LPCN: Least Polar-angle Connected Node Algorithm to Find a Polygon Hull in a Connected Euclidean Graph

Farid Lalem, Ahcène Bounceur, Madani Bezoui, Saoudi Massinissa, Reinhardt Euler, M. Tahar Kechadi, Marc Sevaux

LPCN: Least Polar-angle Connected Node Algorithm to Find a Polygon Hull in a Connected Euclidean Graph

Farid Lalem*, Ahcène Bounceur*, Madani Bezoui§, Massinissa Saoudi*†, Reinhardt Euler*
Tahar Kechadi*† and Marc Sevaux*‡

* Université de Bretagne Occidentale, Lab-STICC UMR CNRS 6285
‡ Université de Bretagne Sud, Lab-STICC UMR CNRS 6285
§University of Boumerdes, Department of Mathematics
†UCD, University College Dublin

Email: Ahcene.Bounceur@univ-brest.fr

***Abstract*—Finding the polygon hull in a connected Euclidean graph can be considered as the problem of finding the convex hull with the exception that at any iteration a vertex can be chosen only if it is connected to the vertex chosen at the previous iteration. One of the methods that can be used for this kind of problems is Jarvis' algorithm which allows to find the convex hull and which must be adapted because it does not take into account the connections of the nodes. In this paper, we propose a new algorithm that chooses for a current node and among its neighbors in the graph the nearest polar angle node with respect to the node found in the previous iteration. Its complexity is $O(gh^2)$, where g is the maximum degree of the graph and h the number of the nodes on the hull. For ease of presentation, we first identify some specific graph-structures whose presence may lead a basic version of the algorithm to fail, and we then show how to modify that version to obtain a procedure of the given complexity. Finally, we present some practical applications that can be resolved using the proposed algorithm.**

***Index Terms*—Connected Euclidean graph, Planar graph, Boundary nodes, Polygon hull, Computational geometry, Shape reconstruction.**

I. Introduction

Many real life applications can be modelled as a connected Euclidean graph, like Wireless Sensor Networks (WSNs), pixels of an image, cities of a country, personal computers, etc. Finding the polygon hull of this kind of graphs can help to resolve real problematics like for example:

- Monitoring sensitive sites: [23] uses an algorithm detecting faulty nodes during the monitoring process,
- Anomaly detection: [22] uses the polygon hull of the copula of the data to separate the faulty data from the correct ones,
- Region of interest localization: [24] uses an algorithm based on detecting surface's extremities,
- Shape recognition: [19] presents many applications in the area of image processing,
- Medical and biological images: [37] uses a polygon hull to calculate the number of red cells in a swear images, and the work of [12] allows to remove the geometric distortions between referenced and sensed images,
- Computer vision: in [25] many applications in visual feature detection have been presented,
- Etc.

In this paper, we propose a new algorithm allowing to find the boundary vertices of a connected Euclidean graph, where we try to find a set of vertices allowing to represent the geometric shape of the graph in the form of a polygon hull, i.e., a simple polygon formed by edges of the graph such that all vertices of the graph are either on the polygon or surrounded by it. More precisely, we are looking for a closed cycle of minimum length in the graph such that all vertices are either on or surrounded by that cycle. This problem can be formulated as follows. We consider an undirected graph $G = (V, E)$, where $V = \{v_0, v_1, ..., v_{n-1}\}$ is the set of vertices of the graph and E its set of edges. We assume that all the edges of G are represented as straight lines. A Polygon Hull may be given by a set of vertices $\mathbb{B}_V$ and a set of edges $\mathbb{B}_E$ as follows:

$$\mathbb{B}_V = \{v_0, v_1, ..., v_{i-1}, v_i, v_{i+1}, ..., v_h\} \subseteq V,$$

$$\mathbb{B}_E = \{\{v_0, v_1\}, \{v_1, v_2\}, \cdots, \{v_{h-1}, v_h\}\} \subseteq E$$

such that,
a) $v_0 = v_h$ (i.e., v_0 is the same as v_h),
b) the vertices in $V \backslash \mathbb{B}_V$ represent the set $\mathrm{INT}(\mathbb{B}_V)$,
i.e., the set of vertices lying inside the cycle.

To solve this problem we propose an algorithm called *Least Polar-angle Connected Node (LPCN)* which allows to find a polygon hull (or a boundary) of a given Euclidean graph. The main advantage of this algorithm is the way it defines the boundary. Its main idea is given as follows. In each iteration i the next boundary vertex v_{i+1} is determined by the vertex that has the minimum angle $\varphi_{min}(v_{i-1}, v_i, v_{i+1})$ formed by the edges $\{v_i, v_{i-1}\}$ and $\{v_i, v_{i+1}\}$ (cf. Figure 1), where v_i is

the boundary vertex selected in the current iteration i and v_{i-1} is the boundary vertex found in the previous iteration $i-1$.

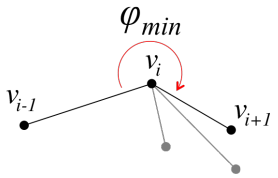

Fig. 1. Angle formed by three vertices of a graph.

Our objective is to determine a simple closed polygon, but it turns out that it may be sufficient to obtain a walk which is not closed. Our algorithm works under the assumption that some particular subgraphs do not exist, for example, boat graphs (as represented in Figure 5), anchor graphs (given in Figure 10), or vertices of degree 1. These graphs require a special treatment, and non-closed walks could be sufficient in case we just want to visit their vertices. We will show later how to overcome these limitations in the presence of such structures. Our first algorithm determines in each iteration the next boundary neighbor of the currently found boundary vertex. Its complexity is $O(gh^2)$, where g is the degree of the graph and h the number of the vertices on the hull.

The structure of this paper is as follows. In Section II, we present the related work. In Section III, we review Jarvis' algorithm [18] and the basic version of the proposed LPCN algorithm. Section IV will present the different limitations caused by sub-graphs that lead our first algorithm to fail and the ways to overcome them. This section also includes the final version of the proposed LPCN algorithm. The validation of the algorithm is done in Section V. Simulation results and complexity are presented in Section VI. Some applications will be presented in Section VII. Finally, Section VIII concludes the paper.

II. Related work

In the following, we review some useful algorithms allowing to find either a convex or a polygon (concave) hull for a given set of points in the plane. Sometimes, a convex hull algorithm can be modified to determine a polygon hull as is the case in this paper, where we propose a polygon hull algorithm based on Jarvis' convex hull algorithm [18] which also uses the angle based search.

A. Convex Hull

1) Graham's Algorithm [17]: This algorithm starts from an extreme point called pivot, which can be the point with minimum x-coordinate. The remaining points will be sorted in the order of increasing angles with respect to the pivot. The algorithm stops with a star-shaped polygon. Finally, the hull will be built by marching around the star-shaped polygon, adding edges when we make a left turn, back-tracking when we make a right turn. The complexity of this algorithm is $O(n \log n)$, where n is the number of the given points.

2) Jarvis' Algorithm [18]: This algorithm starts with the point having minimum x-coordinate, for example. Then repeatedly, it adds the point having the least polar angle with respect to the previous point. This algorithm is detailed in Section III-A since our proposed algorithm is based on it. The complexity of Jarvis' algorithm is $O(nh)$ where h is the number of points on the hull.

3) Quick-hull [6]: The Quick-hull algorithm starts with computing the points with minimum and maximum x-coordinates and minimum and maximum y-coordinates. Clearly, these points will be on the hull. Connecting these four points will lead to a convex quadrilateral. All the points within this quadrilateral can be eliminated from further consideration. The remaining points are classified into four remaining corner triangles and those lying inside these triangles are discarded. The same procedure will be repeated for the newly obtained triangles until there is no point outside the triangles. The complexity of this algorithm is $O(n^2)$.

4) Incremental algorithm [20]: This algorithm can be used in a multidimensional space. It operates by inserting points one at a time and by incrementally updating the hull. If the new point is inside the hull then there is nothing to do. Otherwise, all the edges that the new point can see must be deleted. Then, the new point will be connected to its two neighbor points and the hull updated. By repeating this process for the remaining points outside the current hull, a convex hull will finally be constructed. The complexity of this algorithm is $O(n^{d+1/2})$ where d is the dimension of the considered space.

5) TORCH algorithm [16]: The Total ORder heuristic-based Convex Hull (TORCH) algorithm is a sorting-based algorithm that starts by sorting all points in the x-direction. Then, it determines the four extremal points which are left-most, rightmost, bottommost and topmost points in order to find the four lateral hulls between turning points. As a result of such a sorting procedure, we immediately obtain an approximate convex hull that contains all the extreme vertices of the real convex hull together with a few concave vertices. Then, by eliminating these concave vertices from the approximate convex hull using the geometric operation counterclockwise (CCW), just as in Andrew's Monotone method [5], we get the final convex hull. The TORCH algorithm has $O(n \log n)$ time complexity.

6) Xing et al.'s algorithm [40]: This algorithm aims to compute the convex hull of the planar point set. The algorithm starts by constructing an initial convex polygon (ICP) by computing eight extreme points, and measures the width and the length of the ICP. Then, it maps the point set into a new space using an affine transformation where most of the new points stay inside the new initial convex polygon (NICP). Next, it removes the inner points that are near the boundary of the NICP and applies Quick-hull to the remaining points. Then, it will map the vertices of the found convex hull to their original coordinates in order to obtain the final convex hull. This algorithm has $O(n + n \log n)$ time complexity.

7) Mei's algorithm [27] : In this paper, a GPU-accelerated convex hull algorithm is presented. The algorithm starts by

eliminating the points that are inside a quadrilateral formed by the four extreme points. Then, the remaining points will be distributed into four sub-regions. The points that are situated in the same region will be sorted in parallel according to their coordinates. Then, a novel Sorting-based Preprocessing Approach (SPA) is performed to eliminate the inner points. After that, for each sub-region, it forms a simple chain with the remaining points. By connecting the four obtained chains in CCW, it forms a simple polygon. Finally, Melkman's algorithm [28] is applied to calculate the convex hull of the formed simple polygon. This approach has $O(n \log n)$ time complexity.

