loss of generality, we can consider $u$ and $v$ to be those vertices. Then, $w$ is the unique vertex which does not belong to $CH(V)$ and the graph induced by $V \setminus \{a\}$ is a wheel graph defined by $C_{r+2} + w$. Every wheel graph is Hamiltonian and that Hamiltonian cycle can be extended considering also the edges $au$ and $av$. These two possible cases are represented in Figure 5.

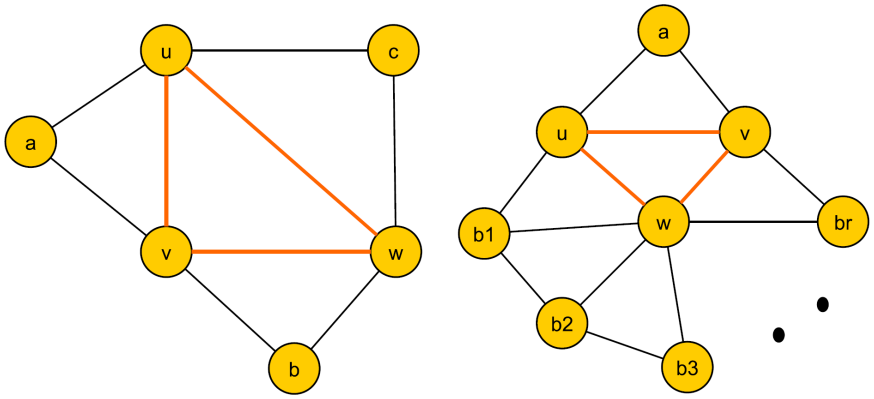


***Figure 5.*** *Delaunay Graphs with one internal triangle.*

Now, we suppose that $DT(V)$ has two internal triangles, $T_1$ and $T_2$. We have two possible cases since these two triangles can share a vertex or an edge. In case that $T_1$ and $T_2$ have a common vertex, $u$, they determine a friendship graph $F_2$ and the remaining vertices of $DT(V)$ belong to $CH(V)$ since otherwise there would be more than two internal triangles. Following a similar reasoning as before with the wheel graph it is possible to obtain a Hamiltonian cycle containing the vertices from $CH(V)$ and $u$. In case that $T_1$ and $T_2$ share an edge then both triangles are contained in a 4-cycle, $C_4$. The Hamiltonian cycle of $DT(V)$ is constructed by the union of $C_4$ and $CH(V)$. We show an example of both situations in Figure 6.

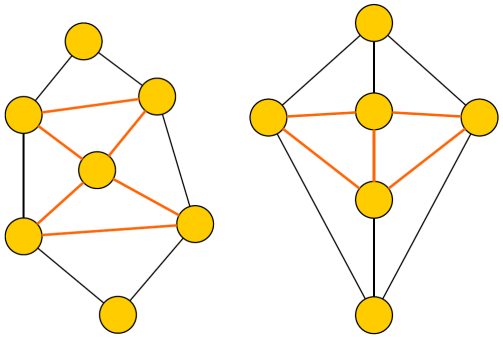

***Figure 6.*** *Delaunay Graphs with two internal triangles.*

However, it is not true that every Delaunay Graph is Hamiltonian:

*Proposition* 4.3. There exist Delaunay Graphs that are not Hamiltonian.

*Proof.* Let us consider the set of vertices $V = \{v_1, v_2, \ldots, v_9\}$ and the Delaunay Graph, $DT(V)$, of Figure 7. Let $C$ be a Hamiltonian cycle of $DT(V)$. The neighbours of $v_4$ in $C$ must be $v_3$ and $v_7$, since they are the only adjacent vertices to $v_4$. Similarly, the neighbours of $v_6$ have to be $v_5$ and $v_7$ and the neighbours of $v_1$ will be $v_2$ and $v_5$. Every vertex on a cycle has exactly two adjacent vertices. Therefore, the neighbours of $v_7$ will be $v_4$ and $v_6$ and the neighbours of $v_5$ will be $v_1$ and $v_6$. Consequently, $v_5$ and $v_7$ cannot be neighbours of $v_8$ in $C$. However, $v_8$ must have as neighbours two of the vertices $\{v_5, v_7, v_9\}$. This is a contradiction.

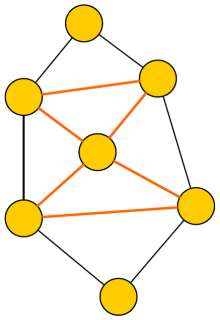

***Figure 7.*** *Non-Hamiltonian Delaunay Graph.*

Notice that in order to find a non-Hamiltonian Delaunay Graph we have needed to consider a configuration with five internal triangles.

In [1] the authors proved that a Delaunay Graph always contains a Hamiltonian path when the distance considered is the square one ($L^\infty$-metric). It is an open problem if this is also true for the euclidean distance.

Finally, in the preliminary section, we mentioned the edge contraction operation. As we said before, this operation preserves several graph properties such us connectivity. One may wonder if this operation also preserves hamiltonicity. This is not true as we can see in the following

*Example* 4.1. Figure 8 shows a Hamiltonian graph that after applying the contraction of the edge $e$, we obtain a graph containing the cut vertex $w$. Therefore, this last graph is not Hamiltonian.