8) Ruano et al.'s algorithm [35]: This is a randomized approximation algorithm for high-dimensional datasets. The algorithm starts by scaling each dimension to the range $[-1, 1]$, and then identifies the minimum and the maximum samples with respect to each dimension. These samples are considered as vertices of the initial convex hull. Then, it generates a population of k facets based on the current vertices of the convex hull and identifies the points furthest from each facet in the current population as the new vertices of the convex hull. This operation is repeated iteratively until there are no newly found vertices. Finally, we obtain the convex hull. The time complexity of this algorithm is not given, but it depends on the number of samples and features, the population size (input parameter k), the number of iterations, the number of vertices of the convex hull, and on the distribution of the samples in the dataset.

9) S-CH algorithm [38]: The Smart Convex Hull (S-CH) algorithm starts by applying a space subdivision method in order to eliminate a maximum of the initial points. Then, it determines the convex hull over the remaining points by applying, for instance, any standard convex hull algorithm. The time complexity of this algorithm is $O(n \log n)$.

B. Polygon or Concave Hull

1) Split and Merge [14]: This algorithm starts with a convex hull of points and obtains the final boundary in two steps: splitting followed by merging. During splitting, one or more sides of the points of the convex hull will be deleted and new sides are added to take care of the inherent concavity. To obtain a smooth polygonal boundary, two or more sides are merged into a single one. The complexity of this algorithm is $O(nh)$.

2) Shape of a set of points [10]: In this work the authors introduce the concept of an α-*shape* which represents an *external shape* (convex or not) of a set of points. They describe an algorithm that allows to find the value of α. For a finite set P of points in the plane, the $\alpha - hull$ for $\alpha \neq 0$ is the intersection of all closed disks of radius $1/\alpha$ containing the points of P (where for negative values of α a closed disk of radius A/α is interpreted as the complement of an open disk of radius $-1/\alpha$). When α approaches 0, the $\alpha - hull$ approaches the ordinary convex hull. Therefore, the $0 - hull$ is stipulated to be the convex hull. The $\alpha - shape$ is a straight-line graph (usually a polygon) derived in a straightforward manner from the $\alpha - hull$. When $\alpha = 0$, it is the convex hull. For large negative values of α, it represents the initial set of points P. The complexity of this algorithm is $O(n \log n)$.

3) Perceptual boundary extraction [8]: This approach is based on a new definition called the $s - shape$, which is obtained by partitioning the plane into a lattice of square cells of side-length s. The $s - shape$ is simply the union of lattice cells containing all the points of P. To obtain a polygonal boundary, another $r - shape$ is defined, where r is obtained from s. To find the $r - shape$, the union of all disks of radius r centered around the points of P is taken. For points $p, q \in P$, the edge $\{p, q\}$ is selected if and only if the boundaries of the disks centered on p and q intersect in a point which lies on the boundary of the union of all the disks. The $r - shape$ of P is then the union of the selected edges. The complexity of this algorithm is $O(n)$.

4) k-nearest neighbor [30]: This algorithm is used to compute a concave hull of a set of points in the plane. It assumes that the currently chosen point is connected to the k nearest points. Then it selects the point that has the least polar angle with respect to the current point, as in Jarvis' algorithm (Section II-A2). A concave hull is not obtained for any point set, and the value of k must be adapted to each case. Its complexity is $O(kh + gh^2)$. Our proposed algorithm uses the same concept except that the graph is connected. However, in our case, the number k is known and may be different for each point. Also, the k nearest neighbors are not necessarily connected.

5) Concaveness Measure [32]: This algorithm aims to identify an n-dimensional concave hull. It is composed of four steps. First, a set of convex hull edges is selected and a threshold value N is chosen. Then, the inner points nearest to the convex hull edge are identified. The shortest distance, called *decision distance*, between these nearest inner points and the edge's points is identified. Third, a decision of searching or not is made by comparing N with the decision distance. If $(length\ of\ edge)/(decision\ distance) > N$, then the search process is executed. Finally, the second and the third steps are repeated until there is no inner point to find. The complexity of this algorithm is $O(n \log n + rn)$ where r depends on the dimension d of the considered space. For example, for a 3-dimensional space, r is equal to $d/2$.

6) Braune et al.'s algorithm [7]: This algorithm finds clusters in a given dataset based on the notion of concave hull. The main idea is to start from the convex hull of the entire dataset and to iteratively shrink the convex hull by replacing edges that are too long with new edges that fit the data more accurately, and then, to recursively split the convex hull path into two separate closed paths as soon as it converges into one point. The time complexity of this algorithm is $O(n \log k)$, where k is the number of points on the convex hull and n the total number of points.

7) Gheibi et al.'s algorithm [15]: A shape reconstruction algorithm which finds a concave hull is presented. The algorithm starts from the convex hull of input samples and concaves it gradually to achieve the real polygonal output. Thereby, in

each step, each selected edge will be replaced with two new constructed edges in such a way that the shape remains a polygon. The selection criterion of these new edges is based on a combination of the following visual perception factors: Voronoi diagram, closeness of points, length of edges, etc. This process is repeated until the stopping condition is satisfied. The time complexity of this algorithm is $O(n \log n)$ time.

8) Ec-shape algorithm [29]: This algorithm aims to find a concave hull which best approximates the geometric shape of a given set of points in the plane. It starts by constructing the Delaunay Triangulation graph G [39] of these points. Then, it constructs a Priority Queue (PQ) of the external edges (EEs) of G in descending order of edge lengths.

Then it removes, repeatedly, the external edge (EE) of the external triangle (ET) from the head of (PQ) and from the current graph G, but only if it satisfies the defined circle constraint and that $G - EE$ (G without the edge EE) is regular. Once the extrernal edge EE is removed from G, the adjacent sides of the external triangle (ET) are added to (PQ) by maintaining the descending order of the edge lengths. This process is repeated until there is no possibility of removing any external edges from the graph G. The time complexity of this algorithm is $O(n \log n)$.

9) Gift Opening algorithm [34]: The idea of this algorithm is to find the convex hull using any known algorithm and then to transform it into a concave hull. To do this, the algorithm starts by adding all external edges to a list sorted by their lengths. The longest edge will be selected and removed from the list. Then, a new point will be selected so that it forms the smallest angle with the endpoints of the removed edge. After that, two new edges will be created from this new point and the endpoints of the removed edge. These two edges will be added to the list. The algorithm stops when no more edge in the list has a length larger than a predefined threshold. The time complexity of this algorithm is $O(n)$.

10) RGH algorithm [21]: The Regularized Geometric Hull (RGH) algorithm is used mainly for biomedical image segmentation. It transforms the points of a dataset into a set of triangles. Each triangle having its maximal edge length greater than a given parameter ζ will be removed. The value of ζ regularizes the convexity or concavity of the geometric hull. The time complexity of this procedure is $O(n^3)$ in the usual case. However, in case that the triangulation is based on Delaunay's algorithm, the time complexity will be $O(n \log n)$.

Further literature can be found in [9], [11], [13], [31], [33]. Table I summarizes all proposed approaches in the related work section regarding convex or polygon hull determination algorithms, also, in terms of complexity, optimality and dimensionality, where n is the total number of points, k a fixed number of the considered nearest neighbors of a point, g the maximum degree of the graph, h the number of points on the convex hull, d the dimension of the considered space, and where r is a number that depends on the dimension d of the considered space, which is equal to $\frac{d}{2}$ for a 3-dimensional space.

III. LPCN ALGORITHM

In this section, we will present the first version of our LPCN (Least Polar-angle Connected Node) algorithm. We will show how to modify Jarvis' algorithm, initially described to find a convex hull, for the purpose to find the polygon hull of a set of nodes of a connected graph.

A. Jarvis' Algorithm

Jarvis' algorithm [18] determines the convex hull of a finite set of points. It cannot be used directly in our work since we are searching for a polygon hull. However, we can modify it by considering in each iteration only nodes that are connected to the current one instead of taking any node.

We recall that Jarvis' algorithm selects in each iteration the node that has the minimum angle with the left horizontal segment passing through the current node. This can work only on one side of a set of finite nodes. In [3], an improvement of Jarvis' algorithm is proposed by dividing the set of nodes into two sets and by changing the orientation of the angles in each set. Another improvement of Jarvis' algorithm uses the polar angle $\varphi(P_p, P_c, P_j)$ formed by the next node P_j, the current node P_c and the one found in the previous iteration P_p. Hence, Jarvis' algorithm can be described as Algorithm 1 as follows:

Algorithm 1 Jarvis' algorithm

```
1:  procedure JARVIS(V)
2:      P_c ← point having the minimum x-coordinate
3:      P_first ← P_c
4:      P_p ← fictive point situated on the left of P_c
5:      B ← {P_c}
6:      repeat
7:          P_k = argmin_{P_j ∈ V} {φ(P_p, P_c, P_j)}
8:          B ← B ∪ {P_k}
9:          P_p ← P_c
10:         P_c ← P_k
11:     until (P_k = P_first)
12:     return B
13: end procedure
```

Let V be a set of n points and h be the number of points defining the convex hull of V. The algorithm finds the points P_0, P_1, ..., P_{h-1} of the convex hull, ordered one at a time, where P_0 is the point with the minimum x-coordinate. Given the point P_i, we wish to find the point P_{i+1}, consecutive to P_i on the hull, for each $1 \leq i \leq (n-2)$. This point is such that the edge $\{P_i, P_{i+1}\}$ has the least polar angle with the edge $\{P_i, P_{i-1}\}$. The algorithm stops once we come back to point P_0. Figure 2 shows an example illustrating this algorithm. We consider a set of 8 points (cf. Figure 2(a)) and we start from point A. The least polar angle point with respect to A is point C (cf. Figure 2(b)). Next, we define the least polar angle point with respect to C and A which is point E (cf. Figure 2(c)). In the same way, we define the least polar angle point with respect to E and C which is point H. The last point defined

TABLE I
COMPARISON WITH EXISTING ALGORITHMS.

Algorithm	Convex Hull	Polygon Hull	Complexity	Optimality	Dimension	Observations
Graham [17]	✓		$O(n \log n)$	-	$2D$	
Jarvis [18]	✓		$O(nh)$	-	$2D$	
Quick-hull [6]	✓		$O(n^2)$	-	$2D$	
Incremental [20]	✓		$O(n^{(d+1)/2})$	-	Multidimensional	
TORCH [16]	✓		$O(n \log n)$	-	$2D$	
NICP [40]	✓		$O(n + n \log n)$	-	$2D$	Requires the Quick-hull algorithm
Mei [27]	✓		$O(n \log n)$	-	$2D$	GPU-accelerated convex hull algorithm
Ruano et al. [35]	✓		[not given]	-	Multidimensional	Complexity depends on many parameters
S-CH [38]	✓		$O(n \log n)$	-	$3D$	Requires a standard convex hull algorithm
Split and Merge [14]		✓	$O(nh)$	Not Optimal	$2D$	Starts from the convex hull of points
Alpha-Shape [10]		✓	$O(n \log n)$	Not Optimal	$2D$	Depends on a parameter α
PBE [8]		✓	$O(n)$	Not Optimal	$2D$	
KNN [30]		✓	$O(nh^2)$	Not Optimal	$2D$	
CM [32]		✓	$O(n \log n + rn)$	Not Optimal	Multidimensional	
Braune et al. [7]		✓	$O(n \log h)$	Not Optimal	$2D$	Starts from the convex hull of points
Gheibi et al. [15]		✓	$O(n \log n)$	Not Optimal	$2D$	Starts from the convex hull of points
EC-Shape [29]		✓	$O(n \log n)$	Not Optimal	$2D$	Requires Delaunay Triangulation (DT)
Gift Opening [34]		✓	$O(n)$	Not Optimal	$2D$	Starts from the convex hull of points
RGH [21]		✓	$O(n^3)$	Not Optimal	$2D$	
LPCN		✓	$O(gh^2)$	Optimal	$2D$	Best complexity

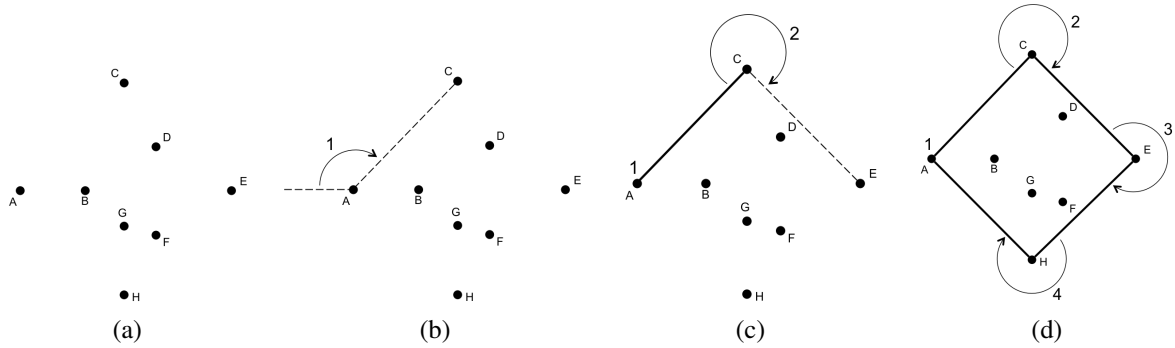

Fig. 2. Jarvis' algorithm for (a) point set in free plane, (b) iteration 1, (c) iteration 2, (d) last iteration.

after H is A which was the starting point for the first iteration. Hence, the convex hull is found (cf. Figure 2(d)).

B. LPCN1: the first version

Given a Euclidean connected graph $G = (V, E)$, the first version of our algorithm LPCN (Least Polar-angle Connected Node) differs from Jarvis' algorithm in the selection of that point (or node) which follows the current one: the next point can only be a neighbor in G of the current one. It only works either for planar graphs or for non-planar graphs without some problematic subgraphs that will be presented in Section IV. It starts from a node that belongs to the boundary and can be outlined as follows. Let V be the set of n nodes of G and h be the number of points of the final polygon hull of V. The algorithm first finds the points P_0, P_1, ..., P_{h-1} of the polygon hull, where P_0 is a point with the minimum x-coordinate. Given the point P_i, we wish to find the next consecutive point P_{i+1} on the hull, wichi is connected to P_i, for each $1 \leq i \leq k-1$, which is connected to P_i. This point P_{i+1} is the one that has the least polar angle with respect to P_i. When we reach the last point P_{h-1}, we have constructed a polygon hull of V. The algorithm stops as soon as we come back to the point P_0. It is presented as Algorithm 2, in which $N(P_c)$ represents the neighborhood of P_c in G.

Algorithm 2 LPCN1: the first version of the LPCN algorithm

procedure LPCN1(V, E)
  $P_c \leftarrow$ point having the minimum x-coordinate
  $P_{first} \leftarrow P_c$
  $P_p \leftarrow$ fictive point situated on the left of P_c
  $\mathbb{B}_V \leftarrow \{P_c\}$
  $\mathbb{B}_E \leftarrow \emptyset$
  repeat
    $P_v \leftarrow \underset{P_j \in N(P_c)}{\text{argmin}} \{\varphi(P_p, P_c, P_j)\}$
    $\mathbb{B}_V \leftarrow \mathbb{B}_V \cup \{P_v\}$; $\mathbb{B}_E \leftarrow \mathbb{B}_E \cup \{\{P_c, P_v\}\}$
    $P_p \leftarrow P_c$
    $P_c \leftarrow P_v$
  until ($P_v = P_{first}$)
  return $\mathbb{B}_V$, $\mathbb{B}_E$
end procedure

Figure 3 shows how the algorithm works. Let us consider the graph of Figure 3(a) where $V = \{A, B, C, D, E, F, G, H\}$. First, we choose the node with the minimum x-coordinate P_c. In this example, this node is A ($P_c = A$) which is also the first node of the boundary $\mathbb{B}_V = \{A\}$.

The algorithm stops when the next boundary node P_k is equal to the first boundary node P_{first}, in this example, node A (i.e., $P_{first} = A$). In the first iteration, P_c is always equal to P_{first}. To start the algorithm, we may consider a fictive node A' that has an x-coordinate smaller than that of A ($P_p = A'$). Note, that other fictive nodes can be considered. Figure 3(b) shows the different starting nodes and their fictive neighbors designed by gray points.

Next, we have to find the minimum polar angle formed by the edge $\{P_c, P_p\}$ (i.e., $\{A, A'\}$) and the edges formed by P_c with each of its neighbors (i.e., $\{A, B\}$, $\{A, C\}$, $\{A, D\}$ and $\{A, G\}$). In this example, the obtained neighbor is $P_k = C$ (cf. Figure 3(c)). Then, the first edge of the searched boundary is $\mathbb{B}_E = \{\{A, C\}\}$. Therefore, $\mathbb{B}_V = \{A, C\}$.

In the next iteration, we will apply the same procedure by searching the minimum polar angle formed by the edge $\{C, A\}$ and the edges formed by C with its neighbors, i.e., $\{C, B\}$ and $\{C, D\}$. The obtained node is D (cf. Figure 3(d)) and the current boundary is now given by $\mathbb{B}_V = \{A, C, D\}$ with $\mathbb{B}_E = \{\{A, C\}, \{C, D\}\}$. In the same way, we determine the other nodes. We found E (cf. Figure 3(e)), H (cf. Figure 3(f)), G (cf. Figure 3(g)) and finally A (cf. Figure 3(h)). Since $A = P_{first}$, we stop the algorithm. The obtained boundary is given by $\mathbb{B}_V = \{A, C, D, E, H\}$ and $\mathbb{B}_E = \{\{A, C\}, \{C, D\}, \{D, E\}, \{E, H\}, \{H, A\}\}$ (cf. Figure 3(i)).

This algorithm does not work in the presence of the following cases:

- Nodes of degree one,
- Anchor and Boat graphs (Figures 5 and 10).

These two cases may lead to a situation of an infinite loop. In the next section, we will describe all situations leading our first algorithm to fail, and we will show how to avoid them.

IV. Problematic subgraphs and LPCN2

LPCN1 presented as Algorithm 2 in the previous Section III-B is only working correctly when the considered graph does not contain some problematic subgraphs which may lead either to a blocking step or to a non-optimal solution.

1) Case 1: zero degree angle: To find the next node on the polygon hull, angles are calculated between edges $\{P_i, P_{i-1}\}$ and $\{P_i, P_{i+1}\}$, with $P_{i+1} \in N(P_i)$, and the node for which the calculated angle is the smallest one will be chosen as the next one. Since the previous node is among the neighbors of the current node, it will be the one to be chosen because the angle formed by the corresponding edges is equal to zero. This leads to a blocking situation. To overcome this limitation, we will consider this angle to be 360^o instead of 0^o. The same solution can be used for the case of nodes having one neighbor only. Figure 4 shows this situation. By taking account of the proposed modification, the obtained boundary is then given by $\mathbb{B}_V = \{A, B, C, E, D\}$ and $\mathbb{B}_E = \{\{A, B\}, \{B, C\}, \{C, E\}, \{E, C\}, \{C, D\}, \{D, A\}\}$.