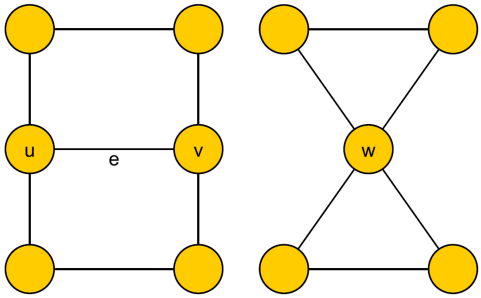


***Figure 8.*** *Hamiltonicity is not preserved under edge contraction.*

# 5. Algorithmic Procedures

In this section, several algorithmic methods are introduced. The main goal is to collect all the data saved by a group of sensors bearing in mind their battery level. Another aim is to avoid possible failures due to the existence of broken nodes. Our last objective is to study the hamiltonicity 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 labelling. 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, working 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.

## *5.1. BatteryContract Algorithm*

This algorithm begins by generating $n$ random sensors, each with a randomly assigned battery level. It then calculates the Euclidean distance between each pair of sensors and constructs the corresponding graph. The battery levels of all sensors are checked, and the shortest route is determined to collect data from each one. The first step is executed by the subprocedure `randoms`, which takes a natural number $n$ and generates a list of tuples. Each tuple contains two random coordinates within $[0, 100] \times [100, 0]$ and a battery percentage between 1 and 100, representing the sensors' locations.

```
> randoms:=proc(n)
> local L; L:=[];
> for i from 1 to n do
> L:=[op(L),[[rand(0..100)(),rand(0..100)()],
rand(1..100)()]]; od;
> return L;
> end proc:
```

Finally, we execute this subprocedure and extract only the list of points with the euclidean coordinates of each sensor.

```
> LIST:=randoms(n);
> PTS:=[seq(LIST[i][1], i=1..nops(LIST))]
```

Notice that in case of having the real location of a group of sensors and real values for their level battery, then we can avoid this subprocedure and directly define the variables `LIST` and `PTS`. In fact, this will be done in Section 6. The second subprocedure has as main goal computing the distance between every pair of sensors. In order to do so, we first denote each point from the previous list by $\{v_i\}_{i=1}^n$.

```
> for i from 1 to nops(LIST) do
> point(v||i, PTS[i][1], PTS[i][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:
```

The third step of our algorithm starts computing the Voronoi Diagram in order to visualize the area of influence of the points.

```
> VoronoiDiagram(PTS);
```

Then, we obtain the Delaunay Graph, which is the dual of Voronoi.

```
> DG:=DelaunayGraph(PTS);
> DrawGraph(DG);
```

After that, we define the weighted graph using the Delaunay Graph and the weight matrix.

```
> G:=MakeWeighted(DG,W)
> DrawGraph(G)
```

Finally, the last part of this step of our algorithmic method is the implementation of the subprocedure `batteryhighlight`. It highlights the vertices whose battery percentage is smaller than 33%. The arguments needed are the weighted graph and the list of points and batteries. The procedure starts by creating an empty set, $M$, and, using a condition, checks if the percentage is smaller than 33% and, if it is true, adds the index to the set $M$. After the loop is completed, the procedure runs the command `HighlightVertex` using as arguments the weighted graph, the set $M$, and the color red as attached label. At the end, the procedure returns the drawing of the graph.

```
> batteryhighlight:=proc(G,L)
> local M; M:={};
> for i from 1 to nops(L) do
> if L[i][2]<=33 then M:={op(M),i};
> fi; od;
> HighlightVertex(G, M, "Red");
> return DrawGraph(G);
> end proc:
```

The `batterycontract` procedure computes a minimum spanning tree (MST) while addressing potential low-battery issues. It first extracts all edges and weights from the original graph $G$, storing them for later restoration. The algorithm then identifies "low-battery nodes" (red vertices) with battery levels below a predefined threshold (33%). If no red vertices are found, it directly computes the MST. Otherwise, it removes edges connecting red vertices, as these connections are considered non-viable. Next, the algorithm reconnects red vertices by selecting the minimum-weight edge linking them to a neighboring node. It then contracts these edges, merging vertices while preserving structural integrity. Isolated red vertices are removed, while non-red ones are reconnected if possible. If the modified graph still contains three or more vertices, an MST is computed; otherwise, the final structure is displayed. This process ensures efficient data collection while optimizing network connectivity and complexity.

```
> batterycontract:=proc(G,L)
> local M, N,P,Q,R,S,H,contractedPairs,
G_new,edge,verts,notM,edgeWeights,ew;
> M:=[];N:=[];P:=[];Q:=[];S:=[seq(L[i][1],
i=1..nops(L))];
> edgeWeights:= [];
> for edge in Edges(G) do
> edgeWeights:= [op(edgeWeights),
[edge, GetEdgeWeight(G, edge)]];
> end do;
> for i from 1 to nops(L) do
> if L[i][2]<=33 then M:=[op(M),i];
> fi; od;
> if M=[] then return
> DrawGraph(EuclideanMinimumSpanningTree(S));
> elif nops(M) = nops(L) then
> return "All vertices have low battery.";
> fi;
> if nops(M)>1 then
> for i from 1 to nops(M)-1 do
> for j from i+1 to nops(M) do
> if evalb({M[i],M[j]} in Edges(G))=true
> then
> G:=DeleteEdge(G,{M[i],M[j]});
> fi; od; od; fi;
> for j from 1 to nops(M) do
> if Neighbors(G, M[j]) = [] then
> G:= DeleteVertex(G, M[j]);
> else N:= [op(N), Neighbors(G, M[j])];
> fi; od;
> for k from 1 to nops(N) do
> P:=[op(P),
[seq(GetEdgeWeight(G,{M[k],N[k][l]}),
l=1..nops(N[k]))]];
> od;
> for h from 1 to nops(P) do
> Q:=[op(Q),FindMinimalElement(P[h],
position)[2]];
> od;
> R:=seq({M[i],N[i][Q[i]]},i=1..nops(Q));
> contractedPairs:= R;
> G_new:= G;
> for edge in contractedPairs do
> if not evalb({edge[1],edge[2]} in
Edges(G_new))
> then
> print("Edge does not exist in the graph:
", edge);
> else
> G_new:= Contract(G_new, edge);
> verts:= Vertices(G_new);
> if edge[1] in M then
> verts:=map(v->`if`(v=edge[1],edge[2],v),
verts);
> fi;
> if edge[2] in M then
> verts:=map(v->`if`(v=edge[2],edge[1],v),
verts);
> fi;
> G_new:= RelabelVertices(G_new, verts);
> fi; od;
> for edge in Edges(G_new) do
> for ew in edgeWeights do
> if edge = ew[1] then
> SetEdgeWeight(G_new, edge, ew[2]);
> break;
> fi; od; od;
> if nops(verts) >= 3 then
> S:=subsop(seq(M[i] = NULL,i=1..nops(M)),S);
> return
> DrawGraph(EuclideanMinimumSpanningTree(S));
> else notM:= [];
> for i to nops(L) do
> if not i in M then
> notM:= [op(notM), i];
> fi; od;
> StyleVertex(G_new, notM, color = gray,
fontcolor = black);
> return DrawGraph(G_new);
> fi;
> end proc;
```