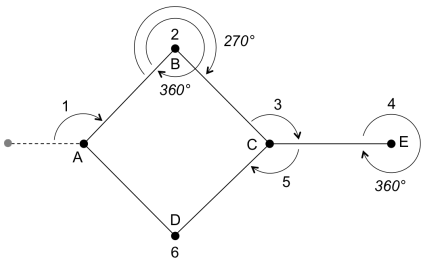

Fig. 4. From 0^o to 360^o angles.

2) Case 2: Boat graph: Figure 5 shows an example of a *Boat Graph* formed by the edges $\{A, C\}$, $\{A, D\}$ and the triangle BCD. A *Generalized Boat Graph* is obtained in the same way if we consider a subdivision of $\{A, D\}$.

When we execute the LPCN1 algorithm presented above by starting from or arriving at point B then some problematic situations will arise.

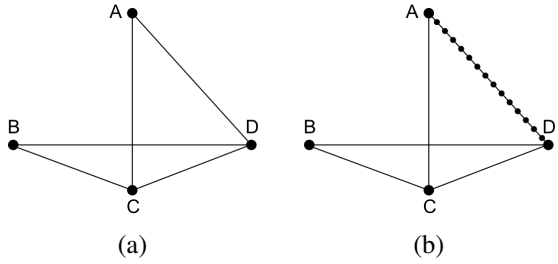

Fig. 5. Boat Graph with possible nodes between A and D.

To explain how this kind of graphs can be disturbing, let us take two examples with *Boat Graphs*. Figure 6 shows the first example of a boat graph which is formed by the edges $\{E, C\}$, $\{E, D\}$ and the triangle BCD. The algorithm starts from node B and then goes to D. From D it goes to E and then to C. From C it goes again to D. However, this time it will not go to E but to B again. Because we start from the node B the algorithm will stop and, unfortunately, it will not visit the edge $\{E, F\}$ nor the nodes and edges that are between F and B. Hence, in this case, the polygon hull cannot be found.

Figure 7 shows the second example where the algorithm starts with the left node A and then goes to B, which is a starting node of a Boat graph formed by the edges $\{E, C\}$, $\{E, D\}$ and the triangle BCD. From B the algorithm goes to D, to E and then to C. From C it goes again to D. From D, this time, it goes to B and then to C and to E again. Finally, it continues to F and to the other nodes. In this case, there is no blocking situation but the solution is not optimal. It is given by $\mathbb{B}_V = \{A, B, D, C, E, F\}$ and $\mathbb{B}_E = \{\{A, B\}, \{B, D\}, \{D, E\}, \{E, C\}, \{C, D\}, \{D, B\}, \{B, C\}, \{C, E\}, \{E, F\}\}$ with 6 nodes and 9 edges whereas the optimal solution given by $\mathbb{B}_V = \{A, B, D, E, F\}$ and $\mathbb{B}_E = \{\{A, B\}, \{B, D\}, \{D, E\}, \{E, F\}\}$ has 5 nodes and 4 edges.

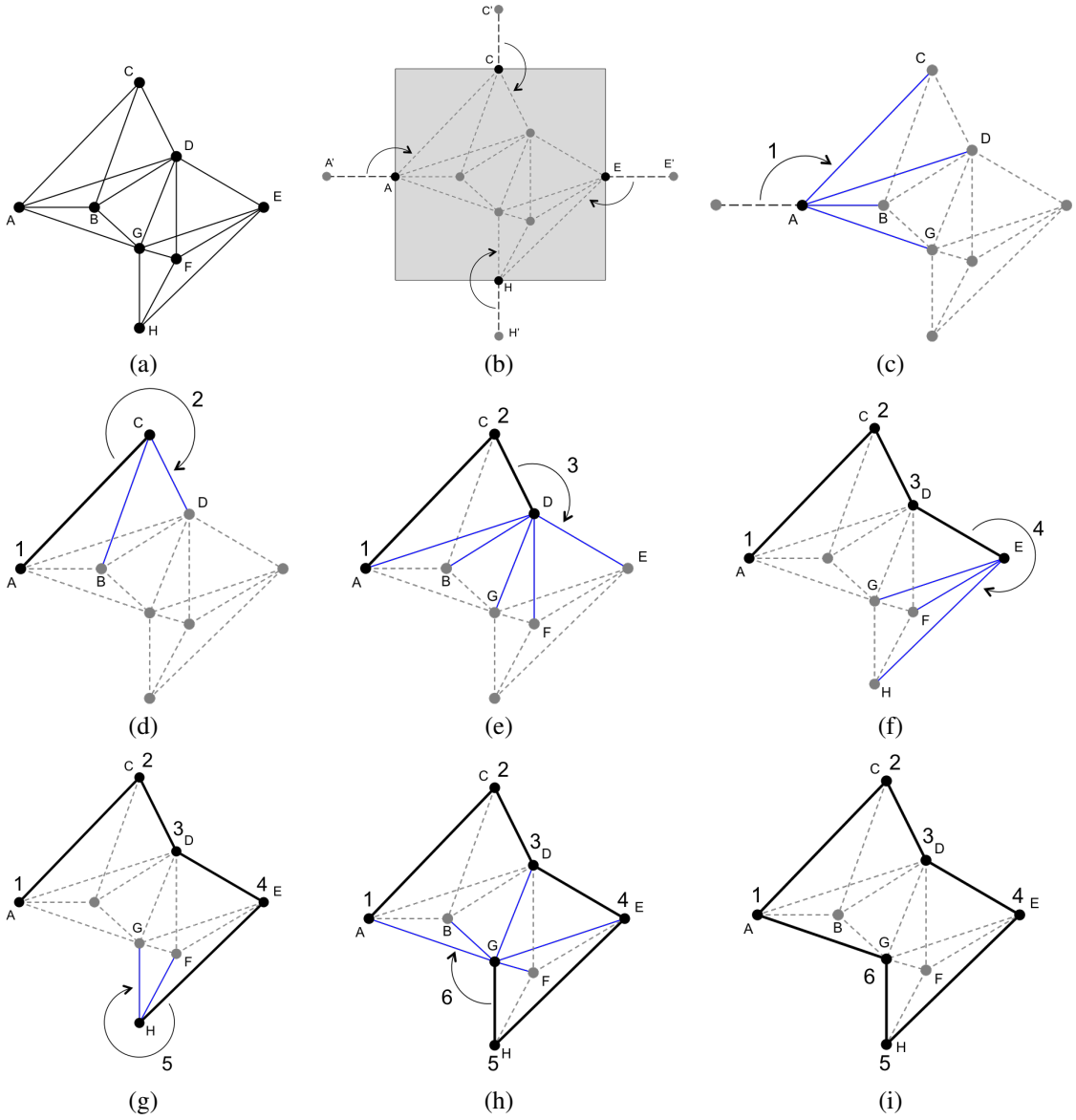

Fig. 3. LPCN algorithm illustration.

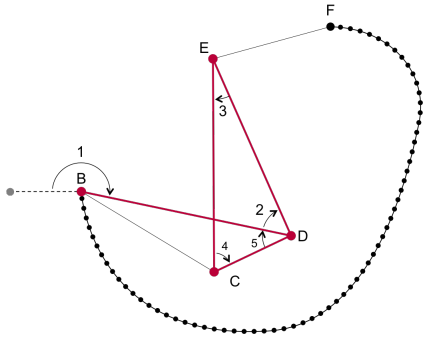

Fig. 6. Boat graph (example 1).

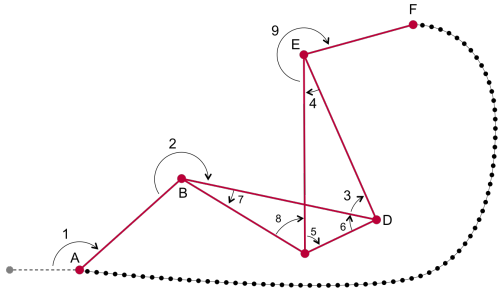

Fig. 7. Boat graph (example 2).

The previous solution can be improved by eliminating the edge $\{E, C\}$ which is justified by its intersection with the edge $\{B, D\}$ which itself leads to a point C inside the boundary $\{A, B, D, E, F\}$, as shown by the example of Figure 8. This

last boundary will be considered as an improved solution with respect to the previous one. Then, this solution is optimal and there is no better solution since B is not connected to E.

We can note that if distances must be taken into account and the total distance has to be minimized then we can take the solution $\mathbb{B}_V = \{A, B, C, E, F\}$ if the length of the path $[B, D, E]$ is shorter than the one of the path $[B, C, E]$. Note, that in this case, the rule of the least polar angle is not respected.

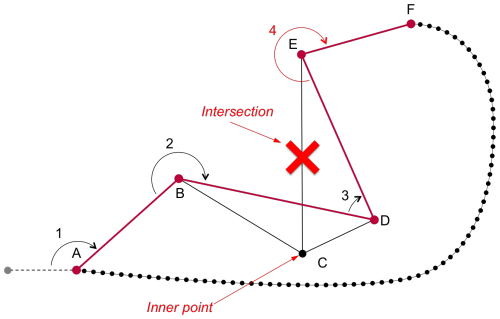

Fig. 8. Boat graph (solution).

We conclude from this first case that any edge of the polygon hull must not intersect with any other [2].

3) Case 3: Intersecting edges: We conclude from Case 2 that two edges of the polygon hull must not intersect outside the endpoint corresponding to the next vertex chosen by the algorithm. As an example, Figure 9(a) shows that accepting the intersecting edges $\{A, B\}$ and $\{E, F\}$ of the polygon hull will lead to non-visited vertices on the polygon hull, which are I , J and G. However, if this intersection is considered then, as shown by Figure 9(b), all the vertices of the polygon hull are visited (i.e., A, B, C, D, E, G, H, I, J). In addition, since the next chosen point is B, this intersection must not consider the endpoint B of the edge $\{D, B\}$. That is to say, if we have two edges $\{A, B\}$ and $\{C, D\}$ and if the intersection with B results in one of the vertices A, B, C or D then it will not be considered as an intersection. This situation is justified by Figures 9(c) and (d). In the first one, if we accept the normal intersection $\{B, C\} \cap \{D, B\} = \{B\}$, the algorithm will choose a vertex which is different from B. Then, it will choose the next vertex C, which will lead to an infinite loop B, C and D. However, if we consider intersection without the endpoints of the edges, then when the algorithm comes back to B, no intersection is detected and the algorithm will choose vertex A as shown by Figure 9(d).

4) Case 4: Anchor graph: Figure 10 shows an example of an *Anchor Graph* formed by the edge $\{A, C\}$ and the triangle BCD. It just differs from the *Boat Graph* by the missing edge $\{A, D\}$. Executing the LPCN1 algorithm presented above in graphs containing an Anchor graph my lead to a non-optimal solution as we are going to show now.

As a first example, let us consider the case where the graph is itself an Anchor Graph. If the algorithm starts from the point B, the solution that will be found by our algorithm is

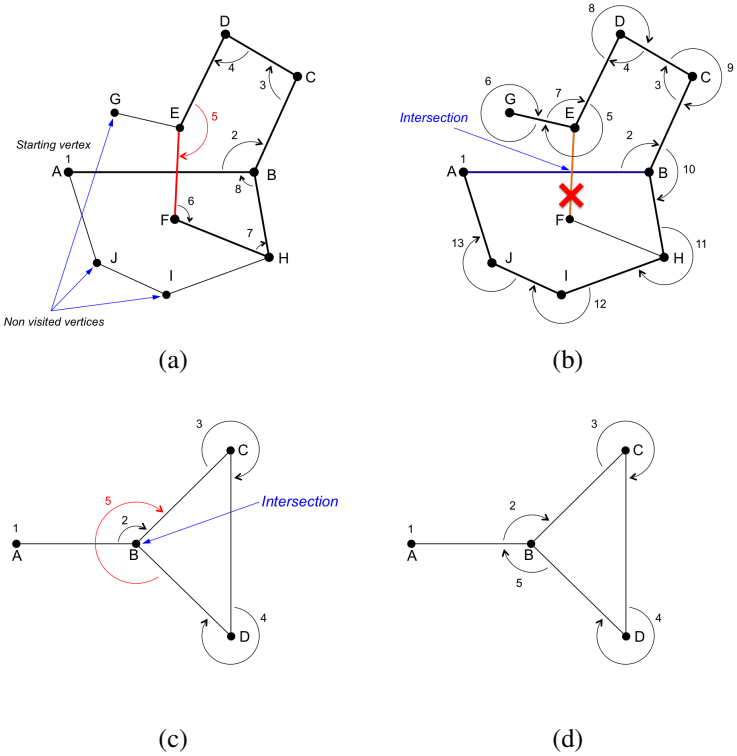

Fig. 9. Edges' intersection.

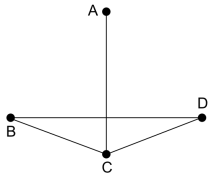

Fig. 10. Anchor graph.

$\mathbb{B}_V = \{B, D, C\}$ and $\mathbb{B}_E = \{\{B, D\}, \{D, C\}, \{C, B\}\}$. One can see that in this solution, as illustrated by Figure 11(a), point A cannot be reached. However, if we start from point A, as shown by Figure 11(b), it can be reached. Note, that the edge selected at one iteration does not have to intersect with the edges of the polygon hull obtained in previous iterations.

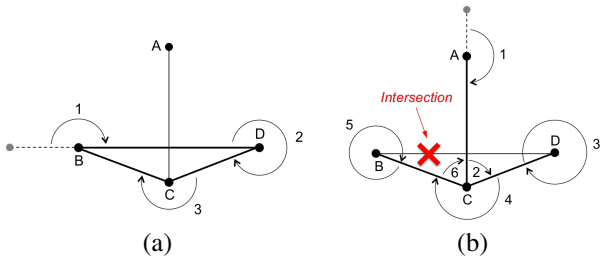

Fig. 11. Anchor graph (problematic situation 1).

A second example related to an Anchor Graph is shown by Figure 12. We can see that if we start from node H, A cannot be reached. But if we start from node A, Figure 13 shows that

it is possible to find the correct polygon hull coming back to A. We conclude from this situation that the algorithm must start from all the boundary nodes having one neighbor.

Just note that at iteration 3, we will not go from node D to node B because edge $\{D, B\}$ intersects with edge $\{A, C\}$. In this case, our solution is obviously correct. However, if on the boundary there is another *Anchor Graph*, as shown by Figure 14, then node J cannot be reached by the algorithm. If another part of the graph starts from the node J as shown by Figure 15, then this part will not be reached by the algorithm. We conclude that all vertices of edges that intersect with the edges of the found polygon hull can be reached by running the algorithm another time on the non-visited vertices and by connecting node J to the other part of the anchor.

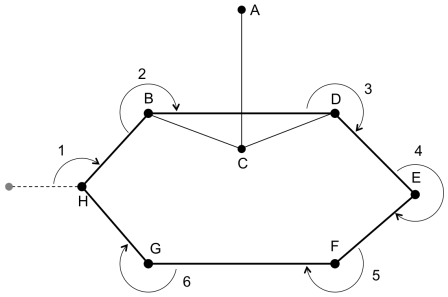

Fig. 12. Anchor graph (problematic situation 2).

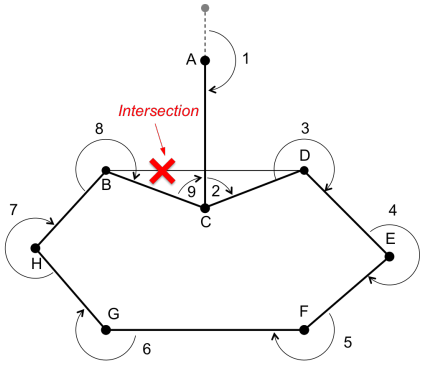

Fig. 13. Anchor graph (problematic situation 3).

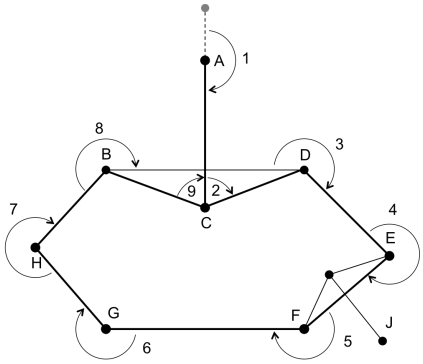

Fig. 14. Anchor graph (problematic situation 4).

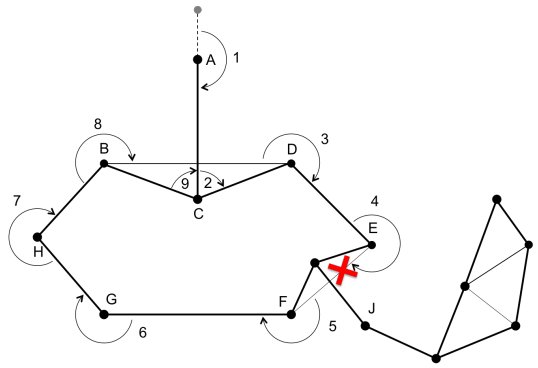

Fig. 15. Anchor graph (problematic situation 5).

5) Case 5: First and last boundary node (Stopping condition): We have previously assumed that the algorithm stops when the last chosen node is equal to the starting node. This means that the algorithm terminates when the first chosen node is selected a second time. However, the two cases presented in Figure 16 (the presence of an *Anchor Graph*) and Figure 17 (the presence of several descending branches from the starting node) show that it is possible to come back to the starting node without visiting all the boundary nodes. As we can see, in the first figure, the set of nodes situated between nodes A and B are not reached by the algorithm. In the second figure, only the boundary nodes of the first descending branch are reached. Hence, the stopping condition is not based on the starting node but on the second visited boundary node, since each node cannot select the same minimum angle twice, as is proven by Corollary 1 in Section V.

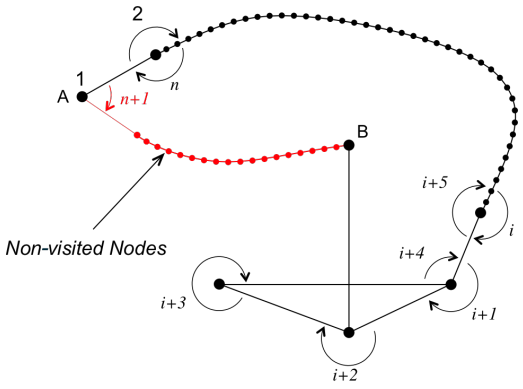

Fig. 16. First and last boundary node (Stopping condition): case 1.

6) Case 6: Three points on the same line: If in addition, the three points of the triangle of an anchor graph are situated on the same line (cf. Figure 18) then from point B both points D and C can be chosen. In this case, the point with the minimum distance must be chosen. In the case of Figure 18, for instance, point C must be selected. However, if these three points do not belong to an anchor graph then the point with the maximum distance must be chosen in order to guarantee the minimum cardinality of $\mathbb{B}_V$. This situation is illustrated by Figure 19

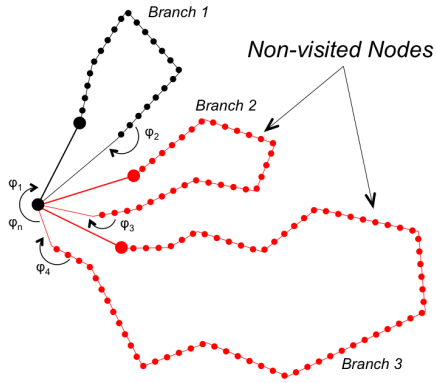

Fig. 17. First and last boundary node (Stopping condition): case 2

where in (a), when the vertex C is chosen, four boundary vertices are obtained. Nevertheless, in (b), only three boundary vertices are chosen if the vertex D is directly chosen from B, because it is further away from B than C from B.