### *5.2. BatteryOrder Algorithm*

In this algorithmic procedure, we deal with a similar problem, but this time we construct an spanning tree following a specific order with respect to the sensors. More concretely, we travel throughout the points following an increasing battery level order. We will start from the sensor having the lowest battery level and we will finish at the sensor with the highest battery level. In this way, we have implemented the procedure `increasingbatord`, which receives as inputs a graph $G$ and a list containing the coordinates of the sensors with each battery level. The output is the list of increasing ordered vertices according to the battery level, the optimum path for

traveling from the point with lowest battery level to the one with most and the total distance of that path. In order to optimize the routes from a vertex to another we have used Dijkstra's algorithm bearing in mind that if a vertex is visited while going through a route among other two, that vertex is considered as visited and that, therefore, we have collected all the data from it.

Concerning the implementation, we first create a list, $N$, containing the battery percentage and the number of the vertex. Then this list is sorted by the battery level percentage. We continue the process by defining $P$, which is the order of the vertices by battery percentage. Now, we check if $P$ has more than one element. If it does, we continue with the procedure. Otherwise, we check if there are elements in the variable `solutions` to see if there is only one vertex in the graph or if there is just one vertex left in $P$ and, therefore, we have to finish the procedure. When $P$ has more than one element, we add elements to `solutions`, which is a list containing the Dijkstra's Algorithm solution to obtain the minimum path from an element of $P$ to the next. A variable `total` is created to obtain the total weight of the route. Also, a list `subgraphs` is created to plot every `subgraph` containing the routes provided by the Dijkstra's Algorithm. Finally, we create a list called `visited` to remove the vertices we have seen trough the path to avoid visiting them in the future.

```
> increasingbatord:= proc(G, L)
> local N,P,i,j,k,solutions,solution,total,
plot,plots,subgraph,subgraphs,count,visited,t;
> N:= [[L[i][2], i] $ (i = 1.. nops(L))];
> print(N);
> N:= sort(N);
> print(N);
> P:= [N[i][2] $ (i = 1.. nops(N))];
> print(P);
> total:= 0;
> solutions:= [];
> subgraphs:= [];
> count:= 0;
> while 0 < nops(P) do
> visited:= [];
> if nops(P) = 1 then
> if 0 < nops(solutions) then
> if solutions[-1][1][-1] = P[1] then
> P:= [];
> break;
> else
> solution:= DijkstrasAlgorithm(G,
solutions[-1][1][-1], P[1]);
> solutions:= [op(solutions), solution];
> total:= total + solution[2];
> subgraph:=InducedSubgraph(G,solution[1]);
> subgraphs:= [op(subgraphs), subgraph];
> P:= [];
> fi;
> else
> print(The graph only has one vertex);
> break;
> fi;
> else
> solution:=DijkstrasAlgorithm(G, P[1],P[2]);
> solutions:= [op(solutions), solution];
> total:= total + solution[2];
> subgraph:= InducedSubgraph(G, solution[1]);
> subgraphs:= [op(subgraphs), subgraph];
> fi;
> count:= count + 1;
> if 2 <= nops(solutions[count][1]) then
> for t to nops(solutions[count][1]) - 1 do
> visited:=[op(visited),solutions[count][1][t]];
> od; fi;
> P:= remove(x -> x in visited, P);
> od;
> print(solutions);
> print("Total: ", total);
> plots:= [];
> for subgraph in subgraphs do
> plot:= DrawGraph(subgraph, layout = tree);
> plots:= [op(plots), plot];
> od;
> plots:=[op(plots),DrawGraph(G,layout=tree)];
> op(plots);
> end proc:
```

### *5.3. DischargeContraction Algorithm*

This algorithm shows the evolution of the graph when we consider a similar situation to the one dealt with in the BatteryContract algorithm, but this time, apart from the initial battery level, every sensor has a specific discharge battery rate. This rate will be applied in each iteration of the procedure and it will be proportional to the vertex degree in the Delaunay Graph. In this way, the implementation of the main procedure `batterywithbdr` receives as inputs the weighted Delaunay Graph, a list of the removed vertices and another list containing the vertices which the removed ones are adjacent to edges that have been contracted to and their battery percentages in the moment of the contraction. The outputs are the graph, a list with the batteries in that iteration and the updates of the vertices removed and contracted lists. This algorithmic method is implemented following these steps:

1. Assigning an initial battery percentage to each vertex as an attribute.
2. Performing contractions for an edge adjacent to a vertex with less than $33\%$ of battery.
3. Repeating the contraction and battery reduction process until only one vertex remains or all remaining vertices have low battery.