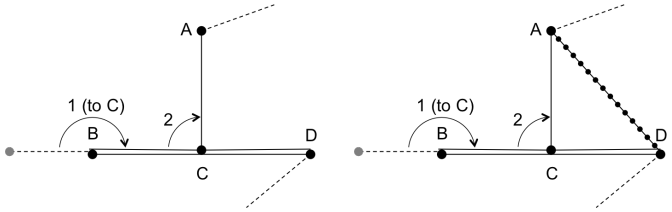

Fig. 18. Anchor graph (a) and Boat graph (b) with three points of the triangle on the same line.

Fig. 19. Three points on the same line.

We will complete this section by discussing another example dealing with the *Anchor Graph*. Figure 20 shows this example. As we can see, the obtained walk is unfortunately not closed. Geometrically, this solution is not correct, but if we assume that the objective is just to visit the vertices of a polygon hull instead of finding its geometric form, this solution can be considered acceptable and therefore, the algorithm LPCN2 (Algorithm 3) can be used in this case.

However, if we want to find the geometric polygon, we will consider the solution indicated in Figure 21. In this case, LPCN2 must be modified as follows. We execute the same iterations from 1 to 8 as in Figure 20. The next node will be C which leads to an edge $\{A, C\}$ that intersects with the boundary edge $\{B, D\}$. Now, instead of eliminating this edge and searching for another edge as in the previous algorithm,

Algorithm 3 LPCN2: the second version of the LPCN algorithm.

1: **procedure** LPCN(V, E)
2: $P_0 \leftarrow P_c \leftarrow$ point having the minimum x-coordinate
3: $P_p \leftarrow$ fictive point situated in the left of P_c
4: $\mathbb{B}_V \leftarrow \{P_c\}$
5: $\mathbb{B}_E \leftarrow \emptyset$
6: $once \leftarrow true$
7: **repeat**
8: $\mathbb{A} \leftarrow \{P \in N(P_c)/\mathbb{B}_{\mathbb{E}} \cap \{\{P_c, P\}\} = \emptyset\}$
9: $P_{min} \leftarrow \underset{P \in \mathbb{A}}{\text{argmin}}\{\varphi(P_p, P_c, P)\}$
10: $\mathbb{B}_V \leftarrow \mathbb{B}_V \cup \{P_{min}\}$
11: $\mathbb{B}_E \leftarrow \mathbb{B}_E \cup \{\{P_c, P_{min}\}\}$
12: $P_p \leftarrow P_c$
13: $P_c \leftarrow P_{min}$
14: **if** $(once = true)$ **then**
15: $once \leftarrow false$
16: $P_{first} \leftarrow P_{min}$
17: **end if**
18: **until** $((P_c = P_0)$ and $(P_{min} = P_{first}))$
19: **return** $\mathbb{B}_V, \mathbb{B}_E$
20: **end procedure**

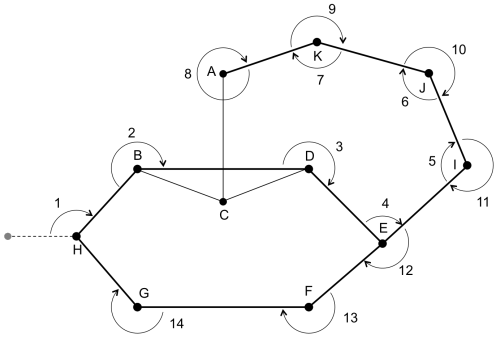

Fig. 20. A special case involving an Anchor graph (solution 1).

we test whether these two edges form an *Anchor Graph*. If this is the case, we will go back to the first node of this *Anchor Graph*, in this case node B. And then, we will choose node C of the *Anchor Graph* instead of node D. Finally, we will continue with iteration 9 as shown in Figure 21. Indeed, this will add an insignificant complexity in case we consider an application from the field of WSNs, where this situation is very rare.

We finish this section with some additional problematic subgraphs that we call *Pseudo-Boat Graph* and *Pseudo-anchor Graph* as shown by Figures 22(a), 22(b), 22(c) and 22(d). These graphs lead to the same results as those presented above for the *Boat Graph* and the *Anchor Graph*. However, if we replace the *Anchor Graph* of Figure 20 by the *Pseudo-anchor Graph* of Figure 22(b) then neither the geometric polygon hull nor the solution shown by Figure 21 can be obtained (cf. Figure 23). However, for this case it is possible to consider the solution of Figure 20.

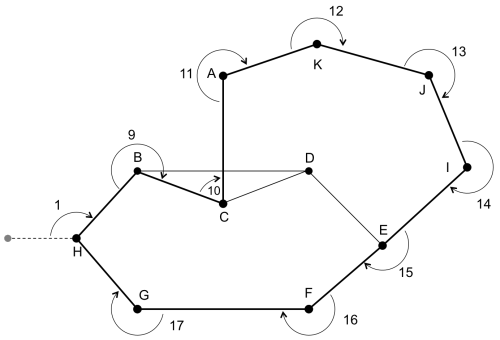

Fig. 21. A special case involving an Anchor graph (solution 2).

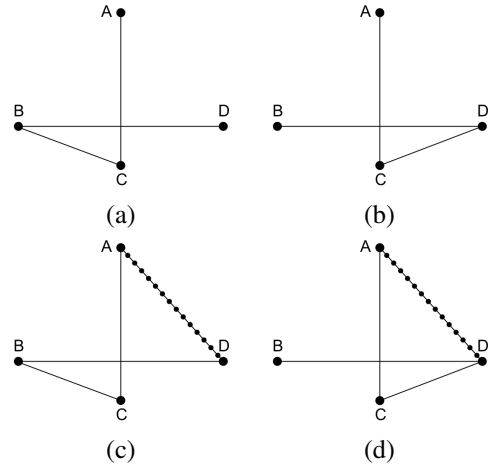

Fig. 22. Pseudo-anchor ((a), (b)) and Pseudo-Boat ((c), (d)) graphs.

V. Algorithm validation

In this section, we will validate the correctness of the proposed algorithm regarding convergence and optimality.

Theorem 1.

1) *The Algorithm $LPCN1$ never calculates the same polar angle more than once.*
2) *The Algorithm $LPCN2$ never calculates the same polar angle more than once.*

Proof.

1) We assume that the algorithm starts from the vertex P_{first} and at iteration i, without loss of generality, we can represent the current set of boundary nodes $\mathbb{B}_V^i = \{P_{first}, ..., P_i\}$ by a vertex $P_{\mathbb{B}_V}^i$. On the other hand, we assume that angles are calculated only in a clockwise direction. Hence, any angle, as given by two specific edges, will be calculated only once. Otherwise, if an angle is calculated a second time then it must pass a second time through the vertex $P_{\mathbb{B}_V}^i$. This means that we pass a second time through the starting vertex P_{first} which represents the stopping condition of LPCN1 (cf. line 12 of Algorithm 2). This leads to a contradiction and therefore it is not possible to calculate any angle more than once.
2) For planar graphs, $LPCN2$ behaves in the same manner as $LPCN1$. For non-planar graphs and in the case where edge crossing is detected (cf. Figure 9), the condition added in line 9 of the LPCN2 (Algorithm 3), prevents the algorithm to return to an already visited angle.

□

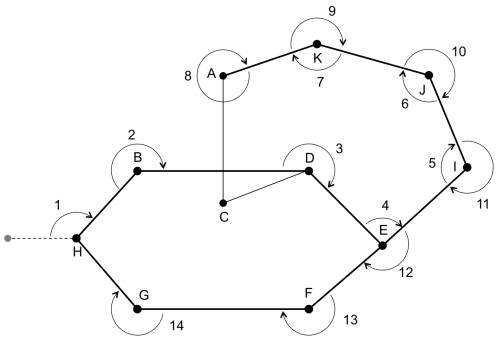

Fig. 23. A special case involving a pseudo-anchor graph.

Corollary 1.

1) *Either Algorithm $LPCN1$ or Algorithm $LPCN2$ never selects the next boundary node from the same boundary node more than once.*

Proof.

1) This is a direct consequence of Theorem 1, since any boundary node is selected based on the calculation of an angle. Assume, that the point v is selected twice from the point v_c by calculating the minimum angle $\varphi(v_p, v_c, v)$. This means that this angle is visited twice, whichi contradicts Theorem 1.

□

Corollary 2.

1) *The algorithm $LPCN1$ visits the point v_i at most $d(v_i)$ times.*
2) *The algorithm $LPCN2$ visits the point v_i at most $d(v_i)$ times.*

Proof.

1) Any vertex $v_i \in V$ has $d(v_i)$ polar angles. Since $LPCN2$ calculates an angle at a given node at most once, by Theorem 1 the number of times the algorithm scans node v_i cannot exceed the number of polar angles. Therefore, the algorithm visits any vertex v_i at most $d(v_i)$ times.
2) In the absence of crossing edges (i.e., if G is planar), $LPCN1$ behaves in the same manner as $LPCN2$.