In the first step, we start creating an attribute to each vertex. This attribute is called `battery` and stores the battery level percentage.

```
> 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);
```

Our algorithm first checks if there are vertices with battery level lower than $33\%$. In such case, that vertex is added to a list $M$. If the list $M$ has more than one vertex, we impose that those vertices are not mutually adjacent to avoid contractions of edges joining vertices with low battery level.

For the second step, we consider the neighbours of each vertex from the list $M$ and store them creating the list $N$. Then, we compute the shortest paths to carry out the

contractions as we did in the BatteryContract algorithmic method. Additionally, we obtain the batteries of the vertices in $N$ and look for the highest value in order to apply the contraction operation to the edge adjacent to that vertex. Then, $R$ is defined as the edge from the vertex added in $M$ to the neighbour vertex with the highest battery percentage and the contraction is done.

The next step in the procedure is checking if the remaining vertices of the graph after the contraction are the correct ones or if a relabeling is necessary. This is due to the fact that after contracting an edge, only the label of one of the adjacent vertices to that edge is retained. Therefore, in this step, we ensure that each vertex's label and its associated attribute (battery level) are properly updated according to the initial conditions. Later, the edge weight is updated and, in case that the edge did not exist, is created with its corresponding weight.

Now the procedure continues reducing the battery level using the discharge rate that we mentioned before and the attributes `battery` are updated. This part is applied to each vertex in the graph and, finally, the outputs are returned.

```
> batterywithbdr:= proc(G, Rvert, Cvert)
> local vertices, bdr, v, battery, VertIndex,
newbattery, oldbattery, dist,
sp, sum, shortP, Rv, Cv, C, H, M, N, h, i, j,
k, P, l, m, verts, Q, R;
> Cv:= Cvert;
> Rv:= Rvert;
> M:= [];
> H:= G;
> vertices:= Vertices(H);
> for v in vertices do
> battery:=GetVertexAttribute(H,v,"battery");
> if battery < 33 then
> M:= [op(M), v];
> fi; od;
> print("Battery <33%", M);
> if 1 < nops(M) then
> for i to nops(M) - 1 do
> for j from i + 1 to nops(M) do
> if evalb({M[i], M[j]} in Edges(H))=true
> then
> H:= DeleteEdge(H, {M[i], M[j]});
> fi; od; od; fi;
> if 0 < nops(M) then
> for m to nops(M) do
> shortP:= [];
> sp:= [];
> dist:= [];
> Rv:= [op(Rv), M[m]];
> N:= Neighbors(H, M[m]);
> 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:= {M[m], N[Q]};
> C:= [N[Q], GetVertexAttribute(H, N[Q],
"battery")];
> Cv:= [op(Cv), C];
> H:= foldl(Contract, H, R);
> verts:= Vertices(H);
> if M[m] in verts then
> for i to nops(verts) do
> if verts[i] = M[m] 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; od; fi;
> 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:
```

Finally, the procedure `batterywithbdr` is executed inside a while loop with the condition that the number of vertices is bigger than one. 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 level of battery. In each iteration the graph is drawn to see the evolution of the contractions.

```
> vert:= Vertices(G);
> DrawGraph(G);
> steps:= nops(vert);
> numvertices:= nops(vert);
> RemovedVertices:= [];
> ContractedVertices:= [];
> while 1 < numvertices do
> G, battery, RemovedVertices, ContractedVertices:
= batterywithbdr(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;
```

### *5.4. HamiltonianGraph Algorithm*

This algorithm examines the hamiltonicity of a given Delaunay Graph $G$ using Lemma 4.1 and Propositions 4.1 and 4.2. It first counts the number of internal triangles in $G$ and then determines whether $G$ is Hamiltonian, semi-Hamiltonian,

or neither. The subroutine `numberK3` performs the triangle count by identifying internal triangles in the graph. Using vertex coordinates, it computes the convex hull and its edges, iterates through all three-vertex combinations, and checks for triangle formation. If a triangle has no edges in the convex hull, it is classified as internal and included in the count. The procedure returns the total number of internal triangles.

```
> numberK3:= proc(G, LISTA)
> local PTS, hull, hullEdges, allEdges, edge,
triangles, internalTriangles,
> triangle, isInternal;
> PTS:= [seq(LISTA[i][1], i = 1.. nops(LISTA))];
> hull:= ConvexHull(PTS);
> hullEdges:= [];
> for i to nops(hull) - 1 do
> hullEdges:= [op(hullEdges),
{hull[i + 1], hull[i]}];
> od;
> hullEdges:= [op(hullEdges),
{hull[-1], hull[1]}];
> allEdges:= Edges(G);
> triangles:= [];
> for i to NumberOfVertices(G) - 2 do
> for j from i + 1 to
NumberOfVertices(G) - 1 do
> for k from j + 1 to
NumberOfVertices(G) do
> if HasEdge(G, {i, j}) and HasEdge(G, {j, k})
and HasEdge(G, {i, k}) then
> triangles:= [op(triangles), {i, j, k}];
> fi; od; od; od;
> internalTriangles:= 0;
> for triangle in triangles do
> isInternal:= true;
> for edge in hullEdges do
> if {op(edge)} subset {op(triangle)} then
> isInternal:= false;
> break;
> fi; od;
> if isInternal then
internalTriangles:= internalTriangles + 1;
> fi; od;
> return internalTriangles;
> end proc;
```

The second step is carried out by the main procedure called `hamiltonianGraph`. It determines whether a graph is Hamiltonian, semi-Hamiltonian, or neither. It first checks the battery levels of all vertices, ensuring that each vertex satisfies a minimum threshold of 33%.