□

Corollary 3. *Either $LPCN1$ or $LPCN2$ finds a solution in a finite number of steps.*

Proof. The proof results directly from Theorem 1. In fact, since the graph is connected and since it contains a finite number of nodes to form a finite number of angles which are calculated at most once by $LPCN1$ and $LPCN2$, by Theorem 1 the number of iterations of both algorithms is finite. □

Theorem 2. *Either $LPCN1$ or $LPCN2$ finds a polygon hull with a minimum number of nodes, without encountering any Anchor Graph.*

Proof. Let us consider the solution given by $\mathbb{B}_V$. If this solution is not optimal, then there is another polygon hull $\mathbb{B}'_V$ containing less vertices than $\mathbb{B}_V$. Therefore:

$$\mathbb{B}_V = \mathbb{B}'_V \cup \chi$$

For a vertex $v \in \chi$ two cases are possible:

- Case 1: v is not a boundary vertex. This case is impossible because v must satisfy the condition $v = \underset{P_j \in N(P_c)}{\arg\min} \{\varphi(P_p, P_c, P_j)\}$ since it is an element of $\mathbb{B}_V$, where P_c is a neighbor vertex of v. Thus, v is also a boundary vertex.
- Case 2: v is a boundary vertex. Since, v is not an element of $\mathbb{B}'_V$, this latter is not a polygon hull of the network.

Therefore, the set $\mathbb{B}_V$ represents a polygon hull of G with a minimum number of nodes. □

From Corollary 3, we deduce that the two proposed algorithms are convergent. From Theorem 2, we conclude that our proposed algorithms determine a boundary with minimum number of nodes, which altogether validates our algorithms.

VI. Simulation results

Let us consider a graph with n vertices, and a boundary of h vertices, and let k denote the maximum degree of G. Then the complexity of LPCN1 is $O(gh)$. For the case of LPCN2, since in each iteration we are searching for an intersection with the edges of the polygon found in previous iterations, we have to include a factor of h. Thus, the complexity of this algorithm is $O(gh^2)$.

All algorithms have been programmed in Java and implemented in *CupCarbon* [26], a software simulator of Wireless Sensor Networks (WSNs). This software offers an API that facilitates the development of algorithms and the visualization of their results in a realistic WSN environment. It offers the possibility to simulate mobiles and targets. The source code is available in [1].

In the following, we will discuss three test examples for the proposed algorithm LPCN2. The first example represents a random network with 100 points (cf. Figure 24(a)). The points represent sensors with a radio range of $100m$ (meters) that are located in a rectangular area of $1200m \times 600m$. In the second example, we use a network with 500 points (cf. Figure 25) that are located in a rectangular area of $2500m \times 1500m$. In the third example, we have 1500 points (cf. Figure 25) under the same conditions as for the second one. The number of the obtained nodes on the boundary is 62 in the first network, 202 in the second and 85 in the third one. The maximum degree is 17 in the first network, 10 in the second and 50 in the third one. The theoretical complexity in each example is, respectively, $O(17{\times}62^2) = O(65k)$, $O(10{\times}202^2) = O(408k)$ and $O(50 \times 85^2) = O(361k)$. The real number of iterations for each example is, respectively, $9k$, $63k$ and $53k$.

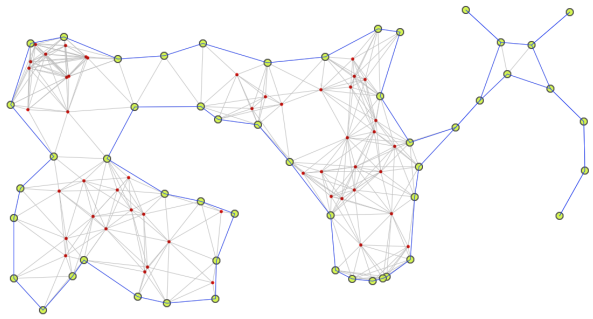

Fig. 24. Example with 100 points.

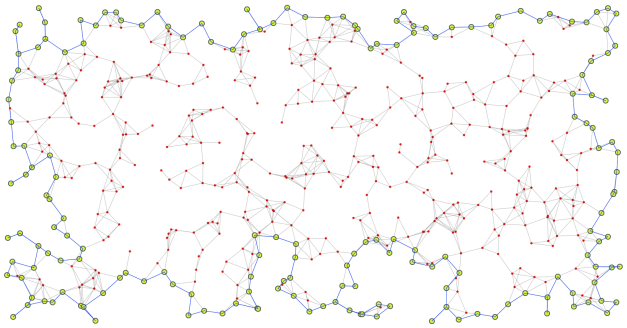

Fig. 25. Example with 500 points.

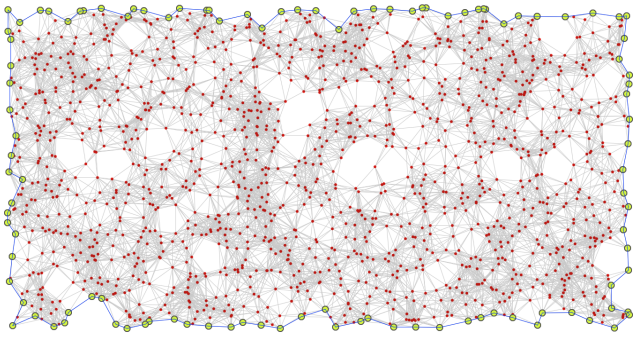

Fig. 26. Example with 1500 points.

VII. Some Applications

In this section, we will present three examples for which the presented algorithm can be used. The first example shows how to find boundary nodes of a Wireless Sensor Network. The second example shows how to extract complex clusters in a set of two-dimensional data. The last one shows how to draw a contour of a zone of interest on a medical image.

A. Finding the boundary nodes of a Wireless Sensor Network

Finding the boundary nodes of a WSN can be done in two different ways: centralized and distributed. In the following, we will present the centralized version since the algorithm is applied as presented in this paper. However, in the distributed version, the procedure is completely different and it is presented in [36]. In the centralized version, as shown by Figure 27(a), each node of the network sends its coordinates to the sink. The sink will then run the LPCN algorithm to find the boundary nodes. Once done, it sends to each sensor node the information whether it is or not a boundary node (cf. Figure 27(b)).

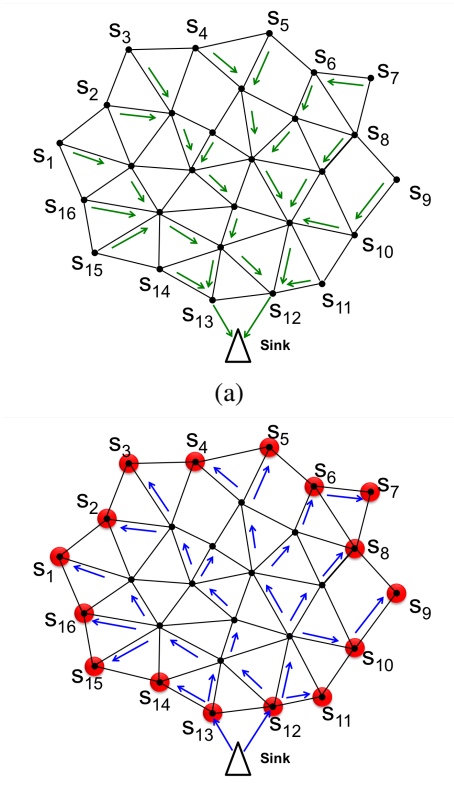

Fig. 27. Boundary nodes of a WSN.

B. Cluster finding and shape reconstruction

To find the clusters of a set of data, we must first connect the points between them. To do this, many existing methods can be used, like Voronoi diagrams [4], k-nearest neighbors, neighbors situated within a certain radius, etc. Second, we start from the point having the smallest x-coordinate and run the LPCN until returning to this starting point. This means that the first cluster is found, which is defined by all the points of the polygon hull and all the points that are inside it. In order to find the second cluster, we must remove from the set of data all the points of the previously found cluster and then restart the same procedure with the remaining points. This procedure is renewed until all clusters have been found. Figure 28 shows an example of a 2-dimensional dataset (cf. Figure 28(a)), the connection between points (Figure 28(b)) and finally the clusters found (cf. Figure 28(c)). The same methodology can be used for shape reconstruction. Since the complexity of the LPCN algorithm depends on the number of the boundary points, it can be used fors huge datasets, which makes it very useful in the context of Big Data.

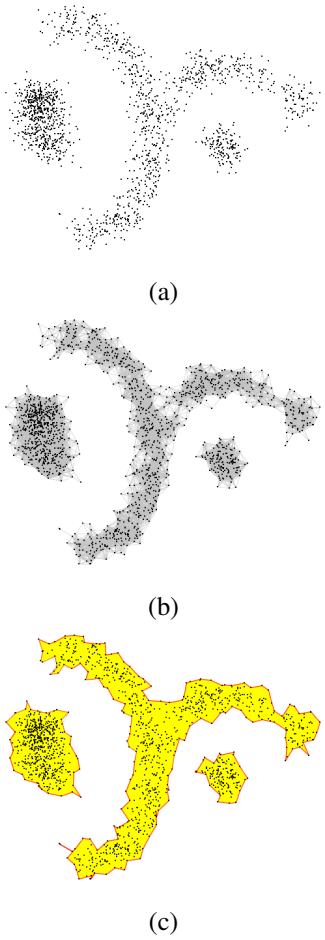

Fig. 28. Finding clusters.