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

If this condition is met, the procedure uses `numberK3` to calculate the number of internal triangles in the graph. If there are no internal triangles, the graph satisfies Proposition 4.1, is classified as Hamiltonian, and a Hamiltonian cycle is computed. If there are fewer than three internal triangles, the graph is also Hamiltonian since it satisfies Proposition 4.2. If three or more internal triangles exist, the procedure attempts to find a semi-Hamiltonian path. If successful, the graph is classified as semi-Hamiltonian; otherwise, it is determined to be neither.

```
> hamiltonianGraph:= proc(H, LISTA)
> local vertices, n, verts, path, numK3;
> vertices:= Vertices(H);
> verts:= 0;
> for n in vertices do
> if 33 <= GetVertexAttribute(H,n,"battery") then
> verts:= verts + 1;
> fi; od;
> if verts = nops(vertices) then
> numK3:= numberK3(H, LISTA);
> if numK3 = 0 then
> print("The graph is Hamiltonian.");
> print("IT(DT(V)) is empty. The graph satisfies
Proposition 1.");
> print("The Hamiltonian cycle is",
FindHamiltonianCycle(H));
> elif numK3 < 3 then
> print("The graph is Hamiltonian.");
> print("The graph satisfies Proposition 2
with fewer than 3 internal triangles.");
> print("Number of internal triangles:",numK3);
> print("The Hamiltonian cycle is",
FindHamiltonianCycle(H));
> else path:= FindHamiltonianPath(H);
> if path <> NULL then
> print("The graph is semihamiltonian.");
> print("Number of internal triangles:",
numK3);
> print("Semihamiltonian Path:", path);
> else print("The graph isn't Hamiltonian
or semihamiltonian."); > fi; fi;
> else print("There are vertices with
low battery.");
> fi;
> end proc;
```

# 6. Case Study: Monitorization of a Sugar Cane Field

***Figure 9.*** *Deployed node in a sugar cane field.*

The proposed methods can be used to manage the battery level 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 house of greenhouses. This is the case of the situation presented here, in which the monitorization of two soil variables – moisture and temperature – of a sugar cane field was needed to control the irrigation.

The monitorization network is built with a set of devices, called nodes, which have different capabilities: 1) measurement of soil moisture and temperature; computation, to execute some local filters and routines, and storing; and communication of the processed data using LoRaWAN protocol. The heart of each node is a Lopy4 board from Pycom, whose core is a ESP32 processor. A photography of the node is shown in Figure 9.

A set of 12 nodes were deployed in a $100 \times 30$ 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 10, where the $x$-axis is aligned with the direction of the sugarcane planks.

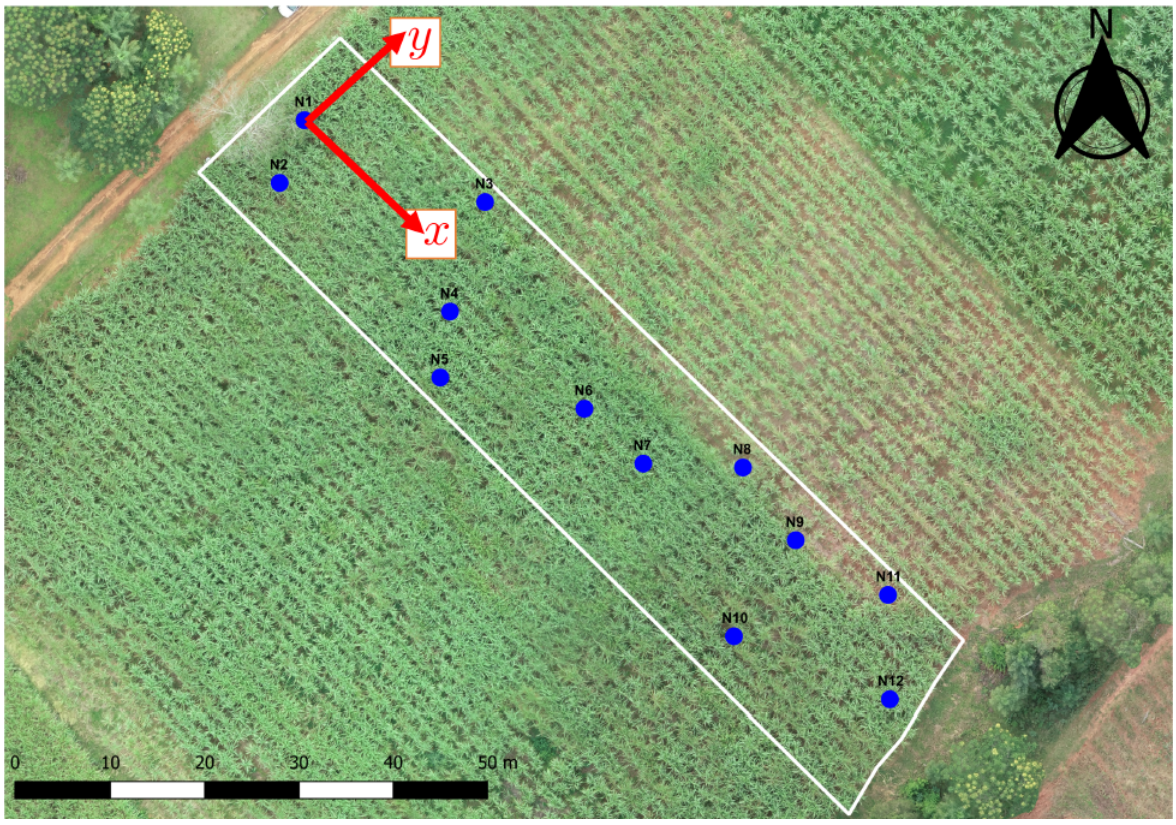


***Figure 10.*** *Aerial photography of the sugar cane field with the nodes marked with blue dots.*

Now, we consider the 12 nodes from Figure 10 and define the variable `PTS` containing the euclidean coordinates of each node.

```
PTS:=[[5,13],[10,6],[25,28],[30,20],[35,3],[45,11],
[55,16],[60,29],[75,22],[80,5],[85,28],[95,12]];
```

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

```
LIST:=[[[5,13],96],[[10,6],16],[[25,30],59],
[[30,20],9],[[35,3],38],[[45,11],50],
[[55,16],14],[[60,30],8],[[75,22],83],
[[80,5],12],[[85,30],93],[[95,12],52]];
```

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

$$\begin{bmatrix} 0 & 8.60 & 25.00 & 25.96 & 31.62 & 40.05 & \dots \\ 8.60 & 0 & 26.63 & 24.41 & 25.18 & 35.36 & \dots \\ 25.00 & 26.63 & 0 & 9.43 & 26.93 & 26.25 & \dots \\ 25.96 & 24.41 & 9.43 & 0 & 17.72 & 17.72 & \dots \\ 31.62 & 25.18 & 26.93 & 17.72 & 0 & 12.81 & \dots \\ 40.05 & 35.36 & 26.25 & 17.49 & 12.81 & 0 & \dots \\ 50.09 & 46.10 & 32.31 & 25.32 & 23.85 & 11.18 & \dots \\ 57.28 & 55.04 & 35.01 & 31.32 & 36.07 & 23.43 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{bmatrix}$$

In Figures 11 and 12, we show the Voronoi diagram and weighted Delaunay Graph associated to the nodes.

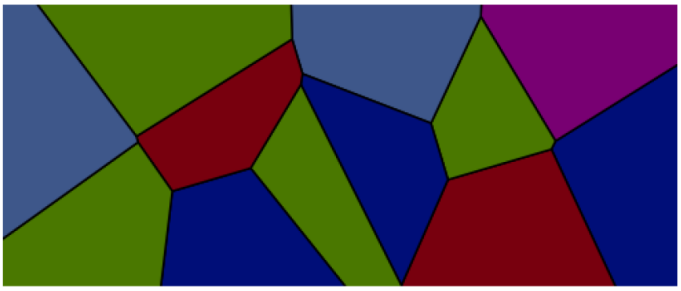

***Figure 11.*** *Voronoi diagram.*

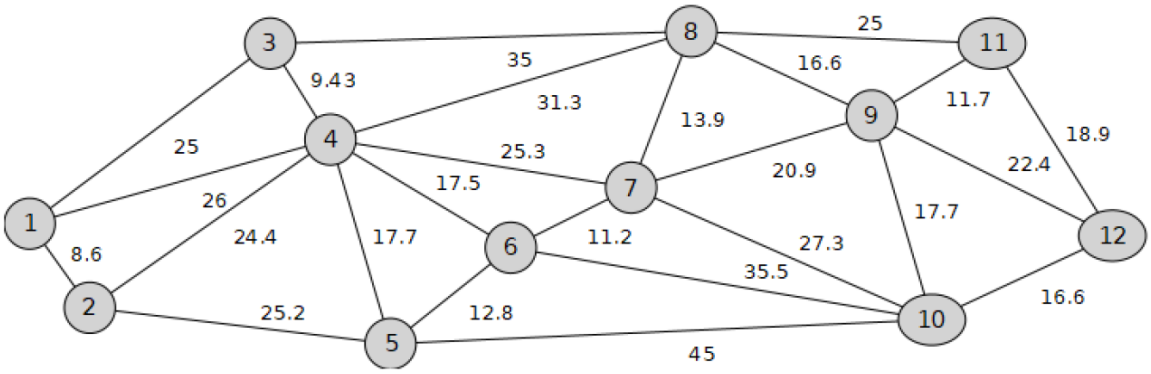


***Figure 12.*** *Weighted Delaunay Graph.*

Now, we show the outputs of the main routines of the BatteryContract algorithm. By executing the subprocedure `batteryhighlight`, we obtain the graph of Figure 13.