C. Contour drawing

The LPCN algorithm can be used for drawing contours in images. In the case of zone-of-interest-extraction, it has to be combined with another algorithm that allows to characterize such a zone. For illustration, let us consider the extraction of the gray zone of the image of Figure 29(a). First, we will transform the region containing this zone to a matrix of

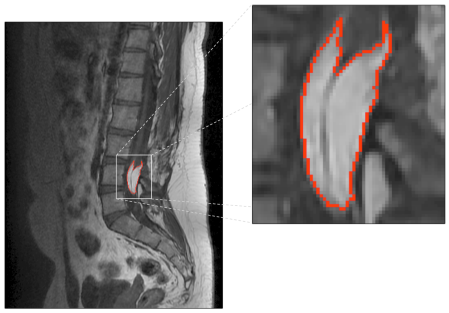

Fig. 30. Contour of a tumor in a real medical image.

pixels. This matrix can be modeled as a Euclidean connected graph as shown by Figures 29(b) and (c). If we consider a simple extraction based on the intensity of pixels, then we will consider only the blue points of the graph, shown by Figure 29(d), which correspond to the gray pixels in the image. For this graph, we will run the LPCN algorithm in order to find the boundary points (cf. Figure 29(e)). Figures 29(f) and (g) show that these points represent the contour of the gray zone of the original image. As a more realistic example, Figure 30 shows the contour of a tumor drawn by using the LPCN algorithm. This picture represents a tumor MRI (Magnetic Resonance Imaging) taken at the hospital of Brest.

VIII. Conclusion

We have presented the LPCN algorithm allowing to find a polygon hull for a Euclidean connected graph. This algorithm was inspired by Jarvis' algorithm which had to be adapted because it is designed to find a convex hull instead of a polygon hull. We have proposed a new algorithm that chooses for each node the nearest polar angle node with respect to the node found in the previous iteration. Its complexity is $O(gh^2)$, where g is the maximum degree of the graph and h the number of nodes on the polygon hull. We have shown that running the algorithm in the presence of specific graph structures can lead to non-valid and non-optimal solutions, and we have indicated how to overcome these difficulties. Also, we have proven the convergence of the algorithm and the optimality of the solution. Finally, we have presented some applications that can be resolved using the proposed algorithm. We are now working on the version where the algorithm can start from any point of the graph instead of the point having minimum x-coordinate, and we are also preparing a 3-dimensional version.

IX. Acknowledgment

This work is part of the research project PERSEPTEUR supported by the French Agence Nationale de la Recherche ANR.

References

[1] *CupCarbon: A Smart City and IoT Wireless Sensor Network Simulator*, 2016. http://www.cupcarbon.com.

[2] *LPCN: Least Polar-angle Connected Node (youtube movie)*, 2016. https://www.youtube.com/watch?v=2tC_bA832TE.

[3] Selim G Akl. Two remarks on a convex hull algorithm. *Information Processing Letters*, 8(2):108–109, 1979.

[4] Harith Alani, Christopher B. Jones, and Douglas Tudhope. Voronoi-based region approximation for geographical information retrieval with gazetteers. *International Journal of Geographical Information Science*, 15(4):287–306, 2001.

[5] Alex M Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[6] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

[7] Christian Braune, Marco Dankel, and Rudolf Kruse. Obtaining shape descriptors from a concave hull-based clustering algorithm. In *International Symposium on Intelligent Data Analysis*, pages 61–72. Springer, 2016.

[8] A Ray Chaudhuri, Bidyut Baran Chaudhuri, and Swapan K Parui. A novel approach to computation of the shape of a dot pattern and extraction of its perceptual border. *Computer Vision and Image Understanding*, 68(3):257–275, 1997.

[9] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational geometry*. Springer, 2000.

[10] Herbert Edelsbrunner, David Kirkpatrick, and Raimund Seidel. On the shape of a set of points in the plane. *Information Theory, IEEE Transactions on*, 29(4):551–559, 1983.

[11] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL, the computational geometry algorithms library. 1998.

[12] Jingfan Fan, Jian Yang, Mahima Goyal, and Yongtian Wang. Rigid registration of 3-d medical image using convex hull matching. In *Bioinformatics and Biomedicine (BIBM), 2013 IEEE International Conference on*, pages 338–341. IEEE, 2013.

[13] Ivor D. Faux and Michael J. Pratt. *Computational geometry for design and manufacture*. Ellis Horwood Ltd, 1979.

[14] Gautam Garai and B. B. Chaudhuri. A split and merge procedure for polygonal border detection of dot pattern. *Image and Vision Computing*, 17(1):75–82, 1999.

[15] A. Gheibi, M. Davoodi, A. Javad, F. Panahi, M. M. Aghdam, M. Asgaripour, and A. Mohades. Polygonal shape reconstruction in the plane. *IET Computer Vision*, 5(2):97–106, March 2011.

[16] Abel JP Gomes. A total order heuristic-based convex hull algorithm for points in the plane. *Computer-Aided Design*, 70:153–160, 2016.

[17] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Process. Lett.*, 1(4):132–133, 1972.

[18] Ray A Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, 1973.

[19] MA Jayaram and Hasan Fleyeh. Convex hulls in image processing: A scoping review. *American Journal of Intelligent Systems*, 6(2):48–58, 2016.

[20] Michael Kallay. The complexity of incremental convex hull algorithms in r^d. *Information Processing Letters*, 19(4):197, 1984.

[21] Marco Körner, Mahesh V Krishna, Herbert Süße, Wolfang Ortmann, and Joachim Denzler. Regularized geometric hulls for bio-medical image segmentation. *The Annals of the BMVA,(4)*, pages 1–12, 2015.

[22] Farid Lalem, Ahcène Bounceur, Rahim Kacimi, Reinhardt Euler, and Massinissa Saoudi. Faulty data detection in wireless sensor networks based on copula theory. In *Proceedings of the International Conference on Big Data and Advanced Wireless Technologies*, BDAW '16, pages 29:1–29:7, New York, NY, USA, 2016. ACM.

[23] Farid Lalem, Rahim Kacimi, Ahcène Bounceur, and Reinhardt Euler. Boundary node failure detection in wireless sensor networks. In *IEEE International Symposium on Networks, Computers and Communications (ISNCC 2016), 11-13 May, Hammamet, Tunisia*, 2016.

[24] George Leifman, Elizabeth Shtrom, and Ayellet Tal. Surface regions of interest for viewpoint selection. *IEEE transactions on pattern analysis and machine intelligence*, 38(12):2544–2556, 2016.

[25] Yali Li, Shengjin Wang, Qi Tian, and Xiaoqing Ding. A survey of recent advances in visual feature detection. *Neurocomputing*, 149:736–751, 2015.

[26] Kamal Mehdi, Massinissa Lounis, Ahcène Bounceur, and Tahar Kechadi. Cupcarbon: A multi-agent and discrete event wireless sensor network design and simulation tool. *In IEEE 7th International Conference*

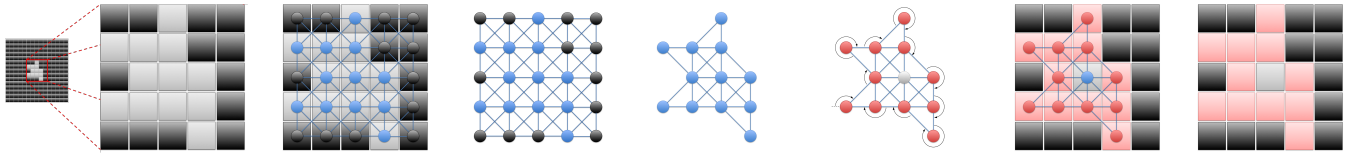

Fig. 29. Contour of a zone of interest.

on Simulation Tools and Techniques (SIMUTools'14), Lisbon, Portugal, March 17-19 2014.

[27] Gang Mei. Cudachain: an alternative algorithm for finding 2d convex hulls on the gpu. *SpringerPlus*, pages 1–26, 2016.

[28] Avraham A. Melkman. On-line construction of the convex hull of a simple polyline. *Inf. Process. Lett.*, 25(1):11–12, April 1987.

[29] Subhasree Methirumangalath, Amal Dev Parakkat, and Ramanathan Muthuganapathy. A unified approach towards reconstruction of a planar point set. *Computers & Graphics*, 51:90–97, 2015.

[30] Adriano Moreira and Maribel Yasmina Santos. Concave hull: A k-nearest neighbours approach for the computation of the region occupied by a set of points. 2007.

[31] Ketan Mulmuley. *Computational geometry: An introduction through randomized algorithms*. Prentice Hall, 1994.

[32] Jin-Seo Park and Se-Jong Oh. A new concave hull algorithm and concaveness measure for n-dimensional datasets. *Journal of information science and engineering*, 29(2):379–392, 2013.

[33] Franco P. Preparata and Michael Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[34] E Rosén, E Jansson, and M Brundin. Implementation of a fast and efficient concave hull algorithm. Technical report, University of Uppsala, Sweden, 2014.

[35] Antonio Ruano, Hamid Reza Khosravani, and Pedro M Ferreira. A randomized approximation convex hull algorithm for high dimensions. *IFAC-PapersOnLine*, 48(10):123–128, 2015.

[36] Massinissa Saoudi, Farid Lalem, Ahcène Bounceur, Reinhardt Euler, M-Tahar Kechadi, Abdelkader Laouid, Madani Bezoui, and Marc Sevaux. D-lpcn: A distributed least polar-angle connected node algorithm for finding the boundary of a wireless sensor network. *Ad Hoc Networks Journal, Elsevier*, 56:56–71, March 2017.

[37] Feminna Sheeba, Robinson Thamburaj, Joy John Mammen, Mohan Kumar, and Vansant Rangslang. Convex hull based detection of overlapping red blood cells in peripheral blood smear images. In *7th WACBE World Congress on Bioengineering 2015*, pages 51–53. Springer, 2015.

[38] Vaclav Skala, Zuzana Majdisova, and Michal Smolik. Space subdivision to speed-up convex hull construction in e3. *Advances in Engineering Software*, 91:12–22, 2016.

[39] Godfried T Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern recognition*, 12(4):261–268, 1980.

[40] Changyuan Xing, Zhongyang Xiong, Yufang Zhang, Xuegang Wu, Jingpei Dan, and Tingping Zhang. An efficient convex hull algorithm using affine transformation in planar point set. *Arabian Journal for Science and Engineering*, 39(11):7785–7793, 2014.