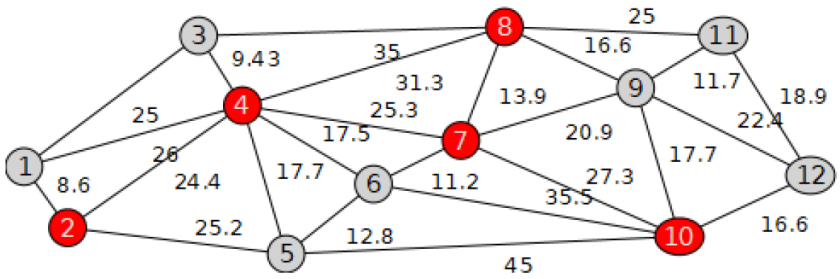


***Figure 13.*** *Output of the subprocedure batteryhighlight.*

From the main procedure `batterycontract`, we obtain that the list of the contractions to be done is $\{1,2\},\{3,4\},\{6,7\},\{8,9\},\{10,12\}$. This means that the edge connecting nodes 1 and 2 must be contracted, also the edge connecting nodes 3 and 4 and so on. We also obtain the minimum spanning tree of Figure 14.

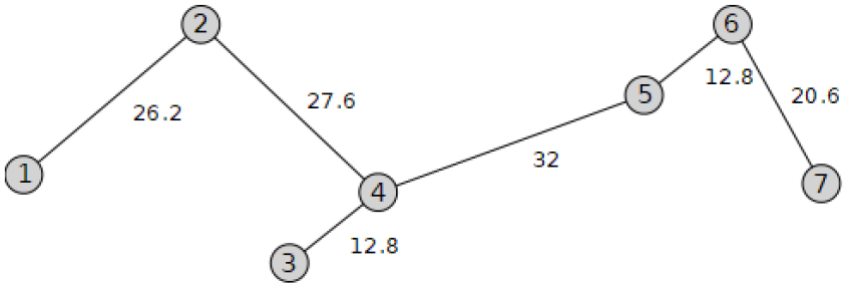


***Figure 14.*** *Output of the subprocedure batterycontract.*

Next, we execute the procedure `increasingbatord` from the BatteryOrder algorithm obtaining that the list of increasing ordered nodes according to the battery level is

$$[8, 4, 10, 7, 2, 5, 6, 12, 3, 9, 11, 1]$$

This procedure also returns the following list containing the optimum path to visit all the nodes in the previous order and the distance of each of those paths.

$$[[[8, 3, 4], 44.44], [[4, 6, 10], 53.00], [[10, 9, 7], 38.60],$$

$$[[7, 6, 5, 2], 49.17], [[2, 5, 10, 12], 86.77],$$

$$[[12, 11], 18.87], [[11, 8, 3, 1], 85.03]]$$

For example, the element $[[8, 3, 4], 44.44]$ means that to travel from node $8$ to node $4$ we must go through the third node and the total distance is $44.44$. Finally, we also obtain the total distance of the complete path, which is $375.88$.

We continue by executing the attribute `battery` and the main procedure `batterywithbdr` from the DischargeContraction algorithm. We first obtain the initial battery of all the nodes, the list of vertices and the initial graph (see Figure 12).

$$InitialBattery : [[1, 96], [2, 16], [3, 59], [4, 9], [5, 38], [6, 50],$$

$$[7, 14], [8, 8], [9, 83], [10, 12], [11, 93], [12, 52]]$$

$$vert := [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$$

After that, we show the evolution of the graph from the first iteration in Figure 15.

$$numvertices := 12 \qquad Battery < 33\%, [2, 4, 7, 8, 10]$$

$$[[1, 93], [3, 54], [5, 33], [6, 45], [9, 78], [11, 90], [12, 48]],$$

$$[2, 4, 7, 8, 10], [[1, 96], [1, 96], [9, 83], [11, 93], [9, 83]]$$

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

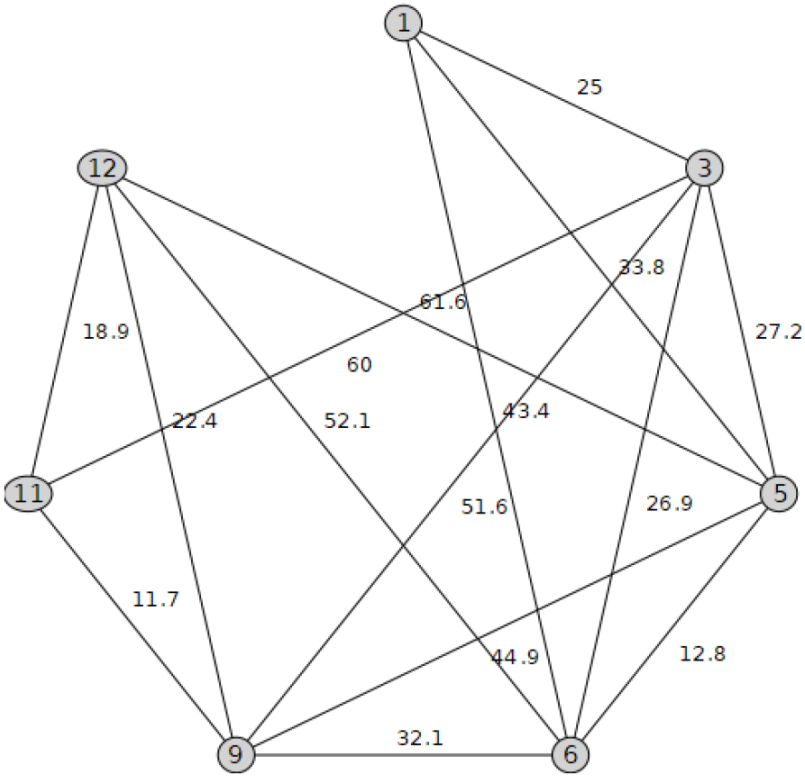


***Figure 15.*** *First iteration of the procedure batterywithbdr.*

Next, we run the DegreeRouteBattery algorithm. We first obtain the initial situation as in the DischargeContraction algorithm. After that, we show the evolution of the graph from one iterations of the procedure `decreasingbatord` in Figure 16.

$$numvertices := 12\ Route, [4, 10, 9, 8, 7, 6, 5, 12, 11, 3, 2, 1]$$

$$[[1, 90], [3, 53], [5, 32], [6, 44], [9, 77], [11, 87], [12, 46]],$$

$$[4, 10, 8, 7, 2], [[1, 96], [9, 83], [1, 96], [1, 96], [1, 96]]$$

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

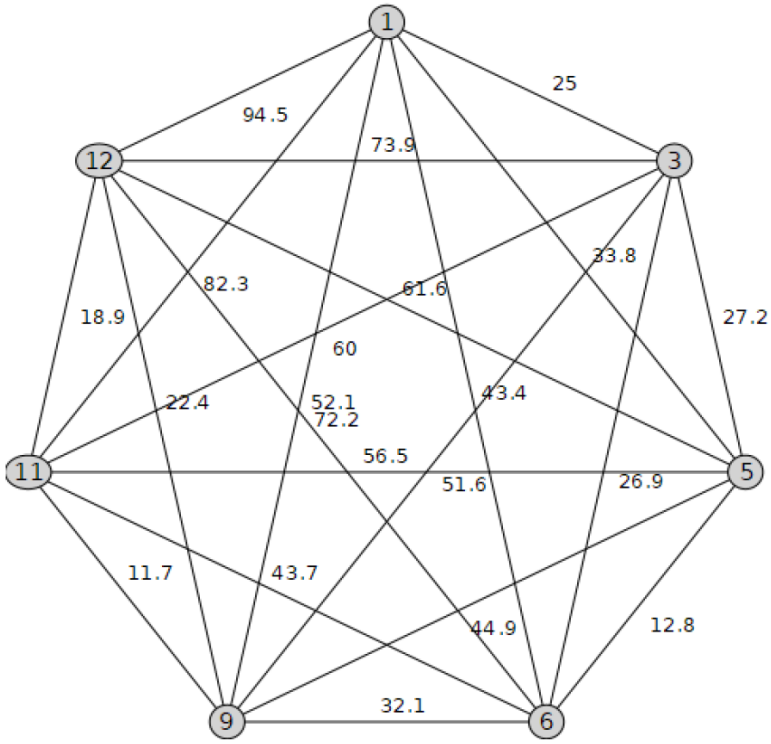


***Figure 16.*** *First iteration of procedure decreasingbatord.*

Finally, we execute the procedures `numberK3` and `hamiltonianGraph`. First, we will assume that all the nodes have a good battery level (more than 33%) and therefore, no contraction is needed and it makes sense to study the hamiltonicity of the Delaunay Graph $G$ trying to find Hamiltonian paths. From the procedure `numberK3` we obtain

$$internalTriangles = 8$$

The main routine `hamiltonianGraph` returns the followin output:

The graph is Hamiltonian

The Hamiltonian cycle is $[1, 2, 4, 5, 6, 7, 9, 10, 12, 11, 8, 3, 1]$

## 7. Conclusions

We believe that the tools and algorithms considered in this paper may be useful and helpful for understanding the link between Graph Theory and sensor networks. In addition, this link may provide new methods to deal with various challenges in network optimization, routing, clustering, localization and coverage. We would also like to point out the importance of further research in this field to unlock the full potential of graph-based approaches in sensor networks.

## Abbreviations

| | |
|---|---|
| IoT | Internet of Things |
| MST | Minimum Spanning Tree |
| LoRaWAN | Long Range Wide Area Network |
| ESP32 | Espressif Systems 32-bit Microcontroller |

## ORCID

0000-0003-0913-6417 (Manuel Ceballos)
0000-0002-1930-1909 (Luis Orihuela)

## Acknowledgments

This work has been partially supported by FQM-326 and PID2020-117800GB-I00. The work of L. Orihuela is partially funded with the Ramón y Cajal programme (RYC2021-032919-I).

## Conflicts of Interest

The authors declare no conflicts of interest.

## References

[1] Árego, 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. https://doi.org/10.1007/s00454-008-9099-1

[2] Akyildiz, I. F., Su, W., Sankarasubramaniam, Y. and Cayirci, E. (2002). A survey on sensor networks. IEEE Communications Magazine, 40(8), 102–114. https://doi.org/10.1109/MCOM.2002.1024422

[3] 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. https://doi.org/10.6688/JISE.202003_36(2).0011

[4] Franz Aurenhammer, F., Klein, R. and Lee, D. (2013). Voronoi Diagrams And Delaunay Triangulations. World Scientific Publishing Company. https://doi.org/10.1142/8685

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

[6] 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. https://doi.org/10.1090/conm/065/891251

[7] Harary, F. (1994). Graph Theory. Reading, MA: Addison-Wesley. https://doi.org/10.1201/9780429493768

[8] 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

[9] 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.

[10] 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, https://doi.org/10.1109/WCNC.2012.6214026

[11] 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. https://doi.org/10.1186/s13673-018-0153-6

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

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

[14] Shao, C. Wireless sensor network target localization algorithm based on two- and three-dimensional Delaunay partitions, (2021). Journal of Sensors, 4047684. https://doi.org/10.1155/2021/4047684

[15] Wagner, K. (1937), Über eine Eigenschaft der ebenen Komplexe. Math. Ann., 114, 570–590, https://doi.org/10.1007/BF01594196, S2CID 123534907

[16] 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. https://doi.org/10.1002/dac.1087

[17] 